



# Argentina programa 2 suscriptor comprimido y desatollo web del entorno

Historia Económica (Instituto Universitario Escuela Argentina de Negocios)



## Capítulo 2: Programación Imperativa

¿Ya estás para salir del tablero? ¡Acompañanos a aprender más sobre programación imperativa y estructuras de datos de la mano del lenguaje JavaScript!

### Lecciones

#### 1. Funciones y tipos de datos

1. Introducción a JavaScript
2. Funciones, declaración
3. Funciones, uso
4. Probando funciones
5. Haciendo cuentas
6. Poniendo topes
7. Libros de la buena memoria
8. Booleanos
9. Palabras, sólo palabras
10. Operando strings
11. ¡GRITAR!
12. ¿Y qué tal si...?
13. ¿De qué signo sos?
14. El retorno del booleano
15. Los premios
16. Tipos de datos
17. Datos de todo tipo

#### 2. Práctica Funciones y Tipos de Datos

1. Comprando Hardware
2. ¿Me conviene?
3. Triangulos
4. Cuadrados
5. ¿Está afnado?
6. ¿Está cerca?
7. Cartelitos
8. Más Cartelitos
9. Cartelitos óptimos
10. Cara o ceca
11. ¡Envido!
12. ¡Quiero retruco!
13. ¡Quiero vale cuatro!

#### 3. Variables y procedimientos

1. ¿Y el tablero?
2. Impresión por pantalla
3. Martin Fierro
4. ¿Y los procedimientos?
5. ¿Y el program?
6. Coerciones
7. El círculo de la vida
8. Plenso que así es más fácil
9. Esto no tiene valor
10. Variables globales
11. Volviéndonos ricos
12. ¿Y esto cuánto vale?

#### 4. Lógica booleana

1. ¡Que el último apague la luz!
2. Negar no cuesta nada
3. Los peripatéticos
4. La verdad detrás de la conjunción
5. ¡Juguemos al T.E.G.!
6. Y ahora... ¿quién podrá ayudarnos?
7. ¡Buen día!
8. La verdad es que no hay una verdad
9. ¡Hola! Mi nombre es Xor
10. Precedencia
11. Un ejercicio sin precedentes
12. ¿Puedo subir?

#### 5. Listas

1. Series favoritas
2. Y esto, es una lista
3. Juegos de azar
4. Listas vacías
5. ¿Cuántos elementos tenés?
6. Agregando sabor
7. Trasladar
8. ¿Y dónde está?
9. Contiene
10. Enésimo elemento
11. Más premios
12. No te olvides de saludar

#### 6. Registros

1. Los primeros registros
2. Tu propio monumento
3. Accediendo al campo
4. Temperatura de planeta
5. Moviendo archivos
6. Registros de dos milenios
7. Postres complejos
8. Listas de registros
9. 60 dulces minutos
10. Hay un registro en mi registro
11. ¡Azúcar!



## 7. Práctica de listas y registros

1. Las ganancias semestrales
2. ¿Y el resto de las ganancias?
3. Todas las ganancias, la ganancia
4. Nos visita un viejo amigo
5. Cuentas claras
6. La ganancia promedio
7. Quién gana, quién pierde
8. Soy el mapa, soy el mapa
9. A filtrar, a filtrar cada cosa en su lugar
10. Un promedio más positivo
11. Esto es lo máximo
12. Como mínimo
13. Los mejores meses del año
14. Publicaciones muy especiales

## Apéndice



## Introducción a JavaScript

¿Ya te cansaste de jugar con bolitas de colores? ● ● ● Tenemos una buena noticia. En este capítulo vamos a aprender **programación imperativa** de la mano de uno de los lenguajes de programación más utilizados de la industria del software: JavaScript.

Introducción a JavaScript





## Funciones, declaración

Gobstones y JavaScript tienen mucho en común. Por ejemplo, en ambos lenguajes podemos declarar **funciones** y usarlas muchas veces.

Sin embargo, como siempre que aprendas un lenguaje nuevo, te vas a topar con un pequeño detalle: **tiene una sintaxis diferente** 😱. La buena noticia es que el cambio no será tan terrible como suena, así que veamos nuestra primera función JavaScript:

```
function doble(numero) {  
    return 2 * numero;  
}
```

Diferente, pero no tanto. Si la comparás con su equivalente Gobstones...

```
function doble(numero) {  
    return (2* numero)  
}
```

...notarás que los paréntesis en el `return` no son necesarios, y que la última línea la terminamos con `;`.

Veamos si se va entendiendo: escribí ahora una función JavaScript `mitad`, que tome un número y devuelva su mitad. Tené en cuenta que el operador de división en JavaScript es `/`.

💡 ¡Dame una pista!

💡 Solución 💡 Consola

```
1 function mitad(numero) {  
2     return numero / 2;  
3 }
```

💡 Enviar

💡 ¡Muy bien! Tu solución pasó todas las pruebas

Perfecto, ¿viste que no era tan terrible? 😊

Si no le pusiste `;` al final de la sentencia habrás visto que funciona igual. De todas formas ponelo, ya que de esa manera evitamos posibles problemas.

