



DART

www.postparaprogramadores.com

Contenido

Programación de Dart - Inicio

Programación de Dart: descripción general

Programación de Dart - Medio ambiente

Programación de Dart: sintaxis

Programación de Dart: tipos de datos

Programación de Dart: variables

Programación de Dart: operadores

Programación de Dart: bucles

Programación de Dart: toma de decisiones

Programación de Dart - Números

Programación de Dart: cadena

Programación de Dart - booleano

Programación de Dart - Listas

Programación de Dart - Listas

Programación de Dart - Mapa

Programación de Dart - Símbolo

Programación de Dart - Runas

Programación de Dart - Enumeración

Programación de Dart - Funciones

Programación de Dart: interfaces

Programación de Dart - Clases

Programación de Dart - Objeto

Programación de Dart - Colección

Programación de Dart - Genéricos

Programación de Dart - Paquetes

Programación de Dart: excepciones

Programación de Dart - Depuración

Programación de Dart - Typedef

Programación de Dart - Bibliotecas

Programación de Dart: asíncrono

Programación de Dart - concurrencia

Programación de Dart: pruebas unitarias

Programación de Dart - HTML DOM

Programación de Dart: descripción general

Dart es un lenguaje orientado a objetos con sintaxis de estilo C que opcionalmente puede compilarse en JavaScript. Admite una amplia gama de ayudas de programación como interfaces, clases, colecciones, genéricos y escritura opcional.

Dart se puede utilizar ampliamente para crear aplicaciones de una sola página. Las aplicaciones de una sola página se aplican solo a sitios web y aplicaciones web. Las aplicaciones de una sola página permiten la navegación entre diferentes pantallas del sitio web sin cargar una página web diferente en el navegador. Un ejemplo clásico es **GMail** — cuando hace clic en un mensaje en su bandeja de entrada, el navegador permanece en la misma página web, pero el código JavaScript oculta la bandeja de entrada y muestra el cuerpo del mensaje en la pantalla.

Google ha lanzado una versión especial de **Chromium** : **Dart VM** . El uso de Dartium significa que no tiene que compilar su código en JavaScript hasta que esté listo para probar en otros navegadores.

La siguiente tabla compara las características de Dart y JavaScript.

Característica	Dardo	JavaScript
Sistema de tipo	Opcional, dinámico	Débil, dinámico
Clases	Sí, herencia única	Prototipo
Interfaces	Sí, múltiples interfaces	No
Concurrencia	Sí, con aislamientos	Sí, con trabajadores web HTML5

Este tutorial proporciona un nivel básico de comprensión del lenguaje de programación Dart.

Descarga más libros de programación GRATIS [click aquí](#)

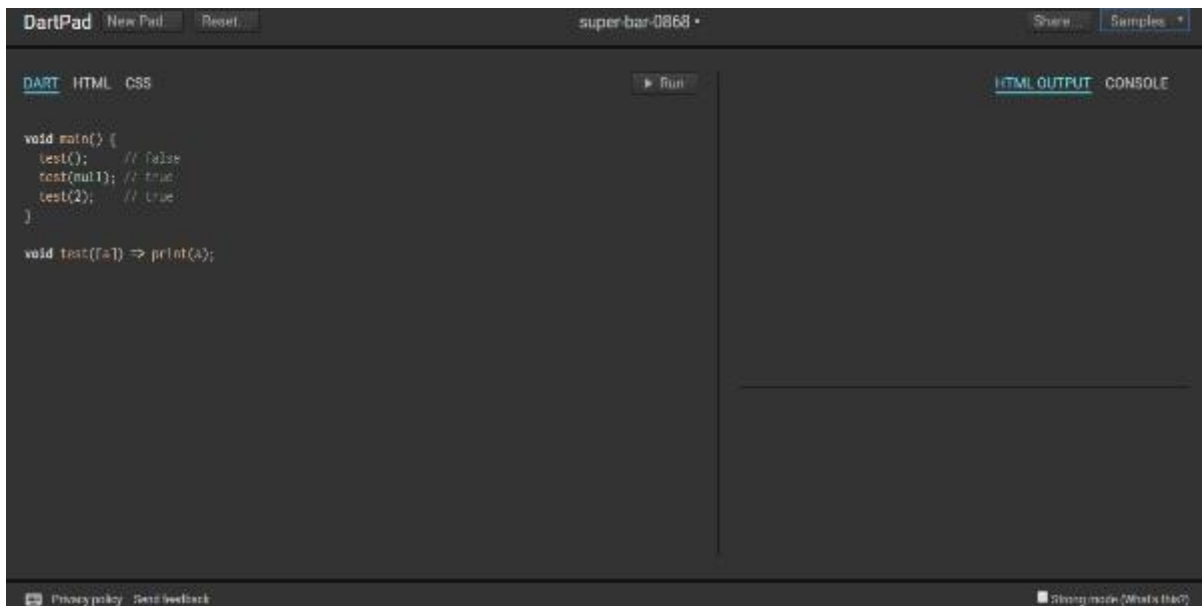
Programación de Dart - Medio ambiente

Este capítulo trata sobre la configuración del entorno de ejecución para Dart en la plataforma Windows.

Ejecutar script en línea con DartPad

Puede probar sus scripts en línea utilizando el editor en línea en <https://dartpad.dartlang.org/> . Dart Editor ejecuta el script y muestra tanto HTML como la salida de la consola. El editor en línea se entrega con un conjunto de ejemplos de códigos preestablecidos.

A continuación se muestra una captura de pantalla del editor **Dartpad** :



Dartpad también permite codificar de manera más restrictiva. Esto se puede lograr marcando la opción Modo fuerte en la parte inferior derecha del editor. El modo fuerte ayuda con -

- Comprobación estática y dinámica más fuerte
- Generación de código JavaScript idiomático para una mejor interoperabilidad.

Puedes probar el siguiente ejemplo usando Dartpad

```
void main() {  
  print('hello world');  
}
```

El código mostrará la siguiente salida

```
hello world
```

Configurando el entorno local

En esta sección, veamos cómo configurar el entorno local.



Síguenos en Instagram para que estés al tanto de los nuevos libros de programación. [Click aqui](#)

Usando el editor de texto

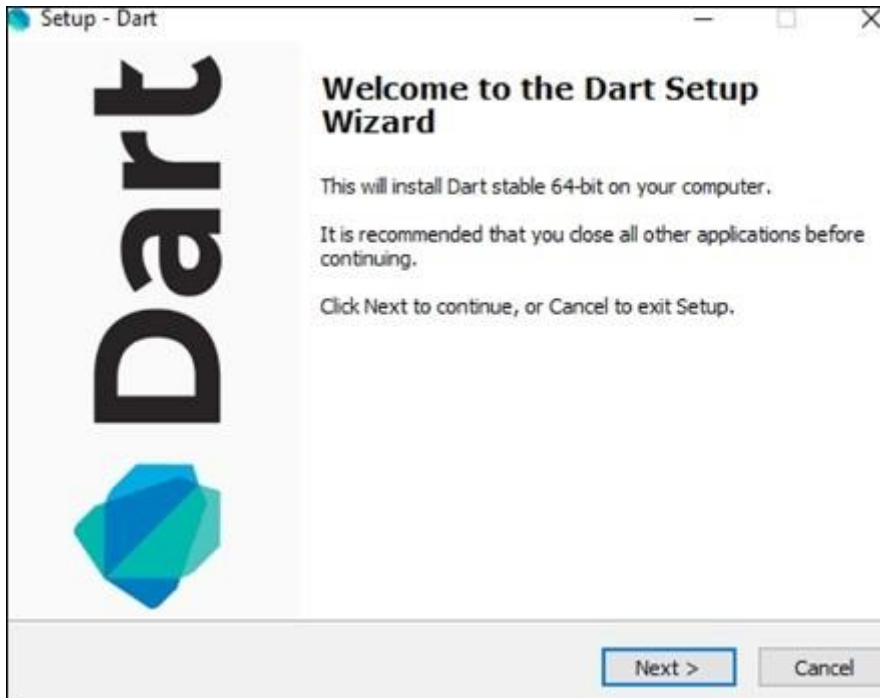
Los ejemplos de algunos editores incluyen Windows Notepad, Notepad ++, Emacs, vim o vi, etc. Los editores pueden variar de un sistema operativo a otro. Los archivos de origen generalmente se nombran con la extensión ".dart".

Instalación del Dart SDK

La versión estable actual de Dart es **1.21.0** . El **dart sdk** se puede descargar desde -

- <https://www.dartlang.org/install/archive>
- <http://www.gekorm.com/dart-windows/>

A continuación se muestra una captura de pantalla de la instalación de Dart SDK:



Al finalizar la instalación del SDK, configure la variable de entorno PATH en -
`<dart-sdk-path>\bin`

Verificando la instalación

Para verificar si Dart se ha instalado correctamente, abra el símbolo del sistema e ingrese el siguiente comando:

```
Dart
```

Si la instalación se realiza correctamente, mostrará el tiempo de ejecución de dart.

Soporte IDE

Una gran cantidad de IDEs admiten scripting en Dart. Los ejemplos incluyen **Eclipse**, **IntelliJ** y **WebStorm** de los cerebros Jet.

A continuación se **detallan** los pasos para configurar el entorno Dart utilizando **WebStorm IDE**.

Instalar WebStorm

El archivo de instalación de WebStorm se puede descargar desde <https://www.jetbrains.com/webstorm/download/#section=windows-version>.

El archivo de instalación de WebStorm está disponible para Mac OS, Windows y Linux.

Después de descargar los archivos de instalación, siga los pasos que se detallan a continuación:

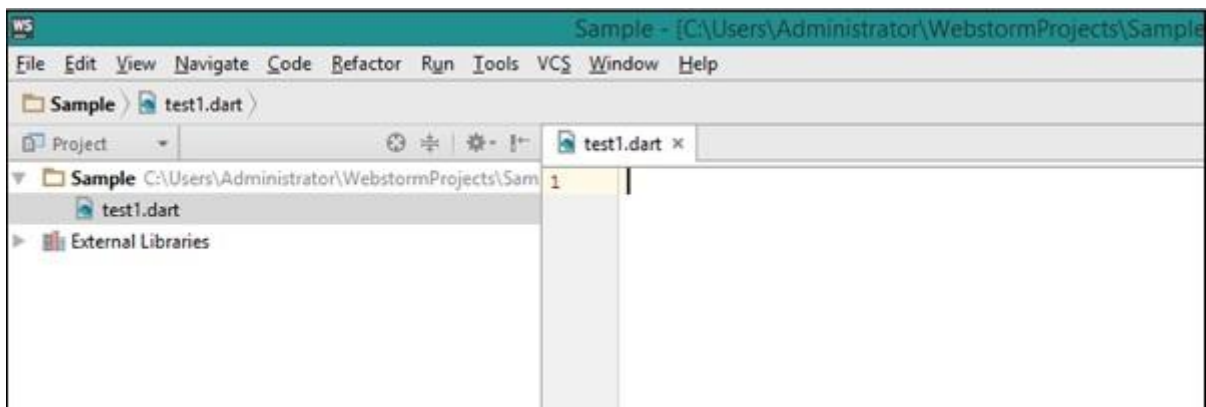
- Instale el SDK de Dart: consulte los pasos enumerados anteriormente
- Cree un nuevo proyecto Dart y configure el soporte de Dart
- Para crear un nuevo proyecto Dart,
 - Haga clic en **Crear nuevo proyecto** desde la pantalla de bienvenida
 - En el siguiente cuadro de diálogo, haga clic en **Dart**
- Si no se especifica ningún valor para la ruta del **SDK de Dart** , proporcione la ruta del SDK. Por ejemplo, la ruta del SDK puede ser **<directorio de instalación de dart> / dart / dartsdk .**

Agregar un archivo Dart al proyecto

Para agregar un archivo Dart al Proyecto:

- Haga clic derecho en el proyecto
- Nuevo → Archivo Dart
- Ingrese el nombre de la secuencia de comandos Dart

A continuación se muestra una captura de pantalla del Editor de WebStorm:



La herramienta dart2js

La herramienta **dart2js** compila el código Dart a JavaScript. Compilar el código Dart en JS permite ejecutar el script Dart en navegadores que no admiten Dart VM.

La herramienta dart2js se envía como parte del Dart SDK y se puede encontrar en la **carpeta / dartsdk / bin** .

Para compilar Dart a JavaScript, escriba el siguiente comando en el terminal

```
dart2js - - out = <output_file>.js <dart_script>.dart
```

Este comando produce un archivo que contiene el equivalente de JavaScript de su código Dart. Puede encontrar un tutorial completo sobre el uso de esta utilidad en el sitio web oficial de Dart.

Programación de Dart: sintaxis

La sintaxis define un conjunto de reglas para escribir programas. Cada especificación de lenguaje define su propia sintaxis. Un programa Dart se compone de:

- Variables y Operadores
- Clases
- Las funciones
- Expresiones y construcciones de programación
- Toma de decisiones y construcciones de bucle
- Comentarios
- Bibliotecas y paquetes
- Typedefs
- Estructuras de datos representadas como colecciones / genéricos

Tu primer código de Dart

Comencemos con el ejemplo tradicional de "Hello World":

```
main() {  
    print("Hello World!");  
}
```

La función **main ()** es un método predefinido en Dart. Este método actúa como el punto de entrada a la aplicación. Un script Dart necesita el método **main ()** para su ejecución. **print ()** es una función predefinida que imprime la cadena o el valor especificado en la salida estándar, es decir, el terminal.

La salida del código anterior será:

```
Hello World!
```

Ejecutar un programa de Dart

Puede ejecutar un programa Dart de dos maneras:

- Por la terminal
- A través del IDE de WebStorm

Por la terminal

Para ejecutar un programa Dart a través del terminal:

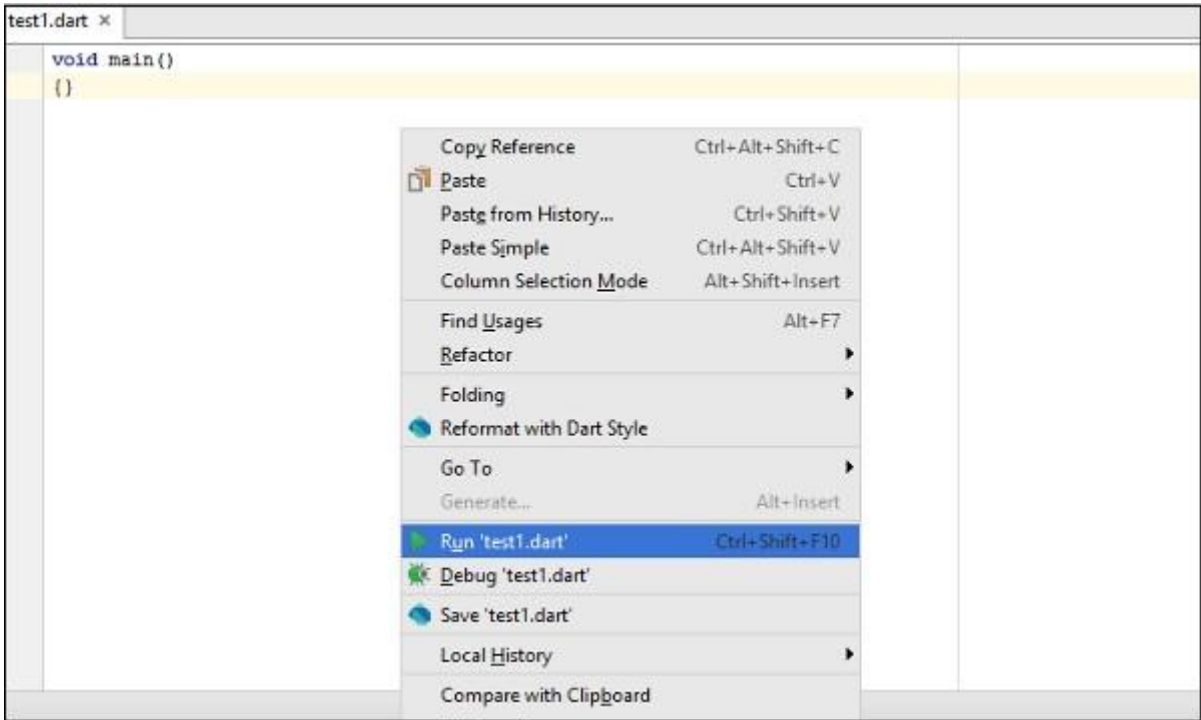
- Navega a la ruta del proyecto actual
- Escriba el siguiente comando en la ventana Terminal


```
dart file_name.dart
```

A través del IDE de WebStorm

Para ejecutar un programa Dart a través del IDE de WebStorm:

- Haga clic derecho en el archivo de script Dart en el IDE. (El archivo debe contener la función **main ()** para permitir la ejecución)
- Haga clic en la opción **'Ejecutar <nombre de archivo>'**. A continuación se muestra una captura de pantalla de la misma:



Alternativamente, se puede hacer clic en el  botón o usar el atajo **Ctrl + Shift + F10** para ejecutar el Dart Script.

Opciones de línea de comandos de Dart

Las opciones de línea de comandos de Dart se utilizan para modificar la ejecución de Dart Script. Las opciones comunes de línea de comandos para Dart incluyen lo siguiente:

No Señor	Opción de línea de comandos y descripción
1	-c o --c Habilita tanto las aserciones como las verificaciones de tipo (modo marcado).
2	--versión Muestra información de la versión de VM.

3	--packages <path> Especifica la ruta al archivo de configuración de resolución del paquete.
4 4	-p <path> Especifica dónde encontrar bibliotecas importadas. Esta opción no se puede usar con --packages.
5 5	-h o --ayuda Muestra ayuda.

Habilitar el modo marcado

Los programas Dart se ejecutan en dos modos, a saber:

- Modo marcado
- Modo de producción (predeterminado)

Se recomienda ejecutar Dart VM en **modo comprobado** durante el desarrollo y las pruebas, ya que agrega advertencias y errores para ayudar al proceso de desarrollo y depuración. El modo marcado aplica varias verificaciones, como la verificación de tipo, etc. Para activar el modo marcado, agregue la opción `-c` o `--` marcada antes del nombre del archivo de script mientras ejecuta el script.

Sin embargo, para garantizar un beneficio de rendimiento al ejecutar el script, se recomienda ejecutar el script en el **modo de producción**.

Considere el siguiente archivo de script **Test.dart** :

```
void main() {
  int n = "hello";
  print(n);
}
```

Ejecute el script ingresando -

```
dart Test.dart
```

Aunque existe una falta de coincidencia de tipos, el script se ejecuta con éxito cuando el modo marcado está desactivado. El script dará como resultado el siguiente resultado:

```
hello
```

Ahora intente ejecutar el script con la opción `-c` o la opción `--` marcado

```
dart -c Test.dart
```

O,

```
dart --checked Test.dart
```

La máquina virtual Dart arrojará un error que indica que hay una falta de coincidencia de tipos.

```
Unhandled exception:
type 'String' is not a subtype of type 'int' of 'n' where
  String is from dart:core
  int is from dart:core
#0 main
(file:///C:/Users/Administrator/Desktop/test.dart:3:9)
#1 _startIsolate.<anonymous closure> (dart:isolate-
patch/isolate_patch.dart :261)
#2 _RawReceivePortImpl._handleMessage (dart:isolate-
patch/isolate_patch.dart:148)
```

Identificadores en Dart

Los identificadores son nombres dados a elementos en un programa como variables, funciones, etc. Las reglas para los identificadores son:

Los identificadores pueden incluir tanto caracteres como dígitos. Sin embargo, el identificador no puede comenzar con un dígito.

- Los identificadores no pueden incluir símbolos especiales, excepto el guión bajo (_) o un signo de dólar (\$).
- Los identificadores no pueden ser palabras clave.
- Deben ser únicos.
- Los identificadores distinguen entre mayúsculas y minúsculas.
- Los identificadores no pueden contener espacios.