Siempre que aprendamos un lenguaje nuevo vamos a tener que aprender una nueva sintaxis. Sin embargo y por fortuna, si tenés los conceptos claros, no es nada del otro mundo 😊.

Aprendamos ahora a usar estas funciones.



## Funciones, uso

¿Y esto con qué se come? Digo, ehm.... ¿cómo se usan estas funciones? ¿Cómo hago para pasárselas y obtener resultados?

Basta con poner el nombre de la función y, entre paréntesis, sus argumentos. ¡Es igual que en Gobstones!

```
doble(3)
```

Y además podemos usarlas dentro de otras funciones. Por ejemplo:

```
function doble(numero) {
    return 2 * numero;
}

function siguienteDelDoble(numero) {
    return doble(numero) + 1;
}
```

O incluso mejor:

```
function doble(numero) {
    return 2 * numero;
}

function siguiente(numero) {
    return numero + 1;
}

function siguienteDelDoble(numero) {
    return siguiente(doble(numero));
}
```

Veamos si se entiende; escribí las siguientes funciones:

- anterior : toma un número y devuelve ese número menos uno
- triple : devuelve el triple de un número
- anteriorDelTriple , que combina las dos funciones anteriores: multiplica a un número por 3 y le resta 1

[Solución](#) [Consola](#)

```
1 function anterior(numero) {
2     return numero - 1;
3 }
4
5 function triple(numero) {
6     return numero * 3;
7 }
8
9 function anteriorDelTriple(numero) {
10    return anterior(triple(numero));
11 }
```

[Enviar](#)

**¡Muy bien! Tu solución pasó todas las pruebas**

Quizás ahora estés pensando: si no tengo un tablero, ¿cómo sé si mi función hace lo que debe? Acompañanos...

---

Esta guía fue desarrollada por Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





## Probando funciones

Quizás ya lo notaste pero, junto al editor, ahora aparece una solapa nueva: la *consola*.

La consola es una herramienta muy útil para hacer pruebas rápidas sobre lo que estás haciendo: te permite, por ejemplo, probar *expresiones*, funciones que vengan con JavaScript, o incluso funciones que vos definas en el editor.

La podés reconocer fácilmente porque arranca con el chiribolito ☒, que se llama *prompt* (<https://es.wikipedia.org/wiki/Prompt>).

Para entender mejor cómo funciona, en qué puede ayudarnos y algunos consejos sobre su uso, te recomendamos mirar este video:

La Consola



Veamos si se entiende, probá en la consola las siguientes expresiones:

- `4 + 5`
- `Math.round(4.5)`
- `funcionMisteriosa(1, 2, 3)` (ya la declaramos por vos y la podés usar)

```
☒ 4 + 5
=> 9
☒ Math.round(4.5)
=> 5
☒ funcionMisteriosa(1, 2, 3)
=> 9
☒ Math.round(4.5)
=> 5
```



## Haciendo cuentas

Además de los operadores matemáticos `+`, `-`, `/` y `*`, existen muchas otras funciones matemáticas comunes, algunas de las cuales ya vienen con JavaScript y están listas para ser usadas.

Sin embargo, la sintaxis de estas funciones matemáticas es *apenitas* diferente de lo que veníamos haciendo hasta ahora: hay que prefijarlas con `Math.`. Por ejemplo, la función que nos sirve para redondear un número es `Math.round`:

```
function cuantoSaleAproximadamente(precio, impuestos) {  
    return Math.round(precio * impuestos);  
}
```

Probá en la consola las siguientes expresiones:

- `Math.round(4.4)`
- `Math.round(4.6)`
- `Math.max(4, 7)`
- `Math.min(4, 7)`

```
=> 4  
✉ Math.round(4.6)  
=> 5  
✉ Math.max(4, 7)  
=> 7  
✉ Math.min(4, 7)  
=> 4  
✉
```



## Poniendo topes

Hagamos un alto en nuestro camino y miremos las funciones `Math.max` y `Math.min`, que nos pueden ahorrar más trabajo del que parece.

Necesitamos una función que diga cuánta plata queda en tu cuenta (que tiene un cierto `saldo`) si extráes un cierto `monto`:

```
// el saldo es $100, el monto a extraer, $30
☒ extraer(100, 30)
70 //quedan $70 ($100 - $30 = $70)
```

Pero como no queremos quedarnos en negativo, si el monto a extraer es mayor al saldo, nuestro saldo debe quedar en cero.

```
☒ extraer(100, 120)
0 //Ups, quisimos sacar más plata de la que teníamos.
//Nos quedamos con $0
```

Como ves, esto es *casi* una resta entre `saldo` y `monto`, con la salvedad de que estamos poniendo un *tope inferior*: no puede dar menos de cero 😊.

En otras palabras (¡preparate!, esto te puede volar la cabeza 🤯) extraer **devuelve el máximo entre la resta `saldo - monto` y 0**.

☒ ¿Te animás a completar la solución que está en el editor?

☒ ¡Dame una pista!

☒ Solución ☒ Consola

```
1 function extraer(saldo, monto) {
2   return Math.max(saldo - monto, 0);
3 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Bien hecho! Ahora andá y probalo en la consola 😊

Como ves, la función `Math.max` nos sirvió para implementar un *tope inferior*. De forma análoga, la función `Math.min` nos puede servir para implementar un *tope superior*.

Ah, y si estás pensando “en Gobstones podría haber hecho esto con un `if`”, ¡tenés razón!. Pero esta solución es mucho más breve y simple 😊.



## Libros de la buena memoria

¡Vemos más operadores! Dani ama el primer día de cada mes 📆, y por eso escribió esta función...

```
function esDiaFavorito(diaDelMes) {  
    return diaDelMes === 1 ;  
}
```

...y la usa así (*y la dejó en la biblioteca para que la pruebes*):

```
✉ esDiaFavorito(13)  
false  
✉ esDiaFavorito(1)  
true
```

Como ves, en JavaScript contamos con operadores como `==`, `>=`, `>`, `<`, `<=` que nos dicen si dos valores son iguales, mayores-o-iguales, mayores, etc. Los vamos a usar bastante 😊.

¡Ahora te toca a vos! Dani también dice que a alguien `leGustaLeer`, cuando la cantidad de libros que recuerda haber leído es mayor a 20. Por ejemplo:

```
✉ leGustaLeer(15)  
false  
✉ leGustaLeer(45)  
true
```

Desarrollá y probá en la consola la función `leGustaLeer`.

✉ Solución ✉ Consola

```
1 function leGustaLeer(libros) {  
2     return libros > 20  
3 }
```

✉ Enviar

✉ ¡Muy bien! Tu solución pasó todas las pruebas

¡Bien hecho!

Capaz pasó desapercibido, pero `leGustaLeer` devuelve `true` o `false`, es decir, es una función que devuelve booleanos. Eso significa que en JavaScript, no sólo hay números sino que también.... hay booleanos 😊.



## Booleanos

Ahora miremos a los booleanos con un poco más de detalle:

- Se pueden negar, mediante el operador `!`: `!hayComida`
- Se puede hacer la conjunción lógica entre dos booleanos (*and*, también conocido en español como *y lógico*), mediante el operador `&&` : `hayComida && hayBebida`
- Se puede hacer la disyunción lógica entre dos booleanos (*or*, también conocido en español como *o lógico*), mediante el operador `||` : `unaExpresion || otraExpresion`

Veamos si se entiende; escribí las siguientes funciones:

- `estaEntre` , que tome tres números y diga si el primero es mayor al segundo y menor al tercero.
- `estaFueraDeRango` : que tome tres números y diga si el primero es menor al segundo o mayor al tercero

Ejemplos:

```
☒ estaEntre(3, 1, 10)
true
☒ estaEntre(90, 1, 10)
false
☒ estaEntre(10, 1, 10)
false
☒ estaFueraDeRango(17, 1, 10)
true
```

Solución  Consola

```
1 function estaEntre(x, y, z) {
2   return (x > y) && (x < z);
3 }
4
5 function estaFueraDeRango(x, y, z) {
6   return (x < y) || (x > z);
7 }
8
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Bien hecho!

Ya fueron suficientes booleanos y cuentas por ahora, ¿no? Exploraremos algo más interesante: los `strings`.

Siguiente Ejercicio: Palabras, sólo palabras  (/argentina-programa/exercises/2768-programacion-imperativa-funciones-y-tipos-de-datos-pal)



## Palabras, sólo palabras

Muchas veces queremos escribir programas que trabajen con texto ☑: queremos saber cuántas palabras hay en un libro, o convertir minúsculas a mayúsculas, o saber en qué parte de un texto está otro.

Para este tipo de problemas tenemos los *strings*, también llamados *cadenas de caracteres*:

- "Ahora la bebé tiene que dormir en la cuna"
- 'El hierro nos ayuda a jugar'
- "¡Hola Miguel!"

Como se observa, todos los strings están encerrados entre comillas simples o dobles. ¡Da igual usar unas u otras! Pero sé consistente: por ejemplo, si abriste comilla doble, tenés que cerrar comilla doble. Además, un string puede estar formado por (casi) cualquier carácter: letras, números, símbolos, espacios, etc.

¿Y qué podemos hacer con los strings? Por ejemplo, compararlos, como a cualquier otro valor:

```
☒ "hola" === "Hola"  
false  
  
☒ "todo el mundo" === "todo el mundo"  
true
```

Veamos si queda claro: escribí la función `esFinDeSemana` que tome un string que represente el nombre de un día de la semana, y nos diga si es "sábado" o "domingo".

```
☒ esFinDeSemana("sábado")  
true  
☒ esFinDeSemana("martes")  
false
```

☒ ¡Dame una pista!