Las siguientes tablas enumeran algunos ejemplos de identificadores válidos e inválidos:

Identificadores válidos	Identificadores inválidos
nombre de pila	Var
nombre de pila	nombre de pila
num1	nombre de pila
\$ resultado	1 número

Palabras clave en Dart

Las palabras clave tienen un significado especial en el contexto de un idioma. La siguiente tabla enumera algunas palabras clave en Dart.

abstract 1	continue	false	new	this
as 1	default	final	null	throw
assert	deferred 1	finally	operator 1	true
async 2	do	for	part 1	try
async* 2	dynamic 1	get 1	rethrow	typedef 1
await 2	else	if	return	var
break	enum	implements 1	set 1	void
case	export 1	import 1	static 1	while
catch	external 1	in	super	with
class	extends	is	switch	yield 2
const	factory 1	library 1	sync* 2	yield* 2

Espacios en blanco y saltos de línea

Dart ignora los espacios, las pestañas y las nuevas líneas que aparecen en los programas. Puede usar espacios, pestañas y líneas nuevas libremente en su programa y puede formatear e sangrar sus programas de una manera ordenada y coherente que hace que el código sea fácil de leer y comprender.

Dart es sensible a mayúsculas y minúsculas

Dart distingue entre mayúsculas y minúsculas. Esto significa que Dart diferencia entre mayúsculas y minúsculas.

Las declaraciones terminan con un punto y coma

Cada línea de instrucción se llama una declaración. Cada instrucción de dardo debe terminar con un punto y coma (;). Una sola línea puede contener múltiples declaraciones. Sin embargo, estas declaraciones deben estar separadas por un punto y coma.

Comentarios en Dart

Los comentarios son una forma de mejorar la legibilidad de un programa. Los comentarios se pueden usar para incluir información adicional sobre un programa como el autor del código, sugerencias sobre una función / construcción, etc. El compilador ignora los comentarios.

Dart admite los siguientes tipos de comentarios:

- **Comentarios de una sola línea (//)** : cualquier texto entre un "//" y el final de una línea se trata como un comentario
- **Comentarios de varias líneas (/ ** /)** : estos comentarios pueden abarcar varias líneas.

Ejemplo

```
// this is single line comment

/* This is a
   Multi-line comment
*/
```

Programación Orientada a Objetos en Dart

Dart es un lenguaje orientado a objetos. Object Orientation es un paradigma de desarrollo de software que sigue el modelado del mundo real. La orientación a objetos considera un programa como una colección de objetos que se comunican entre sí a través de un mecanismo llamado métodos.

- **Objeto** : un objeto es una representación en tiempo real de cualquier entidad. Según Grady Brooch, cada objeto debe tener tres características:
 - **Estado** : descrito por los atributos de un objeto.
 - **Comportamiento** : describe cómo actuará el objeto.

- **Identidad** : un valor único que distingue un objeto de un conjunto de objetos similares.
- **Clase** : una clase en términos de OOP es un plan para crear objetos. Una clase encapsula datos para el objeto.
- **Método** : los métodos facilitan la comunicación entre los objetos.

Ejemplo: dardo y orientación a objetos

```
class TestClass {  
    void disp() {  
        print("Hello World");  
    }  
}  
void main() {  
    TestClass c = new TestClass();  
    c.disp();  
}
```

El ejemplo anterior define una clase **TestClass**. La clase tiene un método **disp ()**. El método imprime la cadena "Hello World" en el terminal. La nueva palabra clave crea un objeto de la clase. El objeto invoca el método **disp ()**.

El código debe producir el siguiente **resultado** :

Hello World

Programación de Dart: tipos de datos

Una de las características más fundamentales de un lenguaje de programación es el conjunto de tipos de datos que admite. Estos son el tipo de valores que se pueden representar y manipular en un lenguaje de programación.

El lenguaje Dart admite los siguientes tipos:

- Números
- Instrumentos de cuerda
- Booleanos
- Liza
- Mapas

Números

Los números en Dart se usan para representar literales numéricos. El Number Dart viene en dos sabores:

- **Entero** : los valores enteros representan valores no fraccionarios, es decir, valores numéricos sin punto decimal. Por ejemplo, el valor "10" es un número entero. Los literales enteros se representan con la palabra clave **int**.

- **Double** - Dart también admite valores numéricos fraccionarios, es decir, valores con puntos decimales. El tipo de datos Doble en Dart representa un número de coma flotante de 64 bits (precisión doble). Por ejemplo, el valor "10.10". La palabra clave **double** se usa para representar literales de coma flotante.

Instrumentos de cuerda

Las cadenas representan una secuencia de caracteres. Por ejemplo, si tuviera que almacenar algunos datos como el nombre, la dirección, etc., se debe usar el tipo de datos de cadena. Una cadena Dart es una secuencia de unidades de código UTF-16. **Las runas** se usan para representar una secuencia de unidades de código UTF-32.

La palabra clave **String** se usa para representar literales de cadena. Los valores de cadena están incrustados en comillas simples o dobles.

Booleano

El tipo de datos booleanos representa valores booleanos verdadero y falso. Dart usa la palabra clave **bool** para representar un valor booleano.

Lista y mapa

La lista de tipos de datos y el mapa se utilizan para representar una colección de objetos. Una **lista** es un grupo ordenado de objetos. El tipo de datos Lista en Dart es sinónimo del concepto de matriz en otros lenguajes de programación. El tipo de datos **Map** representa un conjunto de valores como pares clave-valor. La biblioteca **dart: core** permite la creación y manipulación de estas colecciones a través de las clases predefinidas Lista y Mapa, respectivamente.

El tipo dinámico

Dart es un lenguaje opcionalmente escrito. Si el tipo de una variable no se especifica explícitamente, el tipo de la variable es **dinámico**. La palabra clave **dinámica** también se puede usar como una anotación de tipo explícitamente.

Programación de Dart: variables

Una variable es "un espacio con nombre en la memoria" que almacena valores. En otras palabras, actúa como contenedor de valores en un programa. Los nombres de las variables se denominan identificadores. Las siguientes son las reglas de nomenclatura para un identificador:

- Los identificadores no pueden ser palabras clave.
- Los identificadores pueden contener alfabetos y números.
- Los identificadores no pueden contener espacios y caracteres especiales, excepto el guión bajo (_) y el signo de dólar (\$).

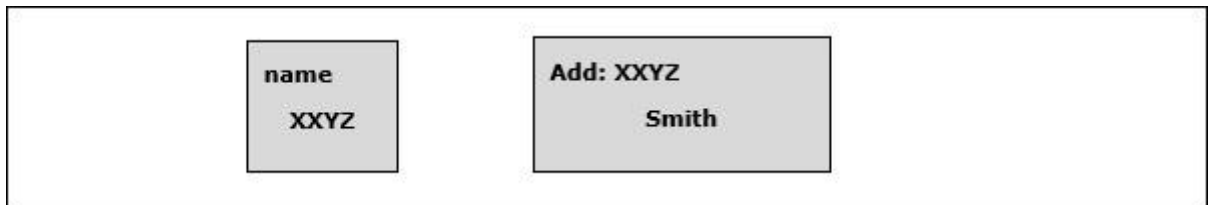
- Los nombres de las variables no pueden comenzar con un número.

Tipo de sintaxis

Se debe declarar una variable antes de usarla. Dart usa la palabra clave `var` para lograr lo mismo. La sintaxis para declarar una variable es la siguiente:

```
var name = 'Smith';
```

Todas las variables en dart almacenan una referencia al valor en lugar de contener el valor. La variable llamada `name` contiene una referencia a un objeto `String` con un valor de `"Smith"`.



Dart admite **la verificación de tipos** al prefijar el nombre de la variable con el tipo de datos. La verificación de tipo garantiza que una variable contenga solo datos específicos de un tipo de datos. La sintaxis para el mismo se da a continuación:

```
String name = 'Smith';  
int num = 10;
```

Considere el siguiente ejemplo:

```
void main() {  
    String name = 1;  
}
```

El fragmento anterior generará una advertencia ya que el valor asignado a la variable no coincide con el tipo de datos de la variable.

Salida

```
Warning: A value of type 'String' cannot be assigned to a  
variable of type 'int'
```

Todas las variables no inicializadas tienen un valor inicial de nulo. Esto se debe a que Dart considera todos los valores como objetos. El siguiente ejemplo ilustra lo mismo:

```
void main() {  
    int num;  
    print(num);  
}
```

Salida

```
Null
```

La palabra clave dinámica

Las variables declaradas sin un tipo estático se declaran implícitamente como dinámicas. Las variables también se pueden declarar utilizando la palabra clave dinámica en lugar de la palabra clave **var**.

El siguiente ejemplo ilustra lo mismo.

```
void main() {  
    dynamic x = "tom";  
    print(x);  
}
```

Salida

tom

Final y Const

Las palabras clave **final** y **const** se usan para declarar constantes. Dart evita modificar los valores de una variable declarada usando la palabra clave **final** o **const**. Estas palabras clave se pueden usar junto con el tipo de datos de la variable o en lugar de la palabra clave **var**.

La palabra clave **const** se usa para representar una constante de tiempo de compilación. Las variables declaradas con la palabra clave **const** son implícitamente finales.

Sintaxis: palabra clave final

```
final variable_name
```

O

```
final data_type variable_name
```

Sintaxis: const Palabra clave

```
const variable_name
```

O

```
const data_type variable_name
```

Ejemplo: palabra clave final

```
void main() {  
    final val1 = 12;  
    print(val1);  
}
```

Salida

12

Ejemplo: palabra clave const

```
void main() {  
    const pi = 3.14;  
    const area = pi*12*12;  
    print("The output is ${area}");  
}
```

El ejemplo anterior declara dos constantes, **pi** y **area** , usando la palabra clave **const** . El valor de la variable de **área** es una constante de tiempo de compilación.

Salida

The output is 452.15999999999997

Nota : Solo las variables **const** pueden usarse para calcular una constante de tiempo de compilación. Las constantes de tiempo de compilación son constantes cuyos valores se determinarán en tiempo de compilación

Ejemplo

Dart lanza una excepción si se intenta modificar las variables declaradas con la palabra clave **final** o **const**. El siguiente ejemplo ilustra lo mismo:

```
void main() {  
    final v1 = 12;  
    const v2 = 13;  
    v2 = 12;  
}
```

El código dado anteriormente arrojará el siguiente error como **salida** :

```
Unhandled exception:  
cannot assign to final variable 'v2='.  
NoSuchMethodError: cannot assign to final variable 'v2='  
#0  NoSuchMethodError._throwNew (dart:core-  
patch/errors_patch.dart:178)  
#1    main (file: Test.dart:5:3)  
#2    _startIsolate.<anonymous closure> (dart:isolate-  
patch/isolate_patch.dart:261)  
#3    _RawReceivePortImpl._handleMessage (dart:isolate-  
patch/isolate_patch.dart:148)
```

Programación de Dart: operadores

Una expresión es un tipo especial de declaración que se evalúa como un valor. Cada expresión se compone de -

- **Operandos** : representa los datos
- **Operador** : define cómo se procesarán los operandos para producir un valor.

Considere la siguiente expresión - "2 + 3". En esta expresión, 2 y 3 son **operandos** y el símbolo "+" (más) es el **operador** .

En este capítulo, discutiremos los operadores que están disponibles en Dart.

- Operadores aritméticos
- Igualdad y operadores relacionales
- Operadores de prueba de tipo
- Operadores bit a bit
- Operadores de Asignación
- Operadores logicos

Operadores aritméticos

La siguiente tabla muestra los operadores aritméticos admitidos por Dart.

[Mostrar ejemplos](#)

No Señor	Operadores y Significado
1	+ Añadir
2	- Sustraer
3	-expr Unario menos, también conocido como negación (invertir el signo de la expresión)
4 4	* * Multiplicar
5 5	// Dividir
6 6	~/ Divide, devolviendo un resultado entero
7 7	% Obtener el resto de una división entera (módulo)

8	++ Incremento
9 9	- Decremento

Igualdad y operadores relacionales

Operadores relacionales prueba o define el tipo de relación entre dos entidades. Los operadores relacionales devuelven un valor booleano, es decir, verdadero / falso.

Suponga que el valor de A es 10 y B es 20.

Mostrar ejemplos

Operador	Descripción	Ejemplo
>	Mas grande que	(A > B) es falso
<	Menor que	(A < B) es verdadero
> =	Mayor que o igual a	(A > = B) es falso
<=	Menor o igual que	(A <= B) es verdadero
==	Igualdad	(A == B) es verdadero
!=	No es igual	(A != B) es verdadero

Operadores de prueba de tipo

Estos operadores son útiles para verificar tipos en tiempo de ejecución.

Mostrar ejemplos

Operador	Sentido
es	Verdadero si el objeto tiene el tipo especificado

¡es!	Falso si el objeto tiene el tipo especificado
------	---

Operadores bit a bit

La siguiente tabla enumera los operadores bit a bit disponibles en Dart y su función:

[Mostrar ejemplos](#)

Operador	Descripción	Ejemplo
Bitwise Y	$a \& b$	Devuelve un uno en cada posición de bit para el cual los bits correspondientes de ambos operandos son unos.
Bitwise O	$a b$	Devuelve uno en cada posición de bit para el cual los bits correspondientes de uno o ambos operandos son unos.
Bitwise XOR	$a \wedge b$	Devuelve un uno en cada posición de bit para el cual los bits correspondientes de cualquiera de los operandos pero no de ambos son unos.
NO bit a bit	$\sim a$	Invierte los bits de su operando.
Shift izquierdo	$a \ll b$	Desplaza a en la representación binaria b (<32) bits hacia la izquierda, desplazando en ceros desde la derecha.
Desplazamiento a la derecha con signo	$a \gg b$	Desplaza a en la representación binaria b (<32) bits a la derecha, descartando los bits desplazados.

Operadores de Asignación

La siguiente tabla enumera los operadores de asignación disponibles en Dart.

[Mostrar ejemplos](#)

No Señor	Operador y Descripción
1	= (Asignación simple) Asigna valores del operando del lado derecho al operando del lado izquierdo Ej : $C = A + B$ asignará el valor de $A + B$ a C
2	?? = Asigne el valor solo si la variable es nula
3	+ = (Agregar y asignación) Agrega el operando derecho al operando izquierdo y asigna el resultado al operando izquierdo. Ej : $C += A$ es equivalente a $C = C + A$
4 4	- = (Restar y Asignación) Resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo. Ej : $C -= A$ es equivalente a $C = C - A$
5 5	* = (Multiplicar y Asignación) Multiplica el operando derecho con el operando izquierdo y asigna el resultado al operando izquierdo. Ej : $C *= A$ es equivalente a $C = C * A$
6 6	/ = (División y asignación) Divide el operando izquierdo con el operando derecho y asigna el resultado al operando izquierdo.

Nota : la misma lógica se aplica a los operadores Bitwise, por lo que se convertirán en $\ll =$, $\gg =$, $\gg =$, $\gg =$, $| =$ y $\wedge =$.

Operadores logicos

Los operadores lógicos se utilizan para combinar dos o más condiciones. Los operadores lógicos devuelven un valor booleano. Suponga que el valor de la variable A es 10 y B es 20.

[Mostrar ejemplos](#)

Operador	Descripción	Ejemplo

&&	Y - El operador devuelve verdadero solo si todas las expresiones especificadas devuelven verdadero	(A> 10 && B> 10) es falso.
	O - El operador devuelve verdadero si al menos una de las expresiones especificadas devuelve verdadero	(A> 10 B> 10) es verdadero.
!	NOT : el operador devuelve el inverso del resultado de la expresión. Por ejemplo:!(7> 5) devuelve falso	!(A> 10) es verdadero.

Expresiones condicionales

Dart tiene dos operadores que le permiten evaluar expresiones que de otro modo podrían requerir declaraciones ifelse:

condición? expr1: expr2

Si la condición es verdadera, la expresión evalúa **expr1** (y devuelve su valor); de lo contrario, evalúa y devuelve el valor de **expr2** .

expr1 ?? expr2

Si **expr1** no es nulo, devuelve su valor; de lo contrario, evalúa y devuelve el valor de **expr2**

Ejemplo

El siguiente ejemplo muestra cómo puede usar la expresión condicional en Dart:

```
void main() {
  var a = 10;
  var res = a > 12 ? "value greater than 10":"value lesser
than or equal to 10";
  print(res);
}
```

Producirá el siguiente resultado:

value lesser than or equal to 10

Ejemplo

Tomemos otro ejemplo:

```
void main() {
  var a = null;
  var b = 12;
```



```
var res = a ?? b;
print(res);
}
```

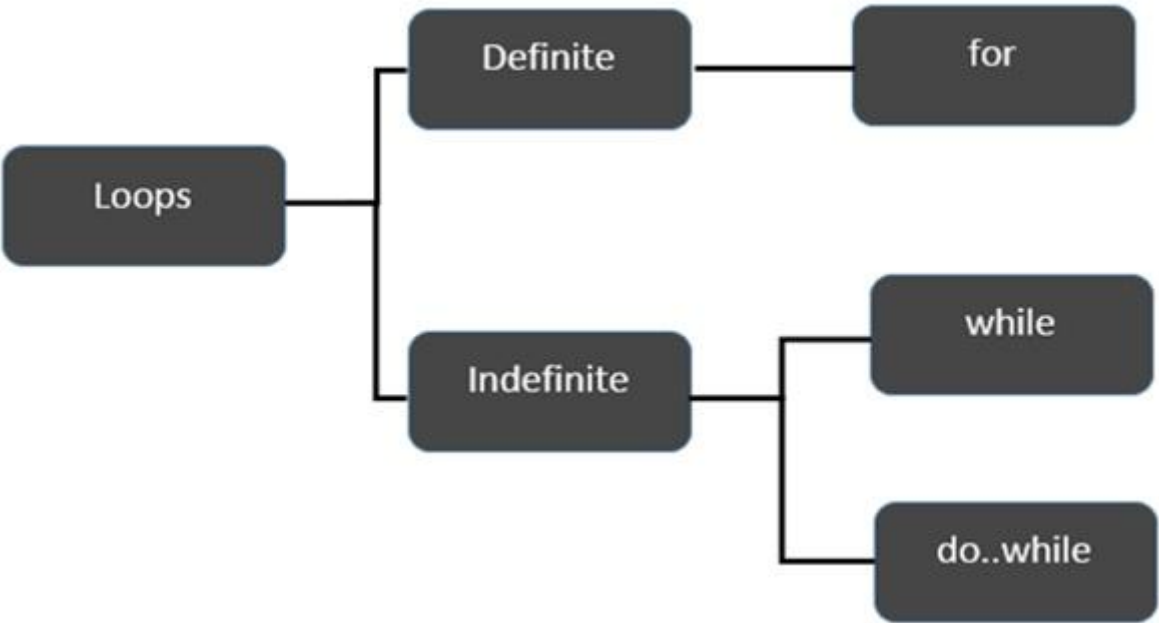
Producirá el siguiente resultado:

12

Programación de Dart: bucles

A veces, ciertas instrucciones requieren ejecución repetida. Los bucles son una forma ideal de hacer lo mismo. Un bucle representa un conjunto de instrucciones que deben repetirse. En el contexto de un bucle, una repetición se denomina **iteración** .

La siguiente figura ilustra la clasificación de los bucles:



Comencemos la discusión con Definite Loops. Un ciclo cuyo número de iteraciones es definitivo / fijo se denomina **ciclo definido** .

No Señor	Bucle y descripción
1	<u>en bucle</u> El bucle for es una implementación de un bucle definido. El bucle for ejecuta el bloque de código durante un número específico de veces. Se puede usar para iterar sobre un conjunto fijo de valores, como una matriz
2	<u>para ... en Loop</u> El bucle for ... in se usa para recorrer las propiedades de un objeto.

Continuando, ahora analicemos los bucles indefinidos. Un bucle indefinido se usa cuando el número de iteraciones en un bucle es indeterminado o desconocido. Los bucles indefinidos se pueden implementar usando:

No Señor	Bucle y descripción
1	<u>mientras Loop</u> El ciclo while ejecuta las instrucciones cada vez que la condición especificada se evalúa como verdadera. En otras palabras, el bucle evalúa la condición antes de que se ejecute el bloque de código.
2	<u>hacer ... mientras Loop</u> El ciclo do ... while es similar al ciclo while, excepto que el ciclo do ... while no evalúa la condición por primera vez.

Pasemos ahora y discutamos las **declaraciones de control de bucle** de Dart.

No Señor	Declaración de control y descripción
1	<u>Declaración de ruptura</u> La instrucción break se usa para quitar el control de una construcción. El uso de break in a loop hace que el programa salga del bucle. El siguiente es un ejemplo de la declaración de ruptura .
2	<u>continuar Declaración</u> La instrucción continue omite las declaraciones posteriores en la iteración actual y lleva el control al principio del ciclo.

Usar etiquetas para controlar el flujo

Una **etiqueta** es simplemente un identificador seguido de dos puntos (:) que se aplica a una declaración o un bloque de código. Se puede usar una etiqueta con **rotura** y **continuar** controlando el flujo con mayor precisión.

No se permiten saltos de línea entre la instrucción '**continuar**' o '**salto**' y su nombre de etiqueta. Además, no debe haber ninguna otra declaración entre un nombre de etiqueta y un bucle asociado.

Ejemplo: etiqueta con rotura

```
void main() {  
    outerloop: // This is the label name
```

```

for (var i = 0; i < 5; i++) {
    print("Innerloop: ${i}");
    innerloop:

    for (var j = 0; j < 5; j++) {
        if (j > 3 ) break ;

        // Quit the innermost loop
        if (i == 2) break innerloop;

        // Do the same thing
        if (i == 4) break outerloop;

        // Quit the outer loop
        print("Innerloop: ${j}");
    }
}

```

La siguiente **salida** se muestra en la ejecución exitosa del código anterior.

```

Innerloop: 0
Innerloop: 0
Innerloop: 1
Innerloop: 2
Innerloop: 3
Innerloop: 1
Innerloop: 0
Innerloop: 1
Innerloop: 2
Innerloop: 3
Innerloop: 2
Innerloop: 3
Innerloop: 0
Innerloop: 1
Innerloop: 2
Innerloop: 3
Innerloop: 4

```

Ejemplo: etiqueta con continuar

```

void main() {
    outerloop: // This is the label name

    for (var i = 0; i < 3; i++) {
        print("Outerloop:${i}");

        for (var j = 0; j < 5; j++) {
            if (j == 3){
                continue outerloop;
            }
        }
    }
}

```

```
        print("Innerloop:${j}");  
    }  
}  
}
```

La siguiente salida se muestra en la ejecución exitosa del código anterior.

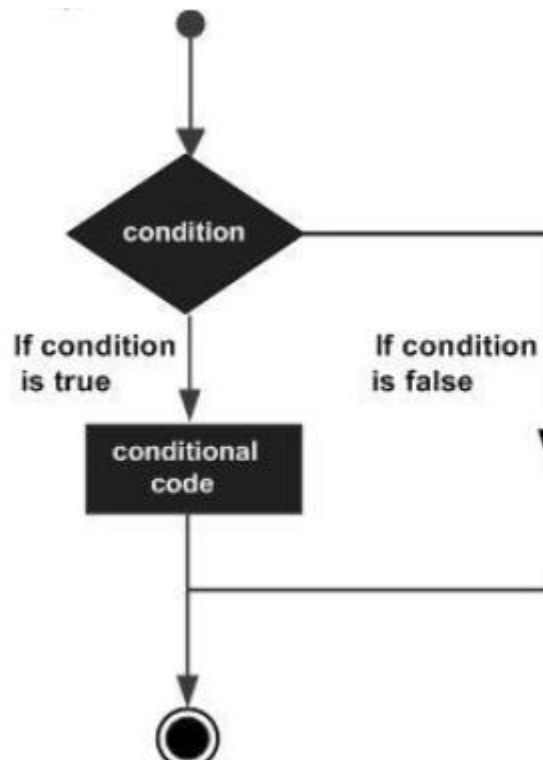
```
Outerloop: 0  
Innerloop: 0  
Innerloop: 1  
Innerloop: 2
```

```
Outerloop: 1  
Innerloop: 0  
Innerloop: 1  
Innerloop: 2
```

```
Outerloop: 2  
Innerloop: 0  
Innerloop: 1  
Innerloop: 2
```

Programación de Dart: toma de decisiones

Una construcción condicional / de toma de decisiones evalúa una condición antes de que se ejecuten las instrucciones.



Las construcciones condicionales en Dart se clasifican en la siguiente tabla.

No Señor	Declaración y descripción
1	<u>si la declaración</u> Una declaración if consiste en una expresión booleana seguida de una o más declaraciones.
2	<u>Si ... otra declaración</u> Un if puede ser seguido por un bloque else opcional . El bloque else se ejecutará si la expresión booleana probada por el bloque if se evalúa como falsa.
3	<u>si no ... si Ladder</u> El otro ... si la escalera es útil para probar múltiples condiciones. La siguiente es la sintaxis de la misma.
4 4	<u>cambiar ... Declaración de caso</u> La instrucción switch evalúa una expresión, hace coincidir el valor de la expresión con una cláusula case y ejecuta las declaraciones asociadas con ese caso.

Programación de Dart - Números

Los números de Dart se pueden clasificar como:

- **int** - Entero de tamaño arbitrario. El tipo de datos **int** se usa para representar números enteros.
- **double** : números de coma flotante de 64 bits (precisión doble), según lo especificado por el estándar IEEE 754. El tipo de datos **double** se utiliza para representar números fraccionarios.

El tipo **num** es heredado por los tipos **int** y **double** . La **biblioteca central de dart** permite numerosas operaciones en valores numéricos.

La sintaxis para declarar un número es la siguiente:

```
int var_name;      // declares an integer variable
double var_name;   // declares a double variable
```

Ejemplo

```
void main() {
    // declare an integer
    int num1 = 10;

    // declare a double value
    double num2 = 10.50;

    // print the values
    print(num1);
    print(num2);
}
```

```
}
```

Producirá el siguiente resultado:

```
10  
10.5
```

Nota : Dart VM generará una excepción si se asignan valores fraccionarios a las variables enteras.

Analizando

La función estática **parse ()** permite analizar una cadena que contiene literal numérico en un número. La siguiente ilustración demuestra lo mismo:

```
void main() {  
    print(num.parse('12'));  
    print(num.parse('10.91'));  
}
```

El código anterior dará como resultado el siguiente resultado:

```
12  
10.91
```

La función de análisis arroja una **Excepción de formato** si se pasa cualquier valor que no sean números. El siguiente código muestra cómo pasar un valor alfanumérico a la función **parse ()**.

Ejemplo

```
void main() {  
    print(num.parse('12A'));  
    print(num.parse('AAAA'));  
}
```

El código anterior dará como resultado el siguiente resultado:

```
Unhandled exception:  
FormatException: 12A  
#0 num.parse (dart:core/num.dart:446)  
#1 main (file:///D:/Demos/numbers.dart:4:13)  
#2 _startIsolate.<anonymous closure>  
    (dart:isolatepatch/isolate_patch.dart:261)  
#3 _RawReceivePortImpl._handleMessage  
    (dart:isolatepatch/isolate_patch.dart:148)
```

Propiedades numéricas

La siguiente tabla enumera las propiedades compatibles con los números Dart.

No Señor	Descripción de propiedad
1	<u>código hash</u> Devuelve un código hash para un valor numérico.
2	<u>isFinite</u> Verdadero si el número es finito; de lo contrario, falso.
3	<u>es infinito</u> Verdadero si el número es infinito positivo o infinito negativo; de lo contrario, falso.
4 4	isNan Verdadero si el número es el valor doble No es un número; de lo contrario, falso.
5 5	<u>isNegative</u> Verdadero si el número es negativo; de lo contrario, falso.
6 6	<u>firmar</u> Devuelve menos uno, cero o más uno dependiendo del signo y el valor numérico del número.
7 7	<u>incluso</u> Devuelve verdadero si el número es un número par.
8	<u>es impar</u> Devuelve verdadero si el número es un número impar.

Métodos numéricos

A continuación se incluye una lista de los métodos de uso común compatibles con números:

No Señor	Método y descripción
1	<u>abdominales</u> Devuelve el valor absoluto del número.
2	<u>fortificar techo</u>

	Devuelve el menor entero no menor que el número.
3	<u>comparar con</u> Compara esto con otro número.
4 4	<u>Piso</u> Devuelve el mayor entero no mayor que el número actual.
5 5	<u>recordatorio</u> Devuelve el resto truncado después de dividir los dos números.
6 6	<u>Redondo</u> Devuelve el entero más cercano a los números actuales.
7 7	<u>para duplicar</u> Devuelve el doble equivalente del número.
8	<u>toInt</u> Devuelve el equivalente entero del número.
9 9	Devuelve la representación de cadena equivalente del número.
10	<u>truncar</u> Devuelve un entero después de descartar cualquier dígito fraccionario.

Programación de Dart: cadena

El tipo de datos de cadena representa una secuencia de caracteres. Una cadena Dart es una secuencia de unidades de código UTF 16.

Los valores de cadena en Dart se pueden representar mediante comillas simples, dobles o triples. Las cadenas de línea simple se representan mediante comillas simples o dobles. Las comillas triples se utilizan para representar cadenas de varias líneas.

La sintaxis de la representación de valores de cadena en Dart es la siguiente:

Sintaxis

```
String variable_name = 'value'
```

OR

```
String variable_name = '''value'''
```


OR

```
String variable_name = '''line1  
line2'''
```

OR

```
String variable_name= '''line1  
line2''''
```

El siguiente ejemplo ilustra el uso del tipo de datos String en Dart.

```
void main() {  
    String str1 = 'this is a single line string';  
    String str2 = "this is a single line string";  
    String str3 = '''this is a multiline line string''';  
    String str4 = """this is a multiline line string""";  
  
    print(str1);  
    print(str2);  
    print(str3);  
    print(str4);  
}
```

Producirá la siguiente **salida** :

```
this is a single line string  
this is a single line string  
this is a multiline line string  
this is a multiline line string
```

Las cuerdas son inmutables. Sin embargo, las cadenas se pueden someter a varias operaciones y la cadena resultante se puede almacenar como un nuevo valor.

Interpolación de cuerdas

El proceso de crear una nueva cadena agregando un valor a una cadena estática se denomina **concatenación** o **interpolación** . En otras palabras, es el proceso de agregar una cadena a otra cadena.

El operador plus (+) es un mecanismo de uso común para concatenar / interpolar cadenas.

Ejemplo 1

```
void main() {  
    String str1 = "hello";  
    String str2 = "world";  
    String res = str1+str2;  
  
    print("The concatenated string : ${res}");  
}
```

```
}
```

Producirá el siguiente **resultado** :

The concatenated string : Helloworld

Ejemplo 2

Puede usar "\$ {}" para interpolar el valor de una expresión Dart dentro de cadenas. El siguiente ejemplo ilustra lo mismo.

```
void main() {  
    int n=1+1;  
  
    String str1 = "The sum of 1 and 1 is ${n}";  
    print(str1);  
  
    String str2 = "The sum of 2 and 2 is ${2+2}";  
    print(str2);  
}
```

Producirá el siguiente **resultado** :

The sum of 1 and 1 is 2
The sum of 2 and 2 is 4

Propiedades de cadena

Las propiedades enumeradas en la siguiente tabla son de solo lectura.

No Señor	Descripción de propiedad
1	<u>codeUnits</u> Devuelve una lista no modificable de las unidades de código UTF-16 de esta cadena.
2	<u>esta vacío</u> Devuelve verdadero si esta cadena está vacía.
3	<u>Longitud</u> Devuelve la longitud de la cadena, incluidos el espacio, la tabulación y los caracteres de nueva línea.

Métodos para manipular cadenas

La clase String en la **biblioteca dart: core** también proporciona métodos para manipular cadenas. Algunos de estos métodos se dan a continuación:

No Señor	Métodos y descripción
1	<u>toLowerCase ()</u> Convierte todos los caracteres de esta cadena a minúsculas.
2	<u>toUpperCase ()</u> Convierte todos los caracteres de esta cadena a mayúsculas.
3	<u>podar()</u> Devuelve la cadena sin espacios en blanco iniciales y finales.
4 4	<u>comparar con()</u> Compara este objeto con otro.
5 5	<u>reemplaza todo()</u> Reemplaza todas las subcadenas que coinciden con el patrón especificado con un valor dado.
6 6	<u>división()</u> Divide la cadena en coincidencias del delimitador especificado y devuelve una lista de subcadenas.
7 7	<u>subcadena ()</u> Devuelve la subcadena de esta cadena que se extiende desde startIndex, inclusive, hasta endIndex, exclusive.
8	<u>Encadenar()</u> Devuelve una representación de cadena de este objeto.
9 9	<u>codeUnitAt ()</u> Devuelve la unidad de código UTF-16 de 16 bits en el índice dado.

Programación de Dart - booleano

Dart proporciona un soporte incorporado para el tipo de datos booleanos. El tipo de datos booleanos en DART solo admite dos valores: verdadero y falso. La palabra clave bool se usa para representar un literal booleano en DART.

La sintaxis para declarar una variable booleana en DART es la siguiente:

```
bool var_name = true;  
OR  
bool var_name = false
```

Ejemplo

```
void main() {  
    bool test;  
    test = 12 > 5;  
    print(test);  
}
```

Producirá el siguiente **resultado** :

true

Ejemplo

A diferencia de JavaScript, el tipo de datos booleanos reconoce solo el verdadero literal como verdadero. Cualquier otro valor se considera falso. Considere el siguiente ejemplo:

```
var str = 'abc';  
if(str) {  
    print('String is not empty');  
} else {  
    print('Empty String');  
}
```

El fragmento anterior, si se ejecuta en JavaScript, imprimirá el mensaje 'La cadena no está vacía' ya que la construcción if devolverá verdadero si la cadena no está vacía.

Sin embargo, en Dart, **str** se convierte en *falso como str! = True* . Por lo tanto, el fragmento imprimirá el mensaje '*Cadena vacía*' (cuando se ejecuta en modo sin marcar).

Ejemplo

El fragmento anterior si se ejecuta en modo **marcado** arrojará una excepción. Lo mismo se ilustra a continuación:

```
void main() {  
    var str = 'abc';  
    if(str) {  
        print('String is not empty');  
    } else {  
        print('Empty String');  
    }  
}
```

Producirá la siguiente **salida** , en **modo marcado** :

```
Unhandled exception:
type 'String' is not a subtype of type 'bool' of 'boolean
expression' where
  String is from dart:core
  bool is from dart:core
#0 main (file:///D:/Demos/Boolean.dart:5:6)
#1 _startIsolate.<anonymous closure> (dart:isolate-
patch/isolate_patch.dart:261)
#2 _RawReceivePortImpl._handleMessage (dart:isolate-
patch/isolate_patch.dart:148)
```

Producirá la siguiente **salida** , en **modo sin marcar** :

```
Empty String
```

Nota: El IDE de **WebStorm** se ejecuta en modo marcado, de forma predeterminada.

Programación de Dart - Listas

Una colección muy utilizada en programación es una **matriz** . Dart representa matrices en forma de objetos **List** . Una **lista** es simplemente un grupo ordenado de objetos. La biblioteca **dart: core** proporciona la clase Lista que permite la creación y manipulación de listas.

La representación lógica de una lista en Dart se da a continuación:

test_list	0	1	2
	12	13	14

- **test_list** : es el identificador que hace referencia a la colección.
- La lista contiene los valores 12, 13 y 14. Los bloques de memoria que contienen estos valores se conocen como **elementos** .
- Cada elemento de la Lista se identifica por un número único llamado **índice** . El índice comienza desde **cero** y se extiende hasta **n-1**, donde **n** es el número total de elementos en la Lista. El índice también se conoce como **subíndice** .

Las listas se pueden clasificar como:

- Lista de longitud fija
- Lista cultivable

Discutamos ahora estos dos tipos de **listas** en detalle.

Lista de longitud fija

La longitud de una lista de longitud fija no puede cambiar en tiempo de ejecución. La sintaxis para crear una lista de longitud fija es la siguiente:

Paso 1: declarar una lista

La sintaxis para declarar una lista de longitud fija se proporciona a continuación:

```
var list_name = new List(initial_size)
```

La sintaxis anterior crea una lista del tamaño especificado. La lista no puede crecer ni reducirse en tiempo de ejecución. Cualquier intento de cambiar el tamaño de la lista dará como resultado una excepción.

Paso 2 - Inicializando una lista

La sintaxis para inicializar una lista es la siguiente:

```
lst_name[index] = value;
```

Ejemplo

```
void main() {  
    var lst = new List(3);  
    lst[0] = 12;  
    lst[1] = 13;  
    lst[2] = 11;  
    print(lst);  
}
```

Producirá el siguiente **resultado** :

```
[12, 13, 11]
```

Lista cultivable

La longitud de una lista ampliable puede cambiar en tiempo de ejecución. La sintaxis para declarar e inicializar una lista ampliable es la siguiente:

Paso 1: declarar una lista

```
var list_name = [val1,val2,val3]  
--- creates a list containing the specified values  
OR  
var list_name = new List()  
--- creates a list of size zero
```

Paso 2 - Inicializando una Lista

El índice / subíndice se usa para hacer referencia al elemento que debe rellenarse con un valor. La sintaxis para inicializar una lista es la siguiente:

```
list_name[index] = value;
```

Ejemplo

El siguiente ejemplo muestra cómo crear una lista de 3 elementos.

```
void main() {
```

```
var num_list = [1,2,3];
print(num_list);
}
```

Producirá el siguiente **resultado** :

[1, 2, 3]

Ejemplo

El siguiente ejemplo crea una lista de longitud cero usando el **constructor List () vacío** . La función **add ()** en la clase **List** se usa para agregar elementos dinámicamente a la lista.

```
void main() {
    var lst = new List();
    lst.add(12);
    lst.add(13);
    print(lst);
}
```

Producirá el siguiente **resultado** :

[12, 13]

Propiedades de la lista

La siguiente tabla enumera algunas propiedades de uso común de la clase **Lista** en la **biblioteca dart: core** .

No Señor	Métodos y descripción
1	<u>primero</u> Devuelve el primer caso del elemento.
2	<u>esta vacío</u> Devuelve verdadero si la colección no tiene elementos.
3	<u>no está vacío</u> Devuelve verdadero si la colección tiene al menos un elemento.
4 4	<u>longitud</u> Devuelve el tamaño de la lista.
5 5	<u>último</u>

	Devuelve el último elemento de la lista.
6 6	<u>invertido</u> Devuelve un objeto iterable que contiene los valores de las listas en el orden inverso.
7 7	<u>Soltero</u> Comprueba si la lista tiene solo un elemento y lo devuelve.

Programación de Dart - Listas (operaciones básicas)

En este capítulo, discutiremos cómo llevar a cabo algunas operaciones básicas en las listas, tales como:

No Señor	Operación básica y descripción
1	<u>Insertar elementos en una lista</u> Las listas mutables pueden crecer dinámicamente en tiempo de ejecución. La función List.add () agrega el valor especificado al final de la Lista y devuelve un objeto Lista modificado.
2	<u>Actualizando una lista</u> Las listas en Dart se pueden actualizar mediante: <ul style="list-style-type: none"> • <u>Actualización del índice</u> • <u>Usando la función List.replaceRange ()</u>
3	<u>Eliminar elementos de la lista</u> Las siguientes funciones compatibles con la clase Lista en el dardo: la biblioteca principal se puede usar para eliminar los elementos de una Lista.

Programación de Dart - Mapa

El objeto Map es un simple par clave / valor. Las claves y los valores en un mapa pueden ser de cualquier tipo. Un mapa es una colección dinámica. En otras palabras, los mapas pueden crecer y reducirse en tiempo de ejecución.