Para saber si un día es fin de semana, *ese día tiene que ser "sábado" o ese día tiene que ser "domingo"*. Recordá que el "o lógico" opera booleanos, no strings. ☺

☒ Solución ☒ Consola

```
1 function esFinDeSemana(dia) {  
2   return (dia === "sábado") || (dia === "domingo")  
3 }
```

☒ Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

---

Esta guía fue desarrollada por Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





## Operando strings

¿Y qué podemos hacer con los strings, además de compararlos? ¡Varias cosas! Por ejemplo, podemos preguntarles cuál es su cantidad de letras:

```
☒ longitud("biblioteca")
10
☒ longitud("babel")
5
```

O también podemos *concatenarlos*, es decir, obtener **uno nuevo** que junta dos strings:

```
☒ "aa" + "bb"
"aab"
☒ "sus anaqueles " + "registran todas las combinaciones"
"sus anaqueles registran todas las combinaciones"
```

O podemos preguntarles si uno comienza con otro:

```
☒ comienzaCon("una página", "una")
true
☒ comienzaCon("la biblioteca", "todos los fuegos")
false
```

Veamos si queda claro: escribí una función `longitudNombreCompleto`, que tome un nombre y un apellido, y devuelva su longitud total, contando un espacio extra para separar a ambos:

```
☒ longitudNombreCompleto("Cosme", "Fulanito")
14
```

[☒ Solución](#) [☒ Biblioteca](#) [☒ Consola](#)

```
1 function longitudNombreCompleto(nombre, apellido) {
2   return longitud(nombre) + 1 + longitud(apellido);
3 }
4
```

[☒ Enviar](#)

[☒ ¡Muy bien! Tu solución pasó todas las pruebas](#)

Esta guía fue desarrollada por Franco Bulgarelli bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





## ¡GRITAR!

Una conocida banda, para agregar gritos varios a su canción, nos pidió que desarrollemos una función `gritar`, que tome un string y lo devuelva en mayúsculas y entre signos de exclamación.

Por ejemplo:

```
☒ gritar("miguel")
"¡MIGUEL!"

☒ gritar("benito")
"¡BENITO!"
```

Escribí la función `gritar`. Te dejamos para que uses la función `convertirEnMayuscula`, que, ehm... bueno... básicamente convierte en mayúsculas un string 😊.

☒ ¡Dame una pista!

Tené en cuenta que los signos de admiración "¡" y "!" (al igual que los espacios y otros signos de puntuación) son strings y que los strings se pueden concatenar usando el operador `+`.

Por ejemplo:

```
☒ "todo" + "terreno"
"todoterreno"

☒ "¿" + "Aló" + "?"
"¿Aló?"
```

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function gritar(palabra) {
2   return "¡" + convertirEnMayuscula(palabra) + "!";
3 }
4
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## ¿Y qué tal si...?

Ninguna introducción al lenguaje JavaScript estaría completa sin mostrar al menos una estructura de control que ya conocemos: la alternativa condicional. Veamos un ejemplo:

```
//Equivalente a Math.abs
function valorAbsoluto(unNumero) {
  if (unNumero >= 0) {
    return unNumero;
  } else {
    return -unNumero;
  }
}
```

Veamos si se entiende: escribí una función `maximo`, que funcione como `Math.max` (¡no vale usarla!) y devuelva el máximo entre dos números. Por ejemplo, el máximo entre 4 y 5 es 5, y el máximo entre 10 y 4, es 10.

Solución  Biblioteca  Consola

```
1 function maximo(numero1, numero2) {
2   if (numero1 > numero2) {
3     return numero1;
4   } else {
5     return numero2;
6   }
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## ¿De qué signo sos?

Necesitamos una función `signo`, que dado un número nos devuelva:

- 1 si el número es positivo
- 0 si el número es cero
- -1 si el número es negativo

Escribí la función `signo`. Quizás necesites más de un `if`.

☒ ¡Dame una pista!

Un número es positivo cuando es **mayor a 0** y negativo cuando es **menor a 0**.

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function signo(numero) {  
2   if (numero > 0) {  
3     return 1  
4   } if (numero === 0) {  
5     return 0  
6   } if (numero < 0) {  
7     return -1  
8   }  
9 }  
10 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



# El retorno del booleano

Para cerrar, ahora que ya vimos cómo escribir la alternativa condicional, es momento de un pequeño recordatorio: si usás adecuadamente las expresiones booleanas, ¡no es necesario utilizar esta estructura de control!

Supongamos que queremos desarrollar una función `esMayorDeEdad`, que nos diga si alguien tiene 18 años o más. Una tentación es escribir lo siguiente:

```
function esMayorDeEdad(edad) {  
  if (edad >= 18) {  
    return true;  
  } else {  
    return false;  
  }  
}
```

Sin embargo, este `if` es totalmente innecesario, dado que la expresión `edad >= 18` ya es booleana:

```
function esMayorDeEdad(edad) {  
  return edad >= 18;  
}
```

Mucho más simple, ¿no? 😊

Para Ema un número es de la suerte si:

- es positivo, y
- es menor a 100, y
- no es el 15.

Escribí la función `esNumeroDeLaSuerte` que dado un número diga si cumple la lógica anterior. ¡No vale usar `if`!

Solución  Biblioteca  Consola

```
1 function esNumeroDeLaSuerte(numero) {  
2   return (numero >= 0) && (numero < 100) && (numero != 15)  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

En general, como regla práctica, si tenés ifs que devuelven `true` o `false`s, probablemente lo estás haciendo mal 🤦. Y si bien *funcionará*, habrás escrito código innecesariamente complejo y/o extenso.