Los mapas se pueden declarar de dos maneras:

- Usar mapas literales
- Usar un constructor de mapas

Declarando un Mapa usando Literales de Mapa

Para declarar un mapa usando literales de mapa, debe encerrar los pares clave-valor dentro de un par de llaves "{}" .

Aquí está su **sintaxis** :

```
var identifier = { key1:value1, key2:value2  
[,...,key_n:value_n] }
```

Declarar un mapa usando un constructor de mapas

Para declarar un mapa usando un constructor de mapas, tenemos dos pasos. Primero, declare el mapa y segundo, inicialice el mapa.

La **sintaxis** para **declarar un mapa** es la siguiente:

```
var identifier = new Map()
```

Ahora, use la siguiente sintaxis para **inicializar el mapa** :

```
map_name[key] = value
```

Ejemplo: Mapa Literal

```
void main() {  
    var details = {'Username':'tom','Password':'pass@123'};  
    print(details);  
}
```

Producirá el siguiente **resultado** :

```
{Username: tom, Password: pass@123}
```

Ejemplo: Agregar valores a literales de mapas en tiempo de ejecución

```
void main() {  
    var details = {'Username':'tom','Password':'pass@123'};  
    details['Uid'] = 'U1001';  
    print(details);  
}
```

Producirá el siguiente **resultado** :

```
{Username: tom, Password: pass@123, Uid: U1001}
```

Ejemplo: Constructor de mapas

```
void main() {  
    var details = new Map();  
    details['Username'] = 'admin';  
    details['Password'] = 'admin@123';  
    print(details);  
}
```

```
}
```

Producirá el siguiente **resultado** :

```
{Username: admin, Password: admin@123}
```

Nota : un valor de mapa puede ser cualquier objeto, incluido NULL.

Mapa - Propiedades

La clase **Map** en el paquete `dart: core` define las siguientes propiedades:

No Señor	Descripción de propiedad
1	<u>Llaves</u> Devuelve un objeto iterable que representa claves
2	<u>Valores</u> Devuelve un objeto iterable que representa valores
3	<u>Longitud</u> Devuelve el tamaño del mapa
4 4	<u>esta vacío</u> Devuelve verdadero si el mapa es un mapa vacío
5 5	<u>no está vacío</u> Devuelve verdadero si el mapa es un mapa vacío

Mapa - Funciones

Las siguientes son las funciones comúnmente utilizadas para manipular Maps en Dart.

No Señor	Nombre y descripción de la función
1	<u>añadir todo()</u> Agrega todos los pares clave-valor de otro a este mapa.
2	<u>claro()</u> Elimina todos los pares del mapa.

3	<u>eliminar()</u> Elimina la clave y su valor asociado, si está presente, del mapa.
4 4	<u>para cada()</u> Aplica f a cada par clave-valor del mapa.

Programación de Dart - Símbolo

Los símbolos en Dart son un nombre de cadena dinámico y opaco que se utiliza para reflejar los metadatos de una biblioteca. En pocas palabras, los símbolos son una forma de almacenar la relación entre una cadena legible por humanos y una cadena que está optimizada para ser utilizada por las computadoras.

La reflexión es un mecanismo para obtener metadatos de un tipo en tiempo de ejecución como la cantidad de métodos en una clase, la cantidad de constructores que tiene o la cantidad de parámetros en una función. Incluso puede invocar un método del tipo que se carga en tiempo de ejecución.

En Dart, las clases específicas de reflexión están disponibles en el paquete **dart: mirrors**. Esta biblioteca funciona tanto en aplicaciones web como en aplicaciones de línea de comandos.

Sintaxis

```
Symbol obj = new Symbol('name');
// expects a name of class or function or library to reflect
```

El **nombre** debe ser un nombre de miembro de Dart público válido, un nombre de constructor público o un nombre de biblioteca.

Ejemplo

Considere el siguiente ejemplo. El código declara una clase **Foo** en una biblioteca **foo_lib**. La clase define los métodos **m1**, **m2** y **m3**.

Foo.dart

```
library foo_lib;
// library name can be a symbol

class Foo {
  // class name can be a symbol
  m1() {
    // method name can be a symbol
    print("Inside m1");
  }
  m2() {
    print("Inside m2");
  }
}
```

```

    }
    m3() {
        print("Inside m3");
    }
}

```

El siguiente código carga la biblioteca **Foo.dart** y busca la clase Foo, con ayuda del tipo Símbolo. Como estamos reflejando los metadatos de la biblioteca anterior, el código importa la biblioteca **dart: mirrors** .

FooSymbol.dart

```

import 'dart:core';
import 'dart:mirrors';
import 'Foo.dart';

main() {
    Symbol lib = new Symbol("foo_lib");
    //library name stored as Symbol

    Symbol clsToSearch = new Symbol("Foo");
    // class name stored as Symbol

    if(checkIf_classAvailableInlibrary(lib, clsToSearch))
        // searches Foo class in foo_lib library
        print("class found..");
}

bool checkIf_classAvailableInlibrary(Symbol libraryName,
Symbol className) {
    MirrorSystem mirrorSystem = currentMirrorSystem();
    LibraryMirror libMirror =
mirrorSystem.findLibrary(libraryName);

    if (libMirror != null) {
        print("Found Library");
        print("checkng...class details..");
        print("No of classes found is :
${libMirror.declarations.length}");
        libMirror.declarations.forEach((s, d) => print(s));

        if (libMirror.declarations.containsKey(className))
return true;
        return false;
    }
}

```

Tenga en cuenta que la línea `libMirror.declarations.forEach ((s, d) => print (s));` iterará en cada declaración de la biblioteca en tiempo de ejecución e imprimirá las declaraciones como tipo de **símbolo** .

Este código debería producir el siguiente **resultado** :

```
Found Library
checkng...class details..
No of classes found is : 1
Symbol("Foo") // class name displayed as symbol
class found.
```

Ejemplo: mostrar el número de métodos de instancia de una clase

Consideremos ahora mostrar el número de métodos de instancia en una clase. La clase predefinida **ClassMirror** nos ayuda a lograr lo mismo.

```
import 'dart:core';
import 'dart:mirrors';
import 'Foo.dart';

main() {
  Symbol lib = new Symbol("foo_lib");
  Symbol clsToSearch = new Symbol("Foo");
  reflect_InstanceMethods(lib, clsToSearch);
}

void reflect_InstanceMethods(Symbol libraryName, Symbol
className) {
  MirrorSystem mirrorSystem = currentMirrorSystem();
  LibraryMirror libMirror =
mirrorSystem.findLibrary(libraryName);

  if (libMirror != null) {
    print("Found Library");
    print("checkng...class details..");
    print("No of classes found is :
${libMirror.declarations.length}");
    libMirror.declarations.forEach((s, d) => print(s));

    if (libMirror.declarations.containsKey(className))
print("found class");
    ClassMirror classMirror =
libMirror.declarations[className];

    print("No of instance methods found is
${classMirror.instanceMembers.length}");
    classMirror.instanceMembers.forEach((s, v) =>
print(s));
  }
}
```

Este código debería producir el siguiente **resultado** :

```
Found Library
checkng...class details..
No of classes found is : 1
Symbol("Foo")
found class
No of instance methods found is 8
Symbol("==")
```

```
Symbol("hashCode")
Symbol("toString")
Symbol("noSuchMethod")
Symbol("runtimeType")
Symbol("m1")
Symbol("m2")
Symbol("m3")
```

Convertir símbolo en cadena

Puede convertir el nombre de un tipo como clase o biblioteca almacenada en un símbolo de nuevo a cadena usando la clase **MirrorSystem** . El siguiente código muestra cómo puede convertir un símbolo en una cadena.

```
import 'dart:mirrors';
void main() {
  Symbol lib = new Symbol("foo_lib");
  String name_of_lib = MirrorSystem.getName(lib);

  print(lib);
  print(name_of_lib);
}
```

Debería producir el siguiente **resultado** :

```
Symbol("foo_lib")
```

```
foo_lib
```

Programación de Dart - Runas

Las cadenas son una secuencia de caracteres. Dart representa cadenas como una secuencia de unidades de código Unicode UTF-16. Unicode es un formato que define un valor numérico único para cada letra, dígito y símbolo.

Como una cadena Dart es una secuencia de unidades de código UTF-16, los valores Unicode de 32 bits dentro de una cadena se representan mediante una sintaxis especial. Una **runa** es un número entero que representa un punto de código Unicode.

La clase **String** en la biblioteca **dart: core** proporciona mecanismos para acceder a las **runas** . Se puede acceder a las unidades de código de cadena / runas de tres maneras:

- Usando la función `String.codeUnitAt ()`
- Uso de la propiedad `String.codeUnits`
- Usando la propiedad `String.runes`

Función `String.codeUnitAt ()`

Se puede acceder a las unidades de código en una cadena a través de sus índices. Devuelve la unidad de código UTF-16 de 16 bits en el índice dado.

Sintaxis

```
String.codeUnitAt(int index);
```

Ejemplo

```
import 'dart:core';
void main() {
  f1();
}
f1() {
  String x = 'Runes';
  print(x.codeUnitAt(0));
}
```

Producirá el siguiente **resultado** :

82

Propiedad String.codeUnits

Esta propiedad devuelve una lista no modificable de las unidades de código UTF-16 de la cadena especificada.

Sintaxis

```
String.codeUnits;
```

Ejemplo

```
import 'dart:core';
void main() {
  f1();
}
f1() {
  String x = 'Runes';
  print(x.codeUnits);
}
```

Producirá el siguiente **resultado** :

[82, 117, 110, 101, 115]

Propiedad String.runes

Esta propiedad devuelve un iterable de puntos de código Unicode de esta **cadena** . Las **runas** se extienden iterable.

Sintaxis

String.runes

Ejemplo

```
void main(){
  "A string".runes.forEach((int rune) {
    var character=new String.fromCharCode(rune);
    print(character);
  });
}
```

Producirá el siguiente **resultado** :

A
s
t
r
i
n
g

Los puntos de código Unicode generalmente se expresan como `\ uXXXX` , donde XXXX es un valor hexadecimal de 4 dígitos. Para especificar más o menos de 4 dígitos hexadecimales, coloque el valor entre llaves. Se puede usar el constructor de la clase Runes en la biblioteca dart: core para la misma.

Ejemplo

```
main() {
  Runes input = new Runes(' \u{1f605} ');
  print(new String.fromCharCode(input));
}
```

Producirá el siguiente **resultado** :



Programación de Dart - Enumeración

Se utiliza una enumeración para definir valores constantes con nombre. Un tipo enumerado se declara utilizando la palabra clave **enum** .

Sintaxis

```
enum enum_name {
  enumeration list
}
```

Dónde,

- El *enum_name* especifica el nombre del tipo de enumeración

- La *lista de enumeración* es una lista de identificadores separados por comas.

Cada uno de los símbolos en la lista de enumeración representa un valor entero, uno mayor que el símbolo que lo precede. Por defecto, el valor del primer símbolo de enumeración es 0.

Por ejemplo

```
enum Status {  
  none,  
  running,  
  stopped,  
  paused  
}
```

Ejemplo

```
enum Status {  
  none,  
  running,  
  stopped,  
  paused  
}  
  
void main() {  
  print(Status.values);  
  Status.values.forEach((v) => print('value: $v, index:  
  ${v.index}'));  
  print('running: ${Status.running},  
  ${Status.running.index}');  
  print('running index: ${Status.values[1]}');  
}
```

Producirá el siguiente **resultado** :

```
[Status.none, Status.running, Status.stopped, Status.paused]  
value: Status.none, index: 0  
value: Status.running, index: 1  
value: Status.stopped, index: 2  
value: Status.paused, index: 3  
running: Status.running, 1  
running index: Status.running
```

Programación de Dart - Funciones

Las funciones son los componentes básicos de un código legible, mantenible y reutilizable. Una función es un conjunto de declaraciones para realizar una tarea específica. Las funciones organizan el programa en bloques lógicos de código. Una vez definidas, las funciones pueden ser llamadas para acceder al código. Esto hace que el código sea reutilizable. Además, las funciones facilitan la lectura y el mantenimiento del código del programa.

Una declaración de función le dice al compilador sobre el nombre de una función, el tipo de retorno y los parámetros. Una definición de función proporciona el cuerpo real de la función.

No Señor	Funciones y descripción
1	<u>Definiendo una función</u> Una definición de función especifica qué y cómo se realizaría una tarea específica.
2	<u>Llamar a una función</u> Se debe llamar a una función para ejecutarla.
3	<u>Funciones de retorno</u> Las funciones también pueden devolver el valor junto con el control, de vuelta a la persona que llama.
4 4	<u>Función parametrizada</u> Los parámetros son un mecanismo para pasar valores a funciones.

Parámetros opcionales

Los parámetros opcionales se pueden usar cuando no es necesario pasar obligatoriamente los argumentos para la ejecución de una función. Un parámetro se puede marcar como opcional agregando un signo de interrogación a su nombre. El parámetro opcional debe establecerse como el último argumento en una función.

Tenemos tres tipos de parámetros opcionales en Dart:

No Señor	Descripción de parámetros
1	<u>Parámetro Posicional Opcional</u> Para especificar parámetros posicionales opcionales, use corchetes [].
2	<u>Parámetro con nombre opcional</u> A diferencia de los parámetros posicionales, el nombre del parámetro debe especificarse mientras se pasa el valor. La llave {} se puede usar para especificar parámetros con nombre opcionales.
3	<u>Parámetros opcionales con valores predeterminados</u>

Los parámetros de función también se pueden asignar valores por defecto. Sin embargo, dichos parámetros también pueden ser valores explícitamente pasados.
--

Funciones de dardo recursivo

La recursión es una técnica para iterar sobre una operación al hacer que una función se llame a sí misma repetidamente hasta que llegue a un resultado. La recursión se aplica mejor cuando necesita llamar a la misma función repetidamente con diferentes parámetros desde un bucle.

Ejemplo

```
void main() {
    print(factorial(6));
}
factorial(number) {
    if (number <= 0) {
        // termination case
        return 1;
    } else {
        return (number * factorial(number - 1));
        // function invokes itself
    }
}
```

Debería producir el siguiente **resultado** :

720

Funciones Lambda

Las funciones lambda son un mecanismo conciso para representar funciones. Estas funciones también se denominan funciones de flecha.

Sintaxis

```
[return_type] function_name(parameters) => expression;
```

Ejemplo

```
void main() {
    printMsg();
    print(test());
}
printMsg()=>
print("hello");
```

```
int test()=>123;  
// returning function
```

Debería producir el siguiente **resultado** :

hello 123

Programación de Dart: interfaces

Una **interfaz** define la sintaxis a la que debe adherirse cualquier entidad. Las interfaces definen un conjunto de métodos disponibles en un objeto. Dart no tiene una sintaxis para declarar interfaces. Las declaraciones de clase son en sí mismas interfaces en Dart.

Las clases deben usar la palabra clave `implements` para poder usar una interfaz. Es obligatorio que la clase implementadora proporcione una implementación concreta de todas las funciones de la interfaz implementada. En otras palabras, una clase debe redefinir cada función en la interfaz que desea implementar.

Sintaxis: Implementación de una interfaz

```
class identifier implements interface_name
```

Ejemplo

En el siguiente programa, estamos declarando una clase **Impresora** . La clase **ConsolePrinter** implementa la declaración de interfaz implícita para la clase **Impresora** . La función **principal** crea un objeto de la clase **ConsolePrinter** usando la **nueva** palabra clave. Este objeto se utiliza para invocar la función **print_data** definida en la clase **ConsolePrinter** .

```
void main() {  
    ConsolePrinter cp= new ConsolePrinter();  
    cp.print_data();  
}  
class Printer {  
    void print_data() {  
        print("_____Printing Data_____");  
    }  
}  
class ConsolePrinter implements Printer {  
    void print_data() {  
        print("_____Printing to Console_____");  
    }  
}
```

Debería producir el siguiente **resultado** :

_____Printing to Console_____

Implementando múltiples interfaces

Una clase puede implementar múltiples interfaces. Las interfaces están separadas por una coma. La **sintaxis** para el mismo se da a continuación:

```
class identifier implements interface-  
1,interface_2,interface_4......
```

El siguiente **ejemplo** muestra cómo puede implementar múltiples interfaces en Dart:

```
void main() {  
    Calculator c = new Calculator();  
    print("The gross total : ${c.ret_tot()}");  
    print("Discount :${c.ret_dis()}");  
}  
class Calculate_Total {  
    int ret_tot() {}  
}  
class Calculate_Discount {  
    int ret_dis() {}  
}  
class Calculator implements  
Calculate_Total,Calculate_Discount {  
    int ret_tot() {  
        return 1000;  
    }  
    int ret_dis() {  
        return 50;  
    }  
}
```

Debería producir el siguiente **resultado** :

```
The gross total: 1000  
Discount:50
```

Programación de Dart - Clases

Dart es un lenguaje orientado a objetos. Admite funciones de programación orientada a objetos como clases, interfaces, etc. Una **clase** en términos de OOP es un plan para crear objetos. Una **clase** encapsula datos para el objeto. Dart brinda soporte integrado para este concepto llamado **clase** .

Declarando una clase

Use la palabra clave de **clase** para declarar una **clase** en Dart. Una definición de clase comienza con la clase de palabra clave seguida del **nombre de la clase** ; y el cuerpo de la clase encerrado por un par de llaves. La sintaxis para el mismo se da a continuación:

Sintaxis

```
class class_name {  
    <fields>
```

```
<getters/setters>
<constructors>
<functions>
}
```

La palabra clave de **class** es seguida por el nombre de la clase. Las reglas para los identificadores deben tenerse en cuenta al nombrar una clase.

Una definición de clase puede incluir lo siguiente:

- **Campos** : un campo es cualquier variable declarada en una clase. Los campos representan datos pertenecientes a objetos.
- **Setters and Getters** : permite que el programa inicialice y recupere los valores de los campos de una clase. Un getter / setter predeterminado está asociado con cada clase. Sin embargo, los predeterminados se pueden anular definiendo explícitamente un setter / getter.
- **Constructores** : responsables de asignar memoria para los objetos de la clase.
- **Funciones** : las funciones representan acciones que un objeto puede realizar. A veces también se les conoce como métodos.

Estos componentes juntos se denominan **miembros** de **datos** de la clase.

Ejemplo: declarar una clase

```
class Car {
    // field
    String engine = "E1001";

    // function
    void disp() {
        print(engine);
    }
}
```

El ejemplo declara una clase **Car**. La clase tiene un campo llamado **motor**. El **disp ()** es una función simple que imprime el valor del **motor** de campo.

Crear instancia de la clase

Para crear una instancia de la clase, use la **nueva** palabra clave seguida del nombre de la clase. La sintaxis para el mismo se da a continuación:

Sintaxis

```
var object_name = new class_name([ arguments ])
```

- La **nueva** palabra clave es responsable de la instanciación.
- El lado derecho de la expresión invoca al constructor. El constructor debe pasar valores si está parametrizado.

Ejemplo: instanciar una clase

```
var obj = new Car("Engine 1")
```

Acceso a atributos y funciones

Se puede acceder a los atributos y funciones de una clase a través del objeto. Utilizar el '.' notación de puntos (llamada como **punto**) para acceder a los miembros de datos de una clase.

```
//accessing an attribute  
obj.field_name  
  
//accessing a function  
obj.function_name()
```

Ejemplo

Eche un vistazo al siguiente ejemplo para comprender cómo acceder a los atributos y funciones en Dart:

```
void main() {  
    Car c= new Car();  
    c.disp();  
}  
class Car {  
    // field  
    String engine = "E1001";  
  
    // function  
    void disp() {  
        print(engine);  
    }  
}
```

La **salida** del código anterior es la siguiente:

E1001

Constructores de Dart

Un constructor es una función especial de la clase que se encarga de inicializar las variables de la clase. Dart define un constructor con el mismo nombre que el de la clase. Un constructor es una función y, por lo tanto, puede parametrizarse. Sin embargo, a diferencia de una función, los constructores no pueden tener un tipo de retorno. Si no declara un constructor, se le proporciona un constructor predeterminado **sin argumentos** .

Sintaxis

```
Class_name(parameter_list) {  
    //constructor body
```

```
}
```

Ejemplo

El siguiente ejemplo muestra cómo usar constructores en Dart:

```
void main() {  
    Car c = new Car('E1001');  
}  
class Car {  
    Car(String engine) {  
        print(engine);  
    }  
}
```

Debería producir el siguiente **resultado** :

```
E1001
```

Constructores nombrados

Dart proporciona **constructores** con **nombre** para permitir que una clase defina **múltiples constructores** . La sintaxis de los constructores con nombre es la siguiente:

Sintaxis: Definiendo el constructor

```
Class_name.constructor_name(param_list)
```

Ejemplo

El siguiente ejemplo muestra cómo puede usar constructores con nombre en Dart:

```
void main() {  
    Car c1 = new Car.namedConst('E1001');  
    Car c2 = new Car();  
}  
class Car {  
    Car() {  
        print("Non-parameterized constructor invoked");  
    }  
    Car.namedConst(String engine) {  
        print("The engine is : ${engine}");  
    }  
}
```

Debería producir el siguiente **resultado** :

```
The engine is : E1001  
Non-parameterized constructor invoked
```


La esta palabra clave

La palabra clave **this** se refiere a la instancia actual de la clase. Aquí, el nombre del parámetro y el nombre del campo de la clase son los mismos. Por lo tanto, para evitar ambigüedades, el campo de la clase tiene como prefijo la palabra clave **this**. El siguiente ejemplo explica lo mismo:

Ejemplo

El siguiente ejemplo explica cómo usar **esta** palabra clave en Dart:

```
void main() {  
    Car c1 = new Car('E1001');  
}  
class Car {  
    String engine;  
    Car(String engine) {  
        this.engine = engine;  
        print("The engine is : ${engine}");  
    }  
}
```

Debería producir el siguiente **resultado** :

```
The engine is : E1001
```

Dart Class — Getters and Setters

Getters y setters, también llamados como **descriptores de acceso y mutadores**, permiten que el programa para inicializar y recuperar los valores de los campos de clase, respectivamente. Los captadores o accesorios se definen usando la palabra clave **get**. Los definidores o mutadores se definen usando la palabra clave **set**.

Un getter / setter predeterminado está asociado con cada clase. Sin embargo, los predeterminados se pueden anular definiendo explícitamente un setter / getter. Un getter no tiene parámetros y devuelve un valor, y el setter tiene un parámetro y no devuelve un valor.

Sintaxis: definir un captador

```
Return_type get identifier  
{  
}
```

Sintaxis: definición de un setter

```
set identifier  
{  
}
```

Ejemplo

El siguiente ejemplo muestra cómo puede usar **getters** y **setters** en una clase Dart:

```
class Student {
    String name;
    int age;

    String get stud_name {
        return name;
    }

    void set stud_name(String name) {
        this.name = name;
    }

    void set stud_age(int age) {
        if(age<= 0) {
            print("Age should be greater than 5");
        } else {
            this.age = age;
        }
    }

    int get stud_age {
        return age;
    }
}

void main() {
    Student s1 = new Student();
    s1.stud_name = 'MARK';
    s1.stud_age = 0;
    print(s1.stud_name);
    print(s1.stud_age);
}
```

Este código de programa debería producir el siguiente **resultado** :

```
Age should be greater than 5
MARK
Null
```

Herencia de clase

Dart admite el concepto de herencia, que es la capacidad de un programa para crear nuevas clases a partir de una clase existente. La clase que se extiende para crear clases más nuevas se llama clase principal / superclase. Las clases recién creadas se denominan clases secundarias / secundarias.

Una clase hereda de otra clase usando la palabra clave 'extiende'. **Las clases secundarias heredan todas las propiedades y métodos, excepto los constructores de la clase primaria .**

Sintaxis

```
class child_class_name extends parent_class_name
```

Nota : Dart no admite herencia múltiple.

Ejemplo: herencia de clase

En el siguiente ejemplo, estamos declarando una clase **Shape** . La clase se extiende por la clase **Circle** . Dado que existe una relación de herencia entre las clases, la clase secundaria, es decir, la clase **Car** obtiene un acceso implícito a su miembro de datos de clase primaria.

```
void main() {
    var obj = new Circle();
    obj.cal_area();
}
class Shape {
    void cal_area() {
        print("calling calc area defined in the Shape class");
    }
}
class Circle extends Shape {}
```

Debería producir el siguiente **resultado** :

```
calling calc area defined in the Shape class
```

Tipos de herencia

La herencia puede ser de los siguientes tres tipos:

- **Individual** : cada clase puede, como máximo, extenderse desde una clase principal.
- **Múltiple** : una clase puede heredar de varias clases. Dart no admite herencia múltiple.
- **Multinivel** : una clase puede heredar de otra clase secundaria.

Ejemplo

El siguiente ejemplo muestra cómo funciona la herencia multinivel:

```
void main() {
    var obj = new Leaf();
    obj.str = "hello";
    print(obj.str);
}
```

```

}
class Root {
    String str;
}
class Child extends Root {}
class Leaf extends Child {}
//indirectly inherits from Root by virtue of inheritance

```

La clase **Leaf** deriva los atributos de las clases Root y Child en virtud de la herencia multinivel. Su **salida** es la siguiente:

```
hello
```

Dart - Herencia de clase y anulación de método

La anulación de métodos es un mecanismo por el cual la clase secundaria redefine un método en su clase principal. El siguiente ejemplo ilustra lo mismo:

Ejemplo

```

void main() {
    Child c = new Child();
    c.m1(12);
}
class Parent {
    void m1(int a){ print("value of a ${a}");}
}
class Child extends Parent {
    @override
    void m1(int b) {
        print("value of b ${b}");
    }
}

```

Debería producir el siguiente **resultado** :

```
value of b 12
```

El número y el tipo de los parámetros de la función deben coincidir al anular el método. En caso de una falta de coincidencia en el número de parámetros o su tipo de datos, el compilador Dart arroja un error. La siguiente ilustración explica lo mismo:

```

import 'dart:io';
void main() {
    Child c = new Child();
    c.m1(12);
}
class Parent {
    void m1(int a){ print("value of a ${a}");}
}

```

```
class Child extends Parent {
    @Override
    void m1(String b) {
        print("value of b ${b}");
    }
}
```

Debería producir el siguiente **resultado** :

value of b 12

La palabra clave estática

La palabra clave **estática** se puede aplicar a los miembros de datos de una clase, es decir, **campos** y **métodos** . Una variable estática conserva sus valores hasta que el programa finaliza la ejecución. Los miembros estáticos son referenciados por el nombre de la clase.

Ejemplo

```
class StaticMem {
    static int num;
    static disp() {
        print("The value of num is ${StaticMem.num}") ;
    }
}
void main() {
    StaticMem.num = 12;
    // initialize the static variable }
    StaticMem.disp();
    // invoke the static method
}
```

Debería producir el siguiente **resultado** :

The value of num is 12

La súper palabra clave

La palabra clave **super** se usa para referirse al padre inmediato de una clase. La palabra clave se puede usar para referirse a la versión de superclase de una **variable**, **propiedad** o **método** . El siguiente ejemplo ilustra lo mismo:

Ejemplo

```
void main() {
    Child c = new Child();
    c.m1(12);
}
```

```

class Parent {
    String msg = "message variable from the parent class";
    void m1(int a){ print("value of a ${a}");}
}
class Child extends Parent {
    @override
    void m1(int b) {
        print("value of b ${b}");
        super.m1(13);
        print("${super.msg}")    ;
    }
}

```

Debería producir el siguiente **resultado** :

```

value of b 12
value of a 13
message variable from the parent class

```

Programación de Dart - Objeto

La programación orientada a objetos define un objeto como "cualquier entidad que tenga un límite definido". Un objeto tiene lo siguiente:

- **Estado** : describe el objeto. Los campos de una clase representan el estado del objeto.
- **Comportamiento** : describe lo que puede hacer un objeto.
- **Identidad** : un valor único que distingue un objeto de un conjunto de otros objetos similares. Dos o más objetos pueden compartir el estado y el comportamiento, pero no la identidad.

El operador de período (.) Se usa junto con el objeto para acceder a los miembros de datos de una clase.

Ejemplo

Dart representa datos en forma de objetos. Cada clase en Dart extiende la clase Object. A continuación se muestra un ejemplo simple de creación y uso de un objeto.

```

class Student {
    void test_method() {
        print("This is a test method");
    }

    void test_method1() {
        print("This is a test method1");
    }
}

void main() {
    Student s1 = new Student();
    s1.test_method();
    s1.test_method1();
}

```

```
}
```

Debería producir el siguiente **resultado** :

```
This is a test method  
This is a test method1
```

El operador Cascade (..)

El ejemplo anterior invoca los métodos en la clase. Sin embargo, cada vez que se llama a una función, se requiere una referencia al objeto. El **operador en cascada** se puede utilizar como una abreviatura en casos donde hay una secuencia de invocaciones.

El operador en cascada (..) se puede utilizar para emitir una secuencia de llamadas a través de un objeto. El ejemplo anterior se puede reescribir de la siguiente manera.

```
class Student {  
    void test_method() {  
        print("This is a test method");  
    }  
  
    void test_method1() {  
        print("This is a test method1");  
    }  
}  
void main() {  
    new Student()  
    ..test_method()  
    ..test_method1();  
}
```

Debería producir el siguiente **resultado** :

```
This is a test method  
This is a test method1
```

El método toString ()

Esta función devuelve una representación de cadena de un objeto. Eche un vistazo al siguiente ejemplo para comprender cómo usar el método **toString** .

```
void main() {  
    int n = 12;  
    print(n.toString());  
}
```

Debería producir el siguiente **resultado** :

12

Programación de Dart - Colección

Dart, a diferencia de otros lenguajes de programación, no admite matrices. Las colecciones de Dart se pueden usar para replicar estructuras de datos como una matriz. La biblioteca `dart: core` y otras clases permiten el soporte de la Colección en scripts Dart.

Las colecciones de Dart se pueden clasificar básicamente como:

No Señor	Colección de Dart y descripción
1	<u>Lista</u> Una lista es simplemente un grupo ordenado de objetos. La biblioteca dart: core proporciona la clase <code>Lista</code> que permite la creación y manipulación de listas. <ul style="list-style-type: none">• Lista de longitud fija : la longitud de la lista no puede cambiar en tiempo de ejecución.• Lista de crecimiento : la longitud de la lista puede cambiar en tiempo de ejecución.
2	<u>Conjunto</u> Set representa una colección de objetos en los que cada objeto puede aparecer solo una vez. La biblioteca <code>dart: core</code> proporciona la clase <code>Set</code> para implementar lo mismo.
3	<u>Mapas</u> El objeto <code>Map</code> es un simple par clave / valor. Las claves y los valores en un mapa pueden ser de cualquier tipo. Un mapa es una colección dinámica. En otras palabras, los mapas pueden crecer y reducirse en tiempo de ejecución. La clase <code>Map</code> en la biblioteca <code>dart: core</code> proporciona soporte para lo mismo.
4 4	<u>Cola</u> Una cola es una colección que se puede manipular en ambos extremos. Las colas son útiles cuando desea crear una colección de primero en entrar, primero en salir. En pocas palabras, una cola inserta datos de un extremo y los elimina de otro. Los valores se eliminan / leen en el orden de su inserción.

Colecciones iterativas

La clase `Iterator` de la biblioteca **dart: core** permite un recorrido de recolección fácil. Cada colección tiene una propiedad **iteradora** . Esta propiedad devuelve un iterador que apunta a los objetos en la colección.

Ejemplo

El siguiente ejemplo ilustra el recorrido de una colección utilizando un objeto iterador.

```
importar 'dart: colección' ; void main () { Queue numQ = new
Queue ();
    numQ . addAll ([ 100 , 200 , 300 ]); Iterador i = numQ .
iterador ;

    while ( i . moveNext ()) { print ( i . current ); } }
```

La función **moveNext ()** devuelve un valor booleano que indica si hay una entrada posterior. La propiedad **actual** del objeto iterador devuelve el valor del objeto al que apunta el iterador actualmente.

Este programa debería producir el siguiente **resultado** :

```
100
200
300
```

Programación de Dart - Genéricos

Dart es un **lenguaje opcionalmente escrito** . Las colecciones en Dart son heterogéneas por defecto. En otras palabras, una sola colección Dart puede alojar valores de varios tipos. Sin embargo, se puede hacer una colección Dart para mantener valores homogéneos. El concepto de genéricos se puede utilizar para lograr lo mismo.

El uso de genéricos impone una restricción en el tipo de datos de los valores que puede contener la colección. Dichas colecciones se denominan colecciones de tipo seguro. La seguridad de tipo es una función de programación que garantiza que un bloque de memoria solo pueda contener datos de un tipo de datos específico.

Todas las colecciones Dart admiten la implementación de seguridad de tipos mediante genéricos. Se utiliza un par de corchetes angulares que contienen el tipo de datos para declarar una colección de tipo seguro. La sintaxis para declarar una colección de tipo seguro es la siguiente.

Sintaxis

```
Collection_name <data_type> identifier= new
Collection_name<data_type>
```

Las implementaciones de tipo seguro de Lista, Mapa, Conjunto y Cola se dan a continuación. Esta característica también es compatible con todas las implementaciones de los tipos de colección mencionados anteriormente.

Ejemplo: lista genérica

```

void main () { List < String > logTypes = new List < String >
();
    logTypes . agregar ( "ADVERTENCIA" );
    logTypes . agregar ( "ERROR" );
    logTypes . agregar ( "INFO" );

    // iterar en la lista para ( Tipo de cadena en logTypes )
{ print ( type ); } }

```

Debería producir el siguiente **resultado** :

```

WARNING
ERROR
INFO

```

Un intento de insertar un valor que no sea el tipo especificado dará como resultado un error de compilación. El siguiente ejemplo lo ilustra.

Ejemplo

```

void main() {
    List <String> logTypes = new List <String>();
    logTypes.add(1);
    logTypes.add("ERROR");
    logTypes.add("INFO");

    //iterating across list
    for (String type in logTypes) {
        print(type);
    }
}

```

Debería producir el siguiente **resultado** :

```

1
ERROR
INFO

```

Ejemplo: conjunto genérico

```

void main() {
    Set <int>numberSet = new Set<int>();
    numberSet.add(100);
    numberSet.add(20);
    numberSet.add(5);
    numberSet.add(60);
}

```

```

numberSet.add(70);

// numberSet.add("Tom");
// compilation error;
print("Default implementation
:${numberSet.runtimeType}");

for(var no in numberSet) {
    print(no);
}
}

```

Debería producir el siguiente **resultado** :

```

Default implementation :_CompactLinkedHashSet<int>
100
20
5
60
70

```

Ejemplo: cola genérica

```

importar 'dart: colección' ; void main () { Queue <int> queue
= new Queue <int> (); print ( "Implementación predeterminada
$ {queue.runtimeType}" );
cola . addLast ( 10 );
cola . addLast ( 20 );
cola . addLast ( 30 );
cola . addLast ( 40 );
cola . removeFirst ();

para ( int no en la cola ) { print ( no ); } }

```

Debería producir el siguiente **resultado** :

```

Default implementation ListQueue<int>
20
30
40

```

Mapa genérico

Una declaración de mapa de tipo seguro especifica los tipos de datos de -

- La clave
- El valor

Sintaxis

Map <Key_type, value_type>

Ejemplo

```
void main () { Map < String , String > m = { 'name' : 'Tom' ,  
'Id' : 'E1001' }; print ( 'Mapa: $ {m}' ); }
```

Debería producir el siguiente **resultado** :

```
Map :{name: Tom, Id: E1001}
```

Programación de Dart - Paquetes

Un paquete es un mecanismo para encapsular un grupo de unidades de programación. En ocasiones, las aplicaciones pueden necesitar la integración de algunas bibliotecas o complementos de terceros. Cada idioma tiene un mecanismo para administrar paquetes externos como Maven o Gradle para Java, Nuget para .NET, npm para Node.js, etc. El administrador de paquetes para Dart es **pub** .

Pub ayuda a instalar paquetes en el repositorio. El repositorio de paquetes alojados se puede encontrar en <https://pub.dartlang.org/>.

Los **metadatos del paquete** se definen en un archivo, **pubsec.yaml** . YAML es el acrónimo de **Yet Another Markup Language** . La herramienta de **publicación** se puede utilizar para descargar todas las bibliotecas que requiere una aplicación.

Cada aplicación Dart tiene un archivo **pubspec.yaml** que contiene las dependencias de la aplicación a otras bibliotecas y metadatos de aplicaciones como el nombre de la aplicación, el autor, la versión y la descripción.

El contenido de un archivo **pubspec.yaml** debería verse así:

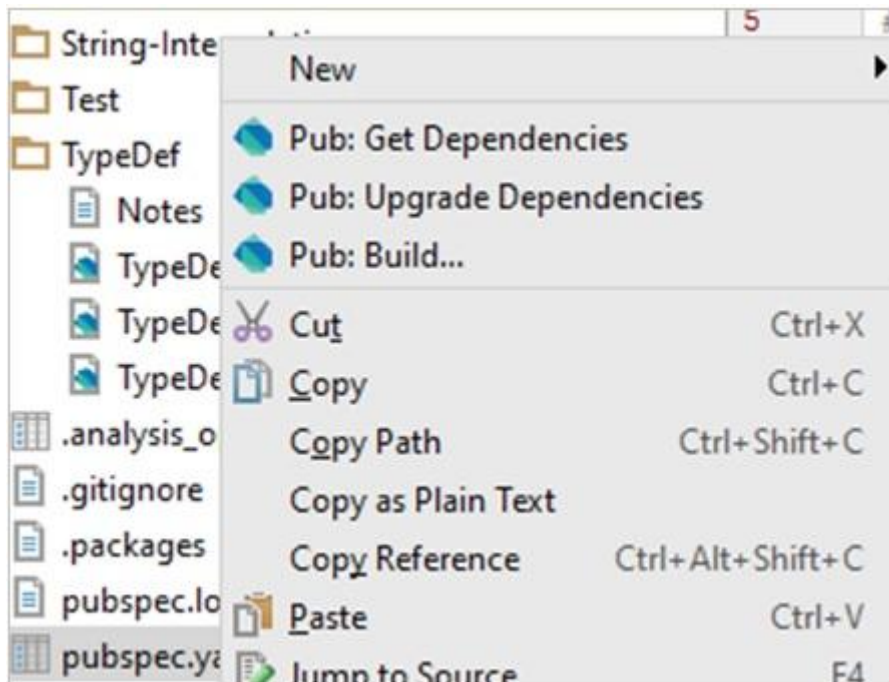
```
name: 'vector_victor'  
version: 0.0.1  
description: An absolute bare-bones web app.  
...  
dependencies: browser: '>=0.10.0 <0.11.0'
```

Los **comandos** importantes de **pub** son los siguientes:

No Señor	Comando y descripción
1	'pub get' Ayuda a obtener todos los paquetes de los que depende su aplicación.

2	'actualización de pub' Actualiza todas sus dependencias a una versión más nueva.
3	'construcción de pub' Esto se usa para compilar su aplicación web y creará una carpeta de compilación, con todos los scripts relacionados.
4 4	'ayuda de pub' Esto le dará ayuda para todos los diferentes comandos de pub.

Si está utilizando un IDE como WebStorm, puede hacer clic derecho en pubspec.yaml para obtener todos los comandos directamente:



Instalar un paquete

Considere un ejemplo en el que una aplicación necesita analizar xml. Dart XML es una biblioteca liviana de código abierto y estable para analizar, recorrer, consultar y crear documentos XML.