Recordá: ¡menos código, más felicidad! 😊

---

Esta guía fue desarrollada por Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 ↗ [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





## Los premios

El jurado de un torneo nos pidió que desarrollemos una función `medallaSegunPuesto` que devuelva la medalla que le corresponde a los primeros puestos, según la siguiente lógica:

- primer puesto: le corresponde "oro"
- segundo puesto: le corresponde "plata"
- tercer puesto: le corresponde "bronce"
- otros puestos: le corresponde "nada"

Ejemplo:

```
☒ medallaSegunPuesto(1)  
"oro"  
☒ medallaSegunPuesto(5)  
"nada"
```

Escribí, y probá en la consola, la función `medallaSegunPuesto`

Solución  Biblioteca  Consola

```
1 function medallaSegunPuesto(puesto) {  
2   if (puesto === 1) {  
3     return "oro";  
4   }  
5   if (puesto === 2) {  
6     return "plata";  
7   }  
8   if (puesto === 3) {  
9     return "bronce";  
10  } else {  
11    return "nada";  
12  }  
13}  
14  
15
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



# Tipos de datos

Como acabamos de ver, en JavaScript existen números, booleanos y strings:

Tipo de dato	Representa	Ejemplo	Operaciones
Números	cantidades	4947	+ , - , * , % , < , etc
Boolean	valores de verdad	true	&& , ! , etc
Strings	texto	"holá"	longitud , comienzaCon , etc

Además, existen operaciones que sirven para todos los *tipos de datos*, por ejemplo:

- === : nos dice si dos cosas son iguales
- !== : nos dice si dos cosas son diferentes

Es importante usar las operaciones correctas con los tipos de datos correctos, por ejemplo, no tiene sentido sumar dos booleanos o hacer operaciones booleanas con los números. Si usas operaciones que no corresponden, cosas muy raras y malas pueden pasar. ☺

Probá en la consola las siguientes cosas:

- 5 + 6 (ok, los números se pueden sumar)
- 5 === 6 (ok, todas las cosas se pueden comparar)
- 8 > 6 (ok, los números se pueden ordenar)
- !true (ok, los booleanos se pueden negar)
- false / true (no está bien, ¡los booleanos no se pueden dividir!)

☒ Consola ☒ Biblioteca

```
☒ 5 + 6
=> 11
☒ 5 === 6
=> false
☒ 8 > 6
=> true
☒ !true
=> false
```



## Datos de todo tipo

Uff, ¡vimos un montón de cosas! 😊 Aprendimos sobre la sintaxis de las funciones en JavaScript, los *tipos de datos* y sus operaciones, e incluso conocimos uno nuevo: los *strings*.

¡Pero no tan rápido! 🎉

Antes de terminar un último desafío: ¿Cuál es el valor de las siguientes expresiones? ¡Marcá todas las correctas!

- 4 + 4 vale 8
- "4" + "4" vale "44"
- 4 + 4 vale "44"
- "on" + "ce" vale "once"
- true && false vale false
- true && false vale 0
- 5 >= 6 vale false
- ! true vale false

✉ Enviar

✉ ¡La respuesta es correcta!



## Comprando Hardware

Queremos comprar una computadora nueva , y nos gustaría saber cuánto nos va a salir. Sabemos que:

- Los monitores cuestan \$60 por cada pulgada
- La memoria cuesta \$200 por cada GB
- El precio base estimado del resto de los componentes es de \$1000

Escribí una función `cuantoCuesta` que tome el número de pulgadas del monitor y la cantidad de memoria, y calcule el costo estimado de nuestra computadora.

```
✉ cuantoCuesta(25, 8)  
4100
```

Solución  Biblioteca  Consola

```
1 function cuantoCuesta(pulgada, gb) {  
2   return 60 * pulgada + 200 * gb + 1000;  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## ¿Me conviene?

Ahora que sabemos `cuantoCuesta` una computadora, queremos saber si una computadora *me conviene*. Esto ocurre cuando:

- sale menos de \$6000, y
- tiene al menos un monitor de 32 pulgadas, y
- tiene al menos 8GB de memoria

Escribí la función `meConviene`, que nuevamente tome el número de pulgadas y cantidad de memoria y nos diga si nos conviene comprarla :

```
☒ meConviene(25, 8)
false // porque el monitor es demasiado chico
☒ meConviene(42, 12)
true // cumple las tres condiciones
```

En la Biblioteca ya está definida la función `cuantoCuesta` lista para ser invocada.

☒ ¡Dame una pista!

¿Y cómo invoco `cuantoCuesta`? Pasándole como primer argumento el número de pulgadas y como segundo la cantidad de memoria. ☺

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function meConviene(pulgadas, gb) {
2   return cuantoCuesta(pulgadas, gb) < 6000 && pulgadas >= 32 && gb >= 8;
3 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Triangulos

¡Hora de hacer un poco de geometría! Queremos saber algunas cosas sobre un triángulo:

- `perimetroTriangulo`: dado los tres lados de un triángulo, queremos saber cuánto mide su perímetro.
- `areaTriangulo`: dada la base y altura de un triángulo, queremos saber cuál es su área.