Los pasos para lograr dicha tarea son los siguientes:

Paso 1 : agregue lo siguiente al archivo pubsec.yaml.

```
name: TestApp
version: 0.0.1
description: A simple console application.
#dependencies:
#  foo_bar: '>=1.0.0 <2.0.0'
```

```
dependencies:  
https://mail.google.com/mail/u/0/images/cleardot.gif  
xml:
```

Haga clic derecho en **pubsec.yaml** y obtenga dependencias. Esto activará internamente el **comando get de pub** como se muestra a continuación.

```
Resolving dependencies...  
+ petitparser 1.5.3 (1.5.4 available)  
+ xml 2.4.3 (2.4.4 available)  
Downloading xml 2.4.3...  
Downloading petitparser 1.5.3...  
Changed 2 dependencies!  
Process finished with exit code 0
```

Los paquetes descargados y sus paquetes dependientes se pueden verificar en la carpeta de paquetes.



Como la instalación se ha completado ahora, necesitamos referirnos al **dart xml** en el proyecto. La sintaxis es la siguiente:

```
importar 'paquete: xml / xml.dart' como xml;
```

Leer cadena XML

Para leer una cadena XML y verificar la entrada, Dart XML utiliza un método **parse ()**. La sintaxis es la siguiente:

```
xml.parse(String input):
```

Ejemplo: análisis de entrada de cadena XML

El siguiente ejemplo muestra cómo analizar la entrada de cadena XML:

```
importar 'paquete: xml / xml.dart' como xml ; void main () {  
  print ( "xml" ); var bookshelfXml = ' ' <? xml version =  
  "1.0"?>  
  
  <estantería de libros>  
    <libro>  
      <title lang = "english"> Cultivar un idioma </title>  
      <price> 29.99 </price>  
    </book>  
  
    <libro>  
      <title lang = "english"> Aprendiendo XML </title>
```

```
        <price> 39.95 </price>
    </book>
    <price> 132.00 </price>
</bookshelf> ' ' ' ;

    documento var = xml . analizar ( estanteríaXml ); print (
document . toString ()); }
```

Debería producir el siguiente **resultado** :

```
xml
<?xml version = "1.0"?><bookshelf>
  <book>
    <title lang = "english">Growing a Language</title>
    <price>29.99</price>
  </book>

  <book>
    <title lang = "english">Learning XML</title>
    <price>39.95</price>
  </book>
  <price>132.00</price>
</bookshelf>
```

Programación de Dart: excepciones

Una excepción (o evento excepcional) es un problema que surge durante la ejecución de un programa. Cuando ocurre una excepción, el flujo normal del programa se interrumpe y el programa / aplicación finaliza de manera anormal.

Las excepciones de Dart incorporadas incluyen:

No Señor	Excepciones y descripción
1	DeferredLoadException Se lanza cuando una biblioteca diferida no se carga.
2	FormatException Se produce una excepción cuando una cadena o algún otro dato no tiene un formato esperado y no se puede analizar ni procesar.
3	IntegerDivisionByZeroException Lanzado cuando un número se divide por cero.
4 4	IOException Clase base para todas las excepciones relacionadas con Inupt-Output.

5 5	IsolateSpawnException Lanzado cuando no se puede crear un aislamiento.
6 6	Se acabó el tiempo Se produce cuando ocurre un tiempo de espera programado mientras se espera un resultado asíncrono.

Cada excepción en Dart es un subtipo de la **excepción** de clase predefinida. Se deben manejar las excepciones para evitar que la aplicación finalice abruptamente.

Los bloques try / on / catch

El bloque **try** incrusta el código que posiblemente podría dar lugar a una excepción. El bloque **on** se utiliza cuando se debe especificar el tipo de excepción. El bloque **catch** se usa cuando el controlador necesita el objeto de excepción.

El bloque de **prueba** debe ser seguido por un bloque de **encendido / capturado** o un bloque **finalmente** (o uno de ambos). Cuando ocurre una excepción en el bloque try, el control se transfiere a la **captura**.

La **sintaxis** para manejar una excepción es la siguiente:

```
try {
    // code that might throw an exception
}
on Exception1 {
    // code for handling exception
}
catch Exception2 {
    // code for handling exception
}
```

Los siguientes son algunos puntos para recordar:

- Un fragmento de código puede tener más de un bloque on / catch para manejar múltiples excepciones.
- El bloque on y el bloque catch son mutuamente inclusivos, es decir, un bloque try puede asociarse tanto con el bloque on como con el bloque catch.

El siguiente código ilustra el manejo de excepciones en Dart:

Ejemplo: uso del bloque ON

El siguiente programa divide dos números representados por las variables **x** e **y** respectivamente. El código arroja una excepción ya que intenta la división por cero. El **bloque on** contiene el código para manejar esta excepción.


```

main() {
    int x = 12;
    int y = 0;
    int res;

    try {
        res = x ~/ y;
    }
    on IntegerDivisionByZeroException {
        print('Cannot divide by zero');
    }
}

```

Debería producir el siguiente **resultado** :

Cannot divide by zero

Ejemplo: uso del bloque catch

En el siguiente ejemplo, hemos usado el mismo código que el anterior. La única diferencia es que el **bloque catch** (en lugar del bloque ON) aquí contiene el código para manejar la excepción. El parámetro de **captura** contiene el objeto de excepción lanzado en tiempo de ejecución.

```

main() {
    int x = 12;
    int y = 0;
    int res;

    try {
        res = x ~/ y;
    }
    catch(e) {
        print(e);
    }
}

```

Debería producir el siguiente **resultado** :

IntegerDivisionByZeroException

Ejemplo: en ... captura

El siguiente ejemplo muestra cómo usar el bloque **on ... catch** .

```

main() {
    int x = 12;
    int y = 0;
    int res;

    try {
        res = x ~/ y;
    }
    on ... catch {
    }
}

```

```
    }  
    on IntegerDivisionByZeroException catch(e) {  
        print(e);  
    }  
}
```

Debería producir el siguiente **resultado** :

IntegerDivisionByZeroException

El finalmente bloque

El **último** bloque incluye código que debe ejecutarse independientemente de la ocurrencia de una excepción. El bloque opcional **finalmente** se ejecuta incondicionalmente después de **try / on / catch** .

La sintaxis para usar el bloque **finalmente** es la siguiente:

```
try {  
    // code that might throw an exception  
}  
on Exception1 {  
    // exception handling code  
}  
catch Exception2 {  
    // exception handling  
}  
finally {  
    // code that should always execute; irrespective of the  
    exception  
}
```

El siguiente ejemplo ilustra el uso de **finalmente** block.

```
main() {  
    int x = 12;  
    int y = 0;  
    int res;  
  
    try {  
        res = x ~/ y;  
    }  
    on IntegerDivisionByZeroException {  
        print('Cannot divide by zero');  
    }  
    finally {  
        print('Finally block executed');  
    }  
}
```

Debería producir el siguiente **resultado** :

Cannot divide by zero
Finally block executed

Lanzar una excepción

La palabra clave **throw** se usa para generar explícitamente una excepción. Se debe manejar una excepción planteada para evitar que el programa salga abruptamente.

La **sintaxis** para generar una excepción explícitamente es:

```
throw new Exception_name()
```

Ejemplo

El siguiente ejemplo muestra cómo utilizar el **throw** de palabras clave para lanzar una excepción -

```
main() {  
  try {  
    test_age(-2);  
  }  
  catch(e) {  
    print('Age cannot be negative');  
  }  
}  
void test_age(int age) {  
  if(age<0) {  
    throw new FormatException();  
  }  
}
```

Debería producir el siguiente **resultado** :

```
Age cannot be negative
```

Excepciones personalizadas

Como se especificó anteriormente, cada tipo de excepción en Dart es un subtipo de la clase integrada **Exception**. Dart permite crear excepciones personalizadas ampliando las existentes. La sintaxis para definir una excepción personalizada es la siguiente:

Sintaxis: Definiendo la Excepción

```
class Custom_exception_Name implements Exception {  
  // can contain constructors, variables and methods  
}
```

Las excepciones personalizadas deben generarse explícitamente y lo mismo debe manejarse en el código.

Ejemplo

El siguiente ejemplo muestra cómo definir y manejar una excepción personalizada.

```
class AmtException implements Exception {
    String errMsg() => 'Amount should be greater than zero';
}
void main() {
    try {
        withdraw_amt(-1);
    }
    catch(e) {
        print(e.errMsg());
    }
    finally {
        print('Ending requested operation.....');
    }
}
void withdraw_amt(int amt) {
    if (amt <= 0) {
        throw new AmtException();
    }
}
```

En el código anterior, estamos definiendo una excepción personalizada, **AmtException** . El código genera la excepción si la cantidad aprobada no está dentro del rango exceptuado. La función **principal** encierra la invocación de la función en el bloque **try ... catch** .

El código debe producir el siguiente **resultado** :

```
Amount should be greater than zero
Ending requested operation....
```

Programación de Dart - Depuración

De vez en cuando, los desarrolladores cometen errores al codificar. Un error en un programa se conoce como error. El proceso de encontrar y corregir errores se llama depuración y es una parte normal del proceso de desarrollo. Esta sección cubre herramientas y técnicas que pueden ayudarlo con las tareas de depuración.

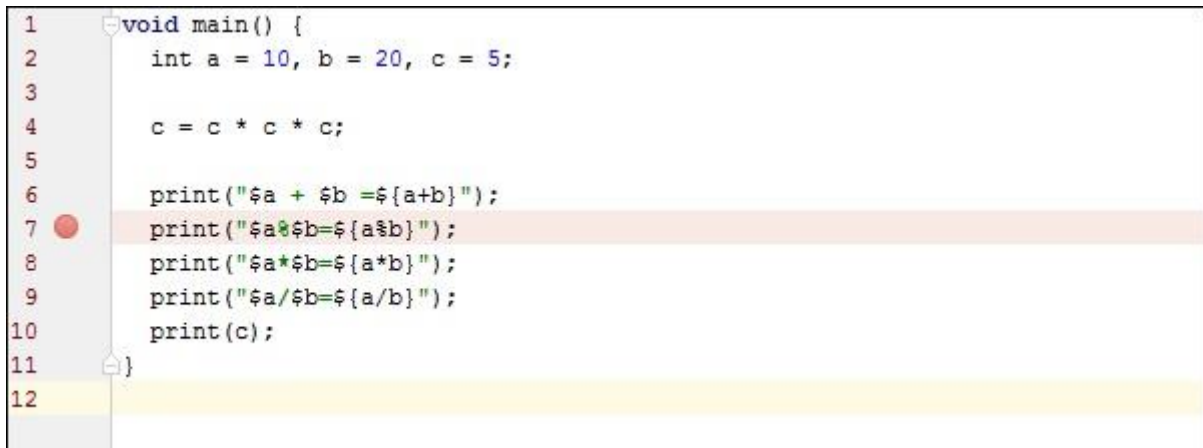
El editor de WebStorm habilita puntos de interrupción y depuración paso a paso. El programa se interrumpirá en el punto donde se adjunta el punto de interrupción. Esta funcionalidad es similar a la que se puede esperar del desarrollo de aplicaciones Java o C #. Puede ver variables, explorar la pila, pasar y entrar en llamadas a métodos y funciones, todo desde el Editor de WebStorm.

Agregar un punto de interrupción

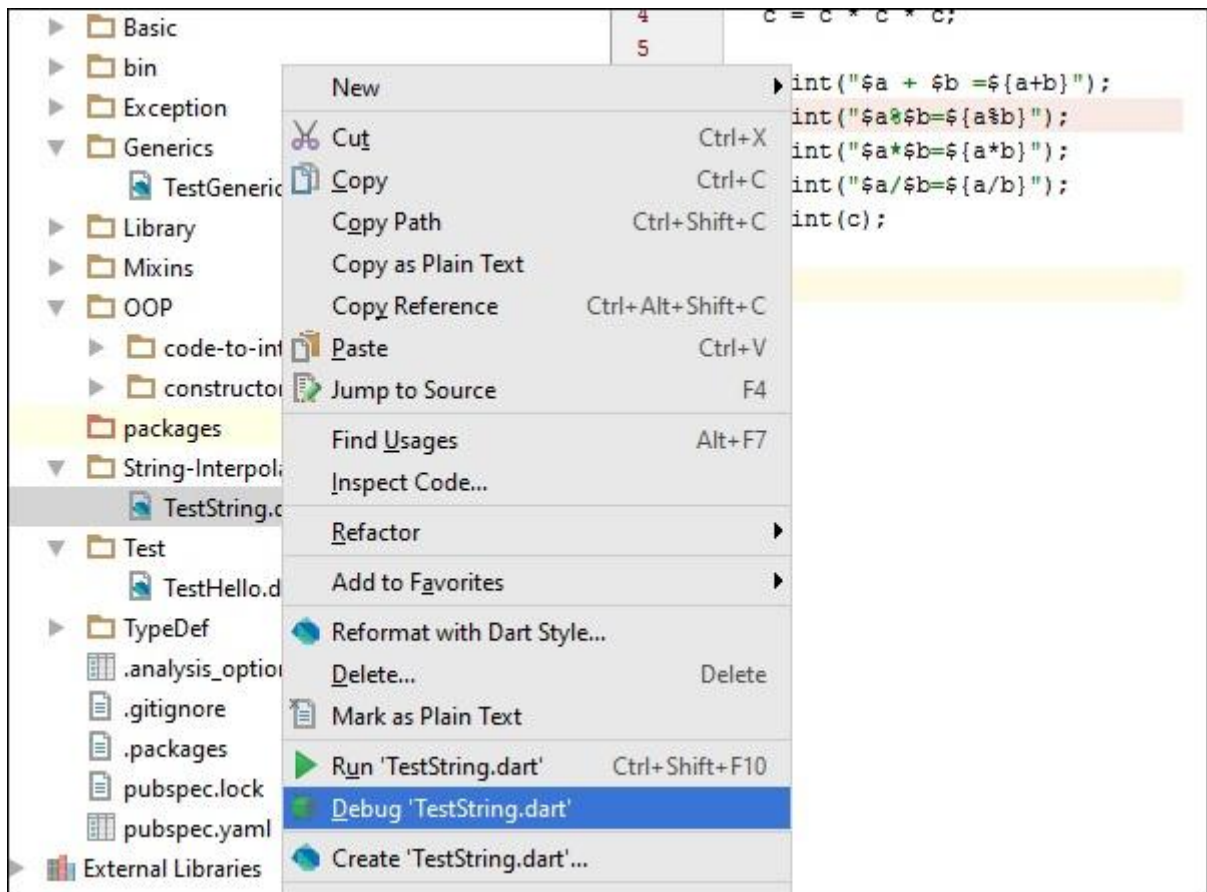
Considere el siguiente fragmento de código. (**TestString.dart**)

```
void main() {  
    int a = 10, b = 20, c = 5;  
    c = c * c * c;  
  
    print("$a + $b = ${a+b}");  
    print("$a%$b = ${a%b}"); // Add a break point here  
    print("$a*$b = ${a*b}");  
    print("$a/$b = ${a/b}");  
    print(c);  
}
```

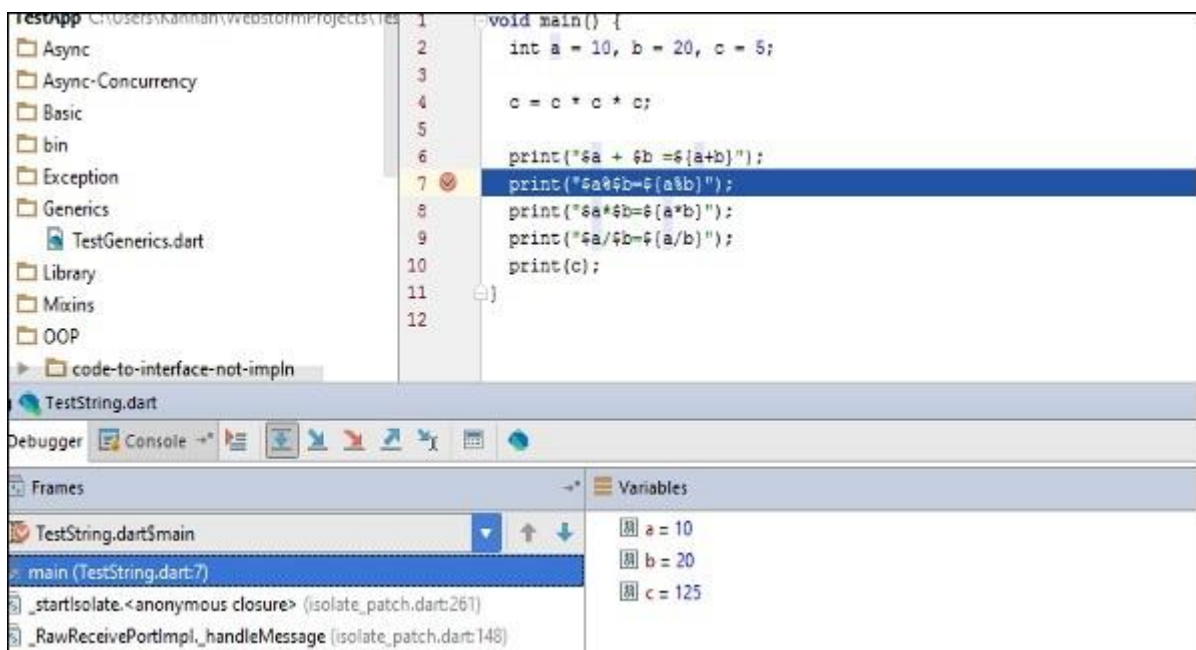
Para **agregar un punto de interrupción**, haga clic en el margen izquierdo para. En la figura siguiente, la línea número 7 tiene un punto de ruptura.



Ejecute el programa en modo de depuración. En el explorador de proyectos, haga clic derecho en el programa de Dart en nuestro caso TestString.dart.



Una vez que el programa se ejecuta en modo de depuración, obtendrá la ventana del depurador como se muestra en la siguiente captura de pantalla. La pestaña de variables muestra los valores de las variables en el contexto actual. Puede agregar observadores para variables específicas y escuchar los cambios de valores utilizando la ventana de relojes.



El ícono de flecha **Paso a paso** (F7) en el menú de depuración ayuda a ejecutar el código una declaración a la vez. Si los métodos principales llaman a una subrutina, esto también irá al código de la subrutina.

Paso a paso (F8): es similar a **Paso a paso** . La diferencia en el uso ocurre cuando la declaración actual contiene una llamada a una subrutina. Si el método principal llama a una subrutina, el paso no profundizará en la subrutina. saltará la subrutina.

Step Out (Shift + F8): ejecuta las líneas restantes de una función en la que se encuentra el punto de ejecución actual. La siguiente instrucción que se muestra es la instrucción que sigue a la llamada a la subrutina.

Después de ejecutarse en modo de depuración, el programa proporciona el siguiente **resultado** :

```
10 + 20 = 30
10 % 20 = 10
10 * 20 = 200
10 / 20 = 0.5
125
```

Programación de Dart - Typedef

Un **typedef** , o un alias de tipo de función, ayuda a definir punteros al código ejecutable dentro de la memoria. En pocas palabras, un **typedef** se puede usar como un puntero que hace referencia a una función.

A continuación se **detallan** los pasos para implementar **typedefs** en un programa Dart.

Paso 1: Definir un typedef

Un **typedef** se puede usar para especificar una firma de función que queremos que coincida con funciones específicas. Una firma de función se define por los parámetros de una función (incluidos sus tipos). El tipo de retorno no es parte de la firma de la función. Su sintaxis es la siguiente.

```
typedef function_name(parameters)
```

Paso 2: Asignación de una función a una variable typedef

Una variable de **typedef** puede apuntar a cualquier función que tenga la misma firma que **typedef** . Puede usar la siguiente firma para asignar una función a una variable **typedef** .

```
type_def var_name = function_name
```

Paso 3: invocar una función

La variable **typedef** se puede usar para invocar funciones. Así es como puede invocar una función:

```
var_name(parameters)
```

Ejemplo

Tomemos ahora un ejemplo para comprender más sobre **typedef** en Dart.