Desarrollá las funciones `perimetroTriangulo` y `areaTriangulo`

☒ ¡Dame una pista!

⤷ Recordá que:

- el *perímetro* de un triángulo se calcula como la suma de sus tres *lados*;
- el *área* de un triángulo se calcula como su *base*, por su *altura*, dividido 2.

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function perimetroTriangulo(lado1, lado2, lado3) {  
2   return lado1 + lado2 + lado3;  
3 }  
4 function areaTriangulo(base, altura) {  
5   return base * altura / 2;  
6 }  
7
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Cuadrados

Y ahora es el turno de los cuadrados; queremos saber

- `perimetroCuadrado` : dado un lado, queremos saber cuánto mide su perímetro.
- `areaCuadrado` : dado un lado, queremos saber cuál es su área

Desarrollá las funciones `perimetroCuadrado` y `areaCuadrado`

¡Dame una pista!

Recordá que:

- el *perímetro* de un cuadrado se calcula como 4 veces su *lado*;
- el *área* de un cuadrado se calcula como su *lado* multiplicado por sí mismo.

Solución  Biblioteca  Consola

```
1 function perimetroCuadrado(lado) {  
2     return lado * 4;  
3 }  
4 function areaCuadrado(lado) {  
5     return lado * lado;  
6 }  
7
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## ¿Está afinado?

Cuando presionamos una tecla de un piano, éste produce un sonido que tiene una cierta [frecuencia](https://es.wikipedia.org/wiki/Frecuencia) (<https://es.wikipedia.org/wiki/Frecuencia>). Y cuando presionamos el *la* central del piano, si está afinado, vamos a escuchar [una nota cuya frecuencia es 440Hz](#) ([https://es.wikipedia.org/wiki/La\\_440](https://es.wikipedia.org/wiki/La_440)).



Desarrollá una función `estaAfinado`, que reciba la frecuencia (un número) del *la* central, y diga si dicha frecuencia es igual a 440Hz.

```
☒ estaAfinado(440)  
true
```

Solución  Biblioteca  Consola

```
1 function estaAfinado(frecuencia) {  
2   return frecuencia === 440;  
3 }  
4
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Siguiente Ejercicio: ¿Está cerca? [\(/argentina-programa/exercises/7089-programacion-imperativa-practica-funciones-y-tipos-de-datos-esta-](#)



## ¿Está cerca?

Ahora queremos saber si el *la* central del piano está *cerca* de estar afinado. Esto ocurre cuando está entre 437Hz y 443Hz, pero NO es exactamente 440Hz. Por ejemplo:

```
☒ estaCerca(443)
true //está en el rango 437-443
☒ estaCerca(442)
true //ídem caso anterior
☒ estaCerca(440)
false //está en el rango,
//pero es exactamente 440
☒ estaCerca(430)
false //está fuera del rango
```

Escribí la función `estaCerca`

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function estaCerca(frecuencia) {
2   return frecuencia >= 437 && frecuencia <= 443 && frecuencia !== 440;
3 }
4
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Cartelitos

Para una importante conferencia, el comité organizador nos pidió que escribamos cartelitos identificatorios que cada asistente va a tener.



Para eso, tenemos que juntar su nombre, su apellido, y su título (*dr., dra., lic., etc*) y armar un único string.

Escribí la función `escribirCartelito`, que tome un título, un nombre y un apellido y forme un único string. Por ejemplo:

```
☒ escribirCartelito("Dra.", "Ana", "Pérez")
"Dra. Ana Pérez"
```

☒ ¡Dame una pista!

Tené en cuenta que los espacios para separar las palabras también son caracteres. ¡No te olvides de incluirlos al armar los cartelitos! 😊

Por ejemplo:

```
☒ "Pepe" + " " + "Palotes"
"Pepe Palotes"
```

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function escribirCartelito(titulo, nombre, apellido) {
2   return titulo + " " + nombre + " " + apellido;
3 }
4
5
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Más Cartelitos

Ah, ¡pero no tan rápido! Algunas veces en nuestro cartelito 📜 sólo queremos el título y el apellido, sin el nombre. Por eso ahora nos toca mejorar nuestra función `escribirCartelito` de forma que tenga 4 parámetros:

1. el título;
2. el nombre;
3. el apellido;
4. new un booleano que nos indique si queremos un cartelito corto con sólo título y apellido, o uno largo, como hasta ahora.

Modificá la función `escribirCartelito`, de forma que se comporte como se describe arriba. Ejemplo:

```
// cartelito corto
☒ escribirCartelito("Lic.", "Tomás", "Peralta", true)
"Lic. Peralta"

// cartelito largo
☒ escribirCartelito("Ing.", "Dana", "Velázquez", false)
"Ing. Dana Velázquez"
```

☒ Solución ☒ Biblioteca ☒ Consola

```
1 //modificá esta función
2 function escribirCartelito(titulo, nombre, apellido, corto) {
3   if (corto) {
4     return titulo + " " + apellido;
5   } else {
6     return titulo + " " + nombre + " " + apellido;
7   }
8 }
9
10
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Cartelitos óptimos

Ahora que ya podemos escribir nuestros cartelitos identificatorios grandes y chicos, queremos una **nueva** función que nos dé el cartelito de tamaño óptimo:

- si nombre y apellido tienen, en total, más de 15 letras, queremos un cartelito corto;
- de lo contrario, queremos un cartelito largo.

Definí la función `escribirCartelitoOptimo` que tome un título, un nombre y un apellido, y utilizando `escribirCartelito` genere un cartelito corto o largo, según las reglas anteriores. Ejemplo:

```
☒ escribirCartelitoOptimo("Ing.", "Carla", "Toledo")
"Ing. Carla Toledo"
☒ escribirCartelitoOptimo("Dr.", "Estanislao", "Schwarzschild")
"Dr. Schwarzschild"
```

☒ Te dejamos en la biblioteca la función `escribirCartelito` definida. ¡Usala cuando necesites!

☒ ¡Dame una pista!

Recordá que contás con las funciones `longitud` y `escribirCartelito`, no tenés que definirlas solo invocarlas. ☺

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function escribirCartelitoOptimo(titulo, nombre, apellido) {
2   return escribirCartelito(titulo, nombre, apellido, (longitud(nombre + apellido) > 15));
3 }
4
5
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Cara o ceca

Hay veces en las que tenemos difíciles decisiones que tomar en nuestras vidas (*como por ejemplo, si comer pizzas o empanadas 😊*), y no tenemos más remedio que dejarlas libradas a la suerte.

Es allí que tomamos una moneda y decimos: *si sale cara, comemos pizzas, si no, empanadas.*

Escribí una función `decisionConMoneda`, que toma tres parámetros y devuelve el segundo si el primero es "cara", o el tercero, si sale "ceca". Por ejemplo:

```
☒ decisionConMoneda("cara", "pizzas", "empanadas")
  "pizzas"
```

☒ Solución ☒ Biblioteca ☒ Consola

```
1 function decisionConMoneda(lado, decision1, decision2) {
2   if (lado === "cara") {
3     return decision1;
4   } else {
5     return decision2;
6   }
7 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## ¡Envido!

Queremos saber el valor de las [cartas de truco](https://es.wikipedia.org/wiki/Truco_argentino) ([https://es.wikipedia.org/wiki/Truco\\_argentino](https://es.wikipedia.org/wiki/Truco_argentino)) cuando jugamos al *envido*. Sabemos que:

- todas las cartas del 1 al 7, inclusive, valen su numeración
- las cartas del 10 al 12, inclusive, valen 0
- no se juega con 8s ni con 9s

Escribí una función `valorEnvido`, que tome un número de carta y devuelva su valor de envido.

```
☒ valorEnvido(12)
0
☒ valorEnvido(3)
3
```

☒ ¡Dame una pista!

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function valorEnvido(carta) {
2   if (carta > 7) {
3     return 0;
4   } else {
5     return carta;
6   }
7 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## ¡Quiero retruco!

Bueno, ehm, no, pará, primero queremos calcular cuántos puntos de envíos suma un jugador. Sabemos que:

- Si las dos cartas son del mismo palo, el valor del envío es la suma de sus valores de envío más 20.
- De lo contrario, el valor del envío es el mayor valor de envío entre ellas.

Utilizando la función `valorEnvío` (que ya escribimos nosotros por vos), desarrollá la función `puntosDeEnvíoTotales` que tome los valores y palos de dos cartas y diga cuánto envío suman en total. Ejemplo:

```
☒ puntosDeEnvíoTotales(1, "espadas", 4, "espadas")
25
☒ puntosDeEnvíoTotales(2, "copas", 3, "bastos")
3
```

☒ ¡Dame una pista!

Solución  Biblioteca  Consola

```
1 function puntosDeEnvíoTotales(carta1, palo1, carta2, palo2) {
2   if (palo1 === palo2) {
3     return 20 + valorEnvío(carta1) + valorEnvío(carta2);
4   } else {
5     return Math.max(valorEnvío(carta1), valorEnvío(carta2));
6   }
7 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



# ¡Quiero vale cuatro!

Cuando se juega al truco, los equipos oponentes alternativamente pueden subir la apuesta. Por ejemplo, si un jugador canta *truco*, otro jugador puede cantarle *retruco*. Obviamente, los puntos que están en juego son cada vez mayores:

Canto	Puntos en juego
truco	2
retruco	3
vale cuatro	4

Escribí la función `valorCantoTruco`, que tome el canto y devuelva cuántos puntos vale.

```
☒ valorCantoTruco("retruco")  
3
```

⚠ Asumí que sólo te van a pasar como argumento un string que represente un canto de truco. Por ejemplo, no vamos a probar la función para el caso `valorCantoTruco("zaraza")`

☒ Solución ☒ Biblioteca ☒ Consola

```
1 function valorCantoTruco(canto) {  
2   if (canto === "truco") {  
3     return 2;  
4   }  
5   if (canto === "retruco") {  
6     return 3;  
7   }  
8   if (canto === "vale cuatro") {  
9     return 4;  
10  }  
11 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## ¿Y el tablero?