Al principio, definamos un **typedef** . Aquí estamos definiendo una firma de función. La función tomará dos parámetros de entrada del tipo **entero** . El tipo de retorno no forma parte de la firma de la función.

```
typedef ManyOperation(int firstNo , int secondNo); //function signature
```

A continuación, definamos las funciones. Defina algunas funciones con la misma firma de función que la del tipo de **definición ManyOperation** .

```
Add(int firstNo,int second){
    print("Add result is ${firstNo+second}");
}
Subtract(int firstNo,int second){
    print("Subtract result is ${firstNo-second}");
}
Divide(int firstNo,int second){
    print("Add result is ${firstNo/second}");
}
```

Finalmente, invocaremos la función a través de **typedef** . Declare una variable del tipo ManyOperations. Asigne el nombre de la función a la variable declarada.

```
ManyOperation oper ;

//can point to any method of same signature
oper = Add;
oper(10,20);
oper = Subtract;
oper(30,20);
oper = Divide;
oper(50,5);
```

La variable **oper** puede apuntar a cualquier método que tome dos parámetros enteros. La referencia de la función **Agregar** se asigna a la variable. Typedefs puede cambiar referencias de funciones en tiempo de ejecución

Pongamos ahora todas las partes juntas y veamos el programa completo.

```
typedef ManyOperation(int firstNo , int secondNo);
//function signature

Add(int firstNo,int second){
    print("Add result is ${firstNo+second}");
}
Subtract(int firstNo,int second){
    print("Subtract result is ${firstNo-second}");
}
Divide(int firstNo,int second){
    print("Divide result is ${firstNo/second}");
}
Calculator(int a, int b, ManyOperation oper){
    print("Inside calculator");
}
```



```

    oper(a,b);
}
void main(){
    ManyOperation oper = Add;
    oper(10,20);
    oper = Subtract;
    oper(30,20);
    oper = Divide;
    oper(50,5);
}

```

El programa debe producir el siguiente **resultado** :

```

Add result is 30
Subtract result is 10
Divide result is 10.0

```

Nota: el código anterior generará un error si la variable **typedef** intenta apuntar a una función con una firma de función diferente.

Ejemplo

Typedefs también se puede pasar como parámetro a una función. Considere el siguiente ejemplo:

```

typedef ManyOperation(int firstNo , int secondNo);
//function signature
Add(int firstNo,int second){
    print("Add result is ${firstNo+second}");
}
Subtract(int firstNo,int second){
    print("Subtract result is ${firstNo-second}");
}
Divide(int firstNo,int second){
    print("Divide result is ${firstNo/second}");
}
Calculator(int a,int b ,ManyOperation oper){
    print("Inside calculator");
    oper(a,b);
}
main(){
    Calculator(5,5,Add);
    Calculator(5,5,Subtract);
    Calculator(5,5,Divide);
}

```

Producirá el siguiente **resultado** :

```

Inside calculator
Add result is 10
Inside calculator
Subtract result is 0
Inside calculator

```

Divide result is 1.0

Programación de Dart - Bibliotecas

Una biblioteca en un lenguaje de programación representa una colección de rutinas (conjunto de instrucciones de programación). Dart tiene un conjunto de bibliotecas integradas que son útiles para almacenar rutinas que se usan con frecuencia. Una biblioteca Dart se compone de un conjunto de clases, constantes, funciones, typedefs, propiedades y excepciones.

Importar una biblioteca

La importación hace que los componentes de una biblioteca estén disponibles para el código de la persona que llama. La palabra clave `import` se usa para lograr lo mismo. Un archivo dart puede tener múltiples declaraciones de importación.

Los URI de la biblioteca Dart incorporada utilizan el esquema `dart:` para referirse a una biblioteca. Otras bibliotecas pueden usar una ruta del sistema de archivos o el paquete: esquema para especificar su URI. Las bibliotecas proporcionadas por un administrador de paquetes, como la herramienta de publicación, utilizan el *paquete: esquema* .

La sintaxis para importar una biblioteca en Dart se da a continuación:

```
import 'URI'
```

Considere el siguiente fragmento de código:

```
import 'dart:io'  
import 'package:lib1/libfile.dart'
```

Si desea utilizar solo parte de una biblioteca, puede importar selectivamente la biblioteca. La sintaxis para el mismo se da a continuación:

```
import 'package: lib1/lib1.dart' show foo, bar;  
// Import only foo and bar.
```

```
import 'package: mylib/mylib.dart' hide foo;  
// Import all names except foo
```

Algunas bibliotecas de uso común se dan a continuación:

No Señor	Biblioteca y descripción
1	dardo: io Archivo, socket, HTTP y otro soporte de E / S para aplicaciones de servidor. Esta biblioteca no funciona en aplicaciones basadas en navegador. Esta biblioteca se importa por defecto.
2	dardo: núcleo

	Tipos incorporados, colecciones y otras funciones básicas para cada programa Dart. Esta biblioteca se importa automáticamente.
3	dart: matemáticas Constantes y funciones matemáticas, más un generador de números aleatorios.
4 4	dart: convertir Codificadores y decodificadores para convertir entre diferentes representaciones de datos, incluidos JSON y UTF-8.
5 5	dart: typed_data Listas que manejan eficientemente datos de tamaño fijo (por ejemplo, enteros de 8 bytes sin signo).

Ejemplo: importación y uso de una biblioteca

El siguiente ejemplo importa el **dart** incorporado de la biblioteca : **math** . El fragmento llama a la función **sqr** (**)** de la biblioteca **matemática** . Esta función devuelve la raíz cuadrada de un número que se le pasó.

```
import 'dart:math';
void main() {
    print("Square root of 36 is: ${sqrt(36)}");
}
```

Salida

Square root of 36 is: 6.0

Encapsulación en Bibliotecas

Las secuencias de comandos de Dart pueden anteponer identificadores con un guión bajo (**_**) para marcar sus componentes como privados. En pocas palabras, las bibliotecas Dart pueden restringir el acceso a su contenido mediante scripts externos. Esto se denomina **encapsulación** . La sintaxis para el mismo se da a continuación:

Sintaxis

_identifier

Ejemplo

Al principio, defina una biblioteca con una función privada.

```
library loggerlib;
void _log(msg) {
    print("Log method called in loggerlib msg:$msg");
}
```

Luego, importe la biblioteca

```
import 'test.dart' as web;
void main() {
    web._log("hello from webloggerlib");
}
```

El código anterior dará como resultado un error.

```
Unhandled exception:
No top-level method 'web._log' declared.
NoSuchMethodError: method not found: 'web._log'
Receiver: top-level
Arguments: [...]
#0 NoSuchMethodError._throwNew (dart:core-
patch/errors_patch.dart:184)
#1 main
(file:///C:/Users/Administrator/WebstormProjects/untitled/Ass
ertion.dart:6:3)
#2 _startIsolate.<anonymous closure> (dart:isolate-
patch/isolate_patch.dart:261)
#3 _RawReceivePortImpl._handleMessage (dart:isolate-
patch/isolate_patch.dart:148)
```

Crear bibliotecas personalizadas

Dart también le permite usar su propio código como biblioteca. Crear una biblioteca personalizada implica los siguientes pasos:

Paso 1: declarar una biblioteca

Para declarar explícitamente una biblioteca, use la **instrucción de biblioteca**. La sintaxis para declarar una biblioteca es la siguiente:

```
library library_name
// library contents go here
```

Paso 2: asociar una biblioteca

Puede asociar una biblioteca de dos maneras:

- Dentro del mismo directorio

```
import 'library_name'
```

- De un directorio diferente

```
import 'dir/library_name'
```

Ejemplo: Biblioteca personalizada

Primero, definamos una biblioteca personalizada, **calculator.dart** .

```
library calculator_lib;
import 'dart:math';

//import statement after the library statement
int add(int firstNumber,int secondNumber){
    print("inside add method of Calculator Library ") ;
    return firstNumber+secondNumber;
}
int modulus(int firstNumber,int secondNumber){
    print("inside modulus method of Calculator Library ") ;
    return firstNumber%secondNumber;
}
int random(int no){
    return new Random().nextInt(no);
}
```

A continuación, importaremos la biblioteca:

```
import 'calculator.dart';
void main() {
    var num1 = 10;
    var num2 = 20;
    var sum = add(num1,num2);
    var mod = modulus(num1,num2);
    var r = random(10);

    print("$num1 + $num2 = $sum");
    print("$num1 % $num2= $mod");
    print("random no $r");
}
```

El programa debe producir el siguiente **resultado** :

```
inside add method of Calculator Library
inside modulus method of Calculator Library
10 + 20 = 30
10 % 20= 10
random no 0
```

Prefijo de biblioteca

Si importa dos bibliotecas con identificadores en conflicto, puede especificar un prefijo para una o ambas bibliotecas. Use la palabra clave **'as'** para especificar el prefijo. La sintaxis para el mismo se da a continuación:

Sintaxis

```
import 'library_uri' as prefix
```

Ejemplo

Primero, definamos una biblioteca: **loggerlib.dart** .

```
library loggerlib;
void log(msg){
    print("Log method called in loggerlib msg:$msg");
}
```

A continuación, definiremos otra biblioteca: **webloggerlib.dart** .

```
library webloggerlib;
void log(msg){
    print("Log method called in webloggerlib msg:$msg");
}
```

Finalmente, importaremos la biblioteca con un prefijo.

```
import 'loggerlib.dart';
import 'webloggerlib.dart' as web;

// prefix avoids function name clashes
void main(){
    log("hello from loggerlib");
    web.log("hello from webloggerlib");
}
```

Producirá el siguiente **resultado** :

```
Log method called in loggerlib msg:hello from loggerlib
Log method called in webloggerlib msg:hello from webloggerlib
```

Programación de Dart: asíncrono

Una **operación asíncrona** se ejecuta en un hilo, separado del hilo **principal de la** aplicación. Cuando una aplicación llama a un método para realizar una operación de forma asíncrona, la aplicación puede continuar ejecutándose mientras el método asíncronico realiza su tarea.

Ejemplo

Tomemos un ejemplo para entender este concepto. Aquí, el programa acepta la entrada del usuario utilizando la **biblioteca IO** .

```
import 'dart:io';
void main() {
    print("Enter your name :");

    // prompt for user input
    String name = stdin.readLineSync();

    // this is a synchronous method that reads user input
    print("Hello Mr. ${name}");
    print("End of main");
}
```

El **readLineSync ()** es un método síncrono. Esto significa que la ejecución de todas las instrucciones que siguen a la llamada a la función **readLineSync ()** se bloqueará hasta que el método **readLineSync ()** finalice la ejecución.

El **stdin.readLineSync** espera la entrada. Se detiene en seco y no se ejecuta más hasta que recibe la entrada del usuario.

El ejemplo anterior dará como resultado el siguiente **resultado** :

```
Enter your name :  
Tom
```

```
// reads user input  
Hello Mr. Tom  
End of main
```

En informática, decimos que algo es **sincrónico** cuando espera que suceda un evento antes de continuar. Una desventaja de este enfoque es que si una parte del código tarda demasiado en ejecutarse, los bloques posteriores, aunque no estén relacionados, no podrán ejecutarse. Considere un servidor web que debe responder a múltiples solicitudes de un recurso.

Un modelo de ejecución síncrono bloqueará la solicitud de cualquier otro usuario hasta que termine de procesar la solicitud actual. En tal caso, como el de un servidor web, cada solicitud debe ser independiente de las demás. Esto significa que el servidor web no debe esperar a que la solicitud actual termine de ejecutarse antes de responder a la solicitud de otros usuarios.

En pocas palabras, debe aceptar solicitudes de nuevos usuarios antes de completar necesariamente las solicitudes de los usuarios anteriores. Esto se denomina asíncrono. La programación asíncrona básicamente significa que no hay un modelo de programación en espera o sin bloqueo. El paquete **dart:async** facilita la implementación de bloques de programación asíncronos en un script Dart.

Ejemplo

El siguiente ejemplo ilustra mejor el funcionamiento de un bloque asíncrono.

Paso 1 : cree un archivo **contact.txt** como se indica a continuación **y** guárdelo en la carpeta de datos del proyecto actual.

```
1, Tom  
2, John  
3, Tim  
4, Jane
```

Paso 2 : escriba un programa que lea el archivo sin bloquear otras partes de la aplicación.

```
import "dart:async";  
import "dart:io";  
  
void main(){  
  File file = new File(  
    Directory.current.path+"\\data\\contact.txt");
```

```
Future<String> f = file.readAsString();

// returns a Future, this is Async method
f.then((data)=>print(data));

// once file is read , call back method is invoked
print("End of main");
// this get printed first, showing fileReading is non
blocking or async
}
```

El **resultado** de este programa será el siguiente:

```
End of main
1, Tom
2, John
3, Tim
4, Jan
```

El "final de main" se ejecuta primero mientras el script continúa leyendo el archivo. La clase **Future**, parte de **dart: async**, se usa para obtener el resultado de un cálculo después de que se haya completado una tarea asíncrona. Este valor **futuro** se usa para hacer algo después de que finaliza el cálculo.

Una vez que se completa la operación de lectura, el control de ejecución se transfiere dentro de "**then ()**". Esto se debe a que la operación de lectura puede llevar más tiempo y, por lo tanto, no quiere bloquear otra parte del programa.

Dart Future

La comunidad Dart define un **futuro** como "un medio para obtener un valor en algún momento en el futuro". En pocas palabras, los **objetos futuros** son un mecanismo para representar los valores devueltos por una expresión cuya ejecución se completará en un momento posterior. Varias de las clases integradas de Dart devuelven un **futuro** cuando se llama a un método asíncrono.

Dart es un lenguaje de programación de subproceso único. Si algún código bloquea el hilo de ejecución (por ejemplo, al esperar una operación que consume mucho tiempo o el bloqueo de E / S), el programa se congela efectivamente.

Las operaciones asíncronas permiten que su programa se ejecute sin bloquearse. Dart utiliza **objetos futuros** para representar operaciones asíncronas.

Programación de Dart - concurrencia

La **concurrencia** es la ejecución de varias secuencias de instrucciones al mismo tiempo. Implica realizar más de una tarea simultáneamente.

Dart usa **Aislamientos** como una herramienta para hacer trabajos en paralelo. El paquete **dart: isolate** es la solución de Dart para tomar código Dart de un solo subproceso y permitir que la aplicación haga un mayor uso del hardware disponible.

Los aislamientos, como su nombre indica, son unidades aisladas de código en ejecución. La única forma de enviar datos entre ellos es pasando mensajes, como la forma en que pasa mensajes entre el cliente y el servidor. Un **aislamiento** ayuda al programa a aprovechar los microprocesadores multinúcleo listos para usar.

Ejemplo

Tomemos un ejemplo para comprender mejor este concepto.

```
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');
}
```

Aquí, el **desove** método de la **Aislar** clase facilita la ejecución de una función, **foo**, en paralelo con el resto de nuestro código. La función de **generación** toma dos parámetros:

- la función que se generará, y
- un objeto que se pasará a la función generada.

En caso de que no haya ningún objeto para pasar a la función generada, se le puede pasar un valor NULL.

Las dos funciones (**foo y main**) pueden no ejecutarse necesariamente en el mismo orden cada vez. No hay garantía de cuándo se ejecutará **foo** y cuándo se ejecutará **main ()**. El resultado será diferente cada vez que ejecute.

Salida 1

```
execution from main1
execution from main2
execution from main3
execution from foo ... the message is :Hello!!
```

Salida 2

```
execution from main1
execution from main2
execution from main3
execution from foo ... the message is :Welcome!!
execution from foo ... the message is :Hello!!
execution from foo ... the message is :Greetings!!
```

A partir de los resultados, podemos concluir que el código Dart puede generar un nuevo **aislamiento** del código en ejecución, como la forma en que el código Java o C # puede iniciar un nuevo hilo.

Los aislamientos difieren de los hilos en que un **aislante** tiene su propia memoria. No hay forma de compartir una variable entre los **aislamientos** ; la única forma de comunicarse entre los **aislamientos** es mediante el envío de mensajes.

Nota : La salida anterior será diferente para diferentes configuraciones de hardware y sistema operativo.

Aislar v / s futuro

Hacer un trabajo computacional complejo de forma asincrónica es importante para garantizar la capacidad de respuesta de las aplicaciones. **Dart Future** es un mecanismo para recuperar el valor de una tarea asincrónica después de que se haya completado, mientras que **Dart Isolates** es una herramienta para abstraer el paralelismo e implementarlo sobre una base práctica de alto nivel.

Programación de Dart: pruebas unitarias

Las pruebas unitarias implican probar cada unidad individual de una aplicación. Ayuda al desarrollador a probar pequeñas funcionalidades sin ejecutar toda la aplicación compleja.

La **biblioteca externa** Dart llamada "prueba" proporciona una forma estándar de escribir y ejecutar pruebas unitarias.

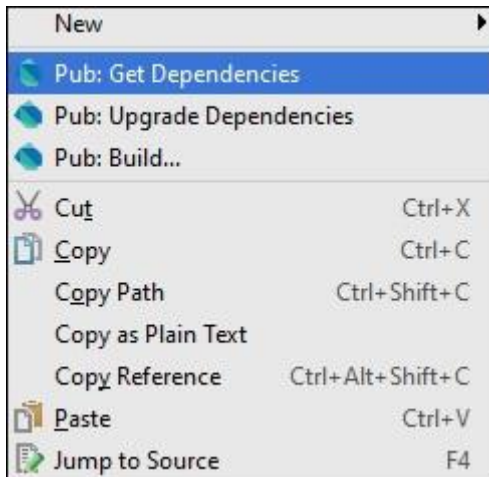
La prueba de la unidad Dart implica los siguientes pasos:

Paso 1: Instalar el paquete "prueba"

Para instalar paquetes de terceros en el proyecto actual, necesitará el archivo **pubspec.yaml** . Para instalar **paquetes de prueba** , primero realice la siguiente entrada en el archivo **pubspec.yaml** :

```
dependencies:
test:
```

Después de hacer la entrada, haga clic derecho en el archivo **pubspec.yaml** y obtenga dependencias. Instalará el paquete "**prueba**". A continuación se muestra una captura de pantalla para el mismo en el Editor de **WebStorm** .



Los paquetes también se pueden instalar desde la **línea de comandos** . Escriba lo siguiente en la terminal:

```
pub get
```

Paso 2: Importar el paquete "prueba"

```
import "package:test/test.dart";
```

Paso 3 Pruebas de escritura

Las pruebas se especifican usando la función **test de nivel superior ()** , mientras que **las afirmaciones de prueba** se hacen usando la función **expect ()** . Para usar estos métodos, deben instalarse como una dependencia de **pub** .

Sintaxis

```
test("Description of the test ", () {  
    expect(actualValue , matchingValue)  
});
```

La función **group ()** se puede usar para agrupar pruebas. La descripción de cada grupo se agrega al comienzo de las descripciones de sus pruebas.

Sintaxis

```
group("some_Group_Name", () {  
    test("test_name_1", () {  
        expect(actual, equals(exptected));  
    });  
    test("test_name_2", () {  
        expect(actual, equals(expected));  
    });  
});
```

Ejemplo 1: una prueba de aprobación

El siguiente ejemplo define un método **Add ()** . Este método toma dos valores enteros y devuelve un entero que representa la **suma** . Para probar este método **add ()** :

Paso 1 : importe el paquete de **prueba** como se indica a continuación.

Paso 2 : defina la prueba con la función **test ()** . Aquí, la función **test ()** usa la función **expect ()** para imponer una aserción.

```
import 'package:test/test.dart';
// Import the test package

int Add(int x,int y)
// Function to be tested {
    return x+y;
}
void main() {
    // Define the test
    test("test to check add method", () {
        // Arrange
        var expected = 30;

        // Act
        var actual = Add(10,20);

        // Asset
        expect(actual,expected);
    });
}
```

Debería producir el siguiente **resultado** :

```
00:00 +0: test to check add method
00:00 +1: All tests passed!
```

Ejemplo 2: una prueba fallida

El método **sustraer ()** definido a continuación tiene un error lógico. La siguiente **prueba** verifica lo mismo.

```
import 'package:test/test.dart';
int Add(int x,int y){
    return x+y;
}
int Sub(int x,int y){
    return x-y-1;
}
void main(){
    test('test to check sub', () {
        var expected = 10;
        // Arrange

        var actual = Sub(30,20);
        // Act
```

```

        expect(actual, expected);
        // Assert
    });
    test("test to check add method", () {
        var expected = 30;
        // Arrange

        var actual = Add(10, 20);
        // Act

        expect(actual, expected);
        // Asset
    });
}

```

Salida: el caso de prueba para la función **add ()** pasa pero la prueba para **restar ()** falla como se muestra a continuación.

```

00:00 +0: test to check sub
00:00 +0 -1: test to check sub
Expected: <10>
Actual: <9>
package:test expect
bin\Test123.dart 18:5 main.<fn>

00:00 +0 -1: test to check add method
00:00 +1 -1: Some tests failed.
Unhandled exception:
Dummy exception to set exit code.
#0 _rootHandleUncaughtError.<anonymous closure>
(dart:async/zone.dart:938)
#1 _microtaskLoop (dart:async/schedule_microtask.dart:41)
#2 _startMicrotaskLoop
(dart:async/schedule_microtask.dart:50)
#3 _Timer._runTimers (dart:isolate-
patch/timer_impl.dart:394)
#4 _Timer._handleMessage (dart:isolate-
patch/timer_impl.dart:414)
#5 _RawReceivePortImpl._handleMessage (dart:isolate-
patch/isolate_patch.dart:148)

```

Agrupación de casos de prueba

Puede agrupar los **casos de prueba** para que agregue más significado a su código de prueba. Si tiene muchos **casos de prueba**, esto ayuda a escribir un código mucho más limpio.

En el código dado, estamos escribiendo un caso de prueba para la función **split ()** y la función **trim** . Por lo tanto, agrupamos lógicamente estos casos de prueba y lo llamamos **String** .

Ejemplo

```
import "package:test/test.dart";
void main() {
  group("String", () {
    test("test on split() method of string class", () {
      var string = "foo,bar,baz";
      expect(string.split(","), equals(["foo", "bar",
"baz"]));
    });
    test("test on trim() method of string class", () {
      var string = "  foo ";
      expect(string.trim(), equals("foo"));
    });
  });
}
```

Salida : la salida agregará el nombre del grupo para cada caso de prueba como se indica a continuación:

```
00:00 +0: String test on split() method of string class
00:00 +1: String test on trim() method of string class
00:00 +2: All tests passed
```

Programación de Dart - HTML DOM

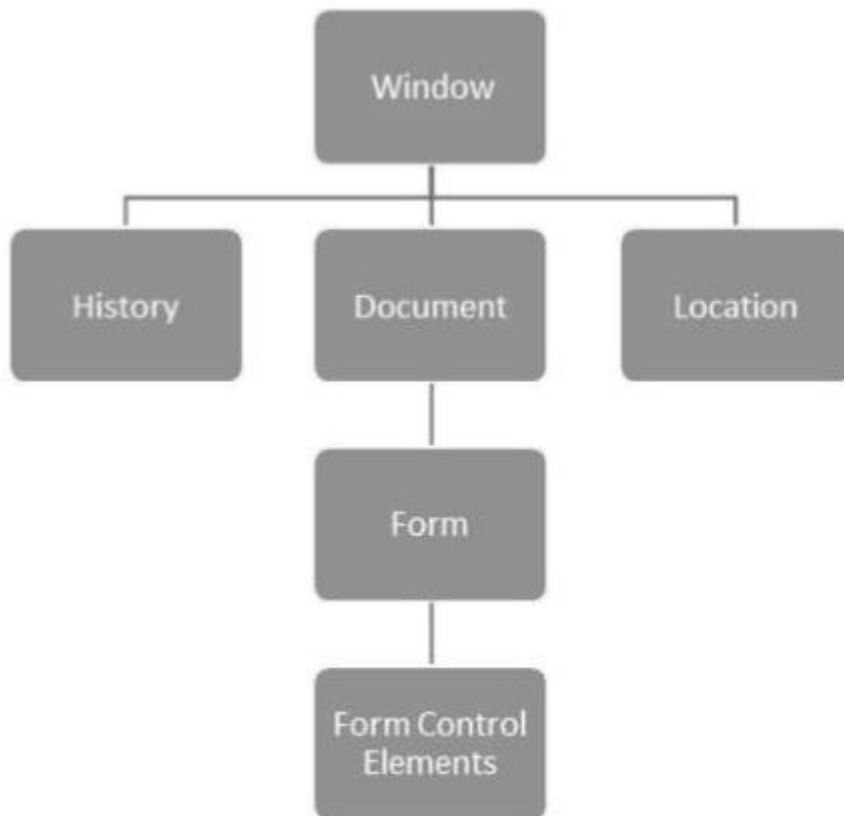
Cada página web reside dentro de una ventana del navegador que puede considerarse como un objeto.

Un **objeto Document** representa el documento HTML que se muestra en esa ventana. El objeto Documento tiene varias propiedades que se refieren a otros objetos que permiten el acceso y la modificación del contenido del documento.

La forma en que se accede y modifica el contenido de un documento se denomina **Modelo de objeto de documento** , o **DOM** . Los objetos están organizados en una jerarquía. Esta estructura jerárquica se aplica a la organización de objetos en un documento web.

- **Ventana** : parte superior de la jerarquía. Es el elemento más externo de la jerarquía de objetos.
- **Documento** : cada documento HTML que se carga en una ventana se convierte en un objeto de documento. El documento contiene el contenido de la página.
- **Elementos** : representan el contenido de una página web. Los ejemplos incluyen los cuadros de texto, el título de la página, etc.
- **Nodos** : a menudo son elementos, pero también pueden ser atributos, texto, comentarios y otros tipos de DOM.

Aquí hay una jerarquía simple de algunos objetos DOM importantes:



Dart proporciona la biblioteca **dart: html** para manipular objetos y elementos en el DOM. Las aplicaciones basadas en consola no pueden usar la biblioteca **dart: html** . Para usar la biblioteca HTML en las aplicaciones web, importe **dart: html** -

```
import 'dart:html';
```

Continuando, discutiremos algunas **operaciones DOM** en la siguiente sección.

Encontrar elementos DOM

La biblioteca **dart: html** proporciona la función **querySelector** para buscar elementos en el DOM.

```
Element querySelector(String selectors);
```

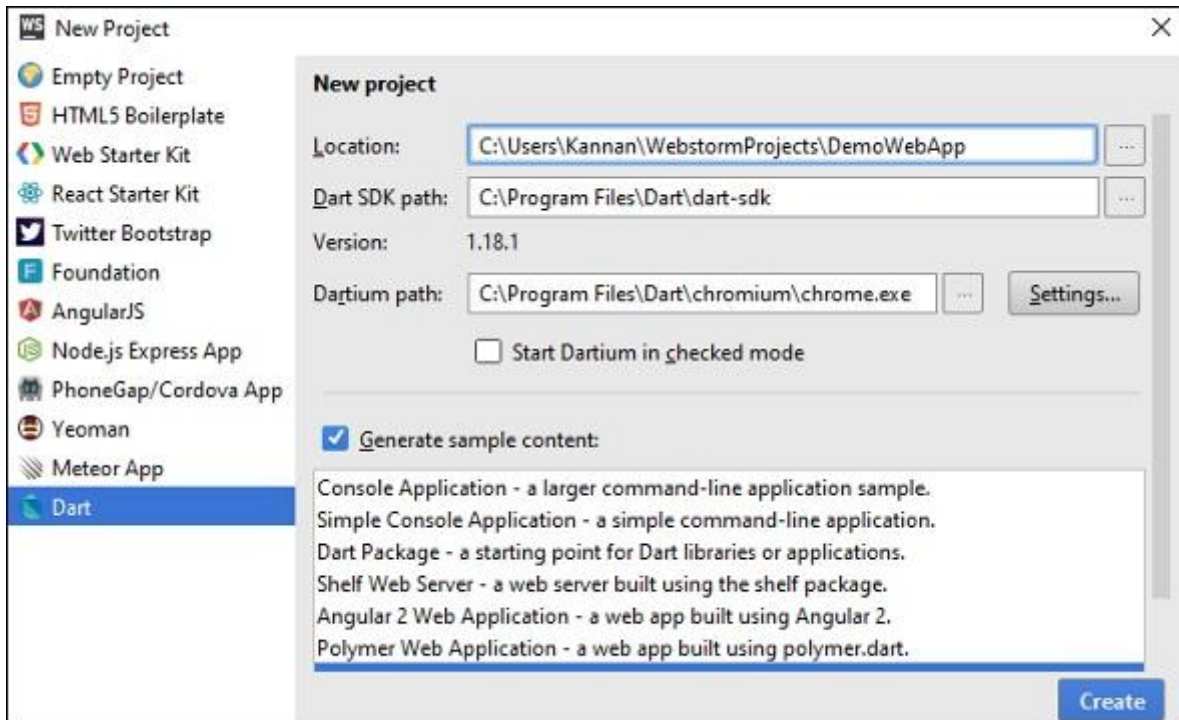
La función **querySelector ()** devuelve el primer elemento que coincide con el grupo especificado de selectores. "**los selectores**" deben ser cadenas usando la sintaxis del selector CSS como se indica a continuación

```
var element1 = document.querySelector('.className');  
var element2 = document.querySelector('#id');
```

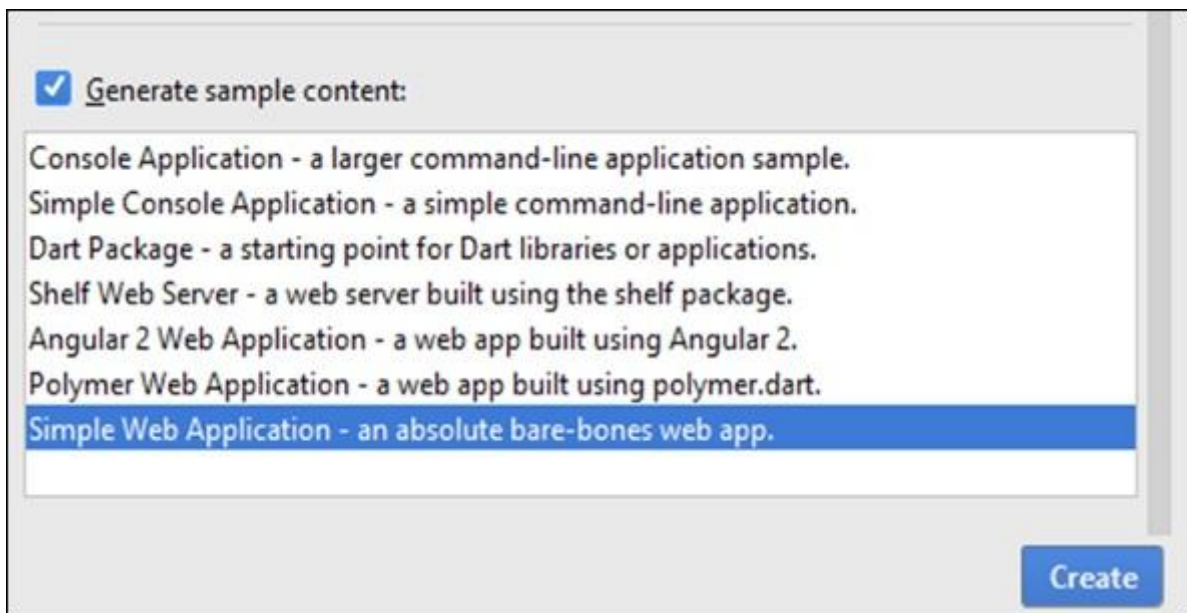
Ejemplo: manipulación de DOM

Siga los pasos que se detallan a continuación, en el IDE de Webstorm:

Paso 1 - Archivo NewProject → En la ubicación, proporcione el nombre del proyecto como **DemoWebApp** .



Paso 1 : en la sección "Generar contenido de muestra", seleccione **SimpleWebApplication** .



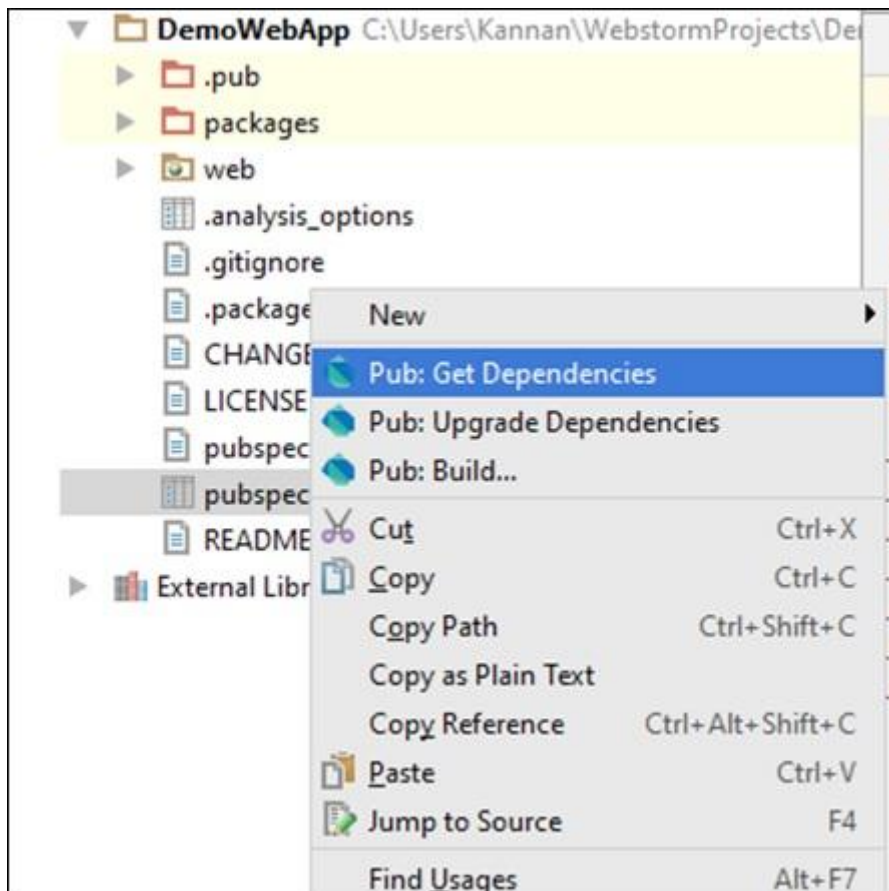
Crearé un proyecto de muestra, **DemoWebApp** . Hay un archivo **pubspec.yaml** que contiene las dependencias que deben descargarse.

```
name: 'DemoWebApp'  
version: 0.0.1  
description: An absolute bare-bones web app.  
  
#author: Your Name <email@example.com>
```



```
#homepage: https://www.example.com
environment:
  sdk: '>=1.0.0 <2.0.0'
dependencies:
  browser: '>=0.10.0 <0.11.0'    dart_to_js_script_rewriter:
'^1.0.1'
transformers:
- dart_to_js_script_rewriter
```

Si está conectado a la Web, estos se descargarán automáticamente; de lo contrario, puede hacer clic con el botón derecho en **pubspec.yaml** y obtener dependencias.



En la carpeta web, encontrará tres archivos: **Index.html**, **main.dart** y **style.css**

Index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "utf-8">
    <meta http-equiv = "X-UA-Compatible" content = "IE =
edge">
    <meta name = "viewport" content = "width = device-
width, initial-scale = 1.0">
```

```

<meta name = "scaffolded-by" content =
"https://github.com/google/stagehand">
<title>DemoWebApp</title>
<link rel = "stylesheet" href = "styles.css">
<script defer src = "main.dart" type =
"application/dart"></script>
<script defer src =
"packages/browser/dart.js"></script>
</head>

<body>
<h1>
<div id = "output"></div>
</h1>
</body>
</html>

```

Main.dart

```

import 'dart:html';
void main() {
  querySelector('#output').text = 'Your Dart web dom app is
running!!!.';
}

```

Ejecute el archivo **index.html** ; verá la siguiente salida en su pantalla.



Manejo de eventos

La biblioteca **dart:html** proporciona el evento **onClick** para DOM Elements. La sintaxis muestra cómo un elemento podría manejar una secuencia de eventos de clic.

```
querySelector('#Id').onClick.listen(eventHanlderFunction);
```

La función **querySelector ()** devuelve el elemento del DOM dado y **onClick.listen ()** tomará un método **eventHandler** que se invocará cuando se genere un evento click. La sintaxis de **eventHandler** se da a continuación:

```
void eventHanlderFunction (MouseEvent event){ }
```

Tomemos ahora un ejemplo para comprender el concepto de Manejo de eventos en Dart.

TestEvent.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "utf-8">
    <meta http-equiv = "X-UA-Compatible" content = "IE =
edge">
    <meta name = "viewport" content = "width = device-
width, initial-scale = 1.0">
    <meta name = "scaffolded-by" content
="https://github.com/google/stagehand">
    <title>DemoWebApp</title>
    <link rel = "stylesheet" href = "styles.css">
    <script defer src = "TestEvent.dart"
type="application/dart"></script>
    <script defer src =
"packages/browser/dart.js"></script>
  </head>

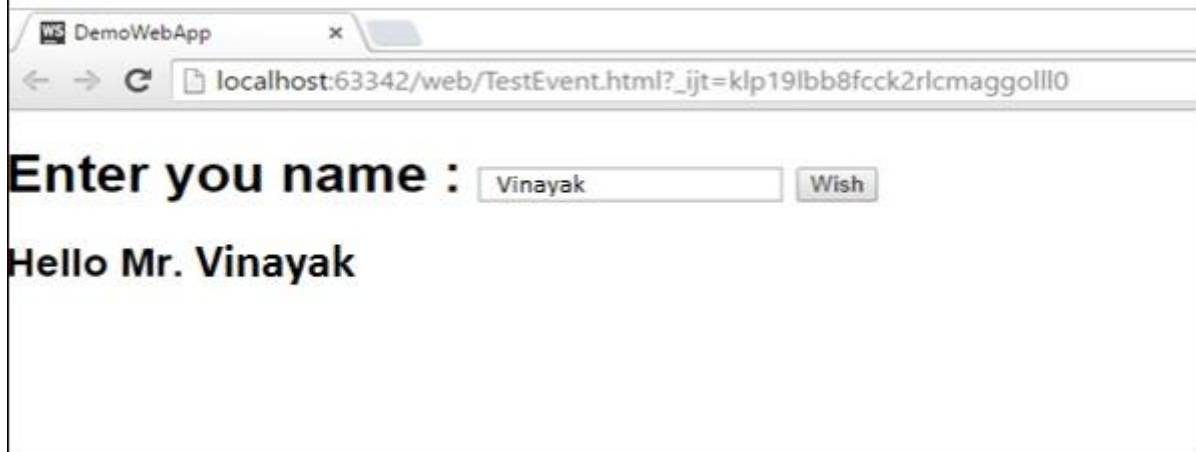
  <body>
    <div id = "output"></div>
    <h1>
      <div>
        Enter you name : <input type = "text" id =
"txtName">
          <input type = "button" id = "btnWish"
value="Wish">
        </div>
      </h1>
      <h2 id = "display"></h2>
    </body>
</html>
```

TestEvent.dart

```
import 'dart:html';
void main() {
  querySelector('#btnWish').onClick.listen(wishHandler);
}
void wishHandler(MouseEvent event){
  String name = (querySelector('#txtName') as
InputElement).value;
  querySelector('#display').text = 'Hello Mr.'+ name;
}
```

Salida

Output



A screenshot of a web browser window. The title bar shows 'DemoWebApp' and a close button. The address bar shows 'localhost:63342/web/TestEvent.html?_ijt=klp19lbb8fcck2rlcmaggolll0'. The main content area displays the text 'Enter you name :' followed by a text input field containing 'Vinayak' and a 'Wish' button. Below this, the text 'Hello Mr. Vinayak' is displayed.

Enter you name :

Hello Mr. Vinayak