Hasta ahora en esta película hay un gran personaje que está faltando: *el tablero*. Seguro está por aparecer, de forma triunfal y rimbombante..., ¿no?

No. En JavaScript, lamentamos informarte, no hay tablero.

⌚ Bueeno, no llores, quizás fuimos un poco duros: en JavaScript no hay tablero, ¡porque no lo necesitas! 😊 Sucedan dos cosas:

1. El tablero nos servía para ver lo que nuestro programa hacía y qué resultados generaba. Nos permitía también observar los cambios de estado a causa del programa. Pero ahora ya tenemos experiencia suficiente como para lanzarnos a programar sin tener que "ver" lo que sucede.
2. Ahora contamos con la **consola**: una herramienta poderosa que nos permite hacer pruebas más detalladas y flexibles.

¿No nos creés? Te presentamos un desafío: usando la consola, decí con tus propias palabras qué hace la función `funcionMisteriosa`, que recibe dos números enteros como argumentos.

¡Vas a ver que podés averiguarlo sin tener un tablero!

```
☒  funcionMisteriosa(5, 8)
=> "woooooowwwwww!"
☒  funcionMisteriosa(1, 2)
=> "woww!"
☒  funcionMisteriosa(10, 15)
=> "ooooooooooooowwwwwwwwwwwww !"
☒
```



## Impresión por pantalla

Ahora que ya te convencimos de que no necesitamos al tablero, vamos a mostrarte que sí hay algo parecido en JavaScript 😊: la impresión por pantalla. Veamos un ejemplo:

```
function funcionEgocentrica() {  
    imprimir("soy una función que imprime por pantalla");  
    imprimir("y estoy por devolver el valor 5");  
    return 5;  
}
```

Probá `funcionEgocentrica` en la consola.

```
funcionEgocentrica  
=> <function>  
funcionEgocentrica()  
soy una función que imprime por pantalla  
y estoy por devolver el valor 5  
=> 5  
funcionEgocentrica()
```



## Martin Fierro

¿Qué acabamos de hacer con esto? Al igual que `Poner(bolita)`, `imprimir` es una funcionalidad que siempre está disponible. Si llamamos a la función anterior, veremos que, además de devolver el valor 5, imprime dos líneas:

```
soy una función que imprime por pantalla  
y estoy por devolver el valor 5
```

Sin embargo, sólo podemos escribir strings y, una vez que escribimos en la pantalla, no hay vuelta atrás: no hay forma de retroceder o deshacer.

Veamos si va quedando claro, escribí una `function versosMartinFierro` que imprima por pantalla los primeros versos del Martín Fierro:

```
Aquí me pongo a cantar  
Al compás de la vigüela;  
Que el hombre que lo desvela  
Una pena extraordinaria
```

Esta function debe devolver 0

[¡Dame una pista!](#)

[Solución](#) [Consola](#)

```
1 function versosMartinFierro() {  
2   imprimir("Aquí me pongo a cantar");  
3   imprimir("Al compás de la vigüela;");  
4   imprimir("Que el hombre que lo desvela");  
5   imprimir("Una pena extraordinaria");  
6   return 0;  
7 }  
8 }  
9 }
```

[Enviar](#)

**¡Muy bien! Tu solución pasó todas las pruebas**

¡Bien hecho! 🎉

Sin embargo, ¿tiene sentido que `versosMartinFierro` devuelva 0? ¿Usamos para algo este resultado? 🤔

Acá parecería que llamamos a esta `function` porque nos interesa su efecto de imprimir líneas; nos da igual lo que retorna. Quizás más que una función, necesitamos definir un procedimiento. ¿Se podrá hacer esto en JavaScript?

La respuesta, ¡en el siguiente ejercicio!



## ¿Y los procedimientos?

En el ejercicio anterior, construiste una `function` que se ejecutaba con el sólo fin de imprimir por pantalla. Y por ello, tuvimos que devolver un valor cualquiera. ¡No te huele mal!

Además, hagamos memoria: cuando queremos reutilizar código, podíamos declarar:

- *funciones*, que siempre devuelven algo y no producen ningún efecto
- *procedimientos*, que no devuelven nada, y producen efectos

Entonces `versosMartinFierro`, no es una función... ¡sino un procedimiento! ¿Cómo se declaran procedimientos en JavaScript?

¡De la misma forma que las funciones!: usando la palabra clave `function`.

```
function versosMartinFierro() {  
    imprimir("Aquí me pongo a cantar");  
    imprimir("Al compás de la vigüela;");  
    imprimir("Que el hombre que lo desvela");  
    imprimir("Una pena extraordinaria");  
}
```

Envía esta nueva versión de `versosMartinFierro`

Solución  Consola

```
1 function versosMartinFierro() {  
2     imprimir("Aquí me pongo a cantar");  
3     imprimir("Al compás de la vigüela;");  
4     imprimir("Que el hombre que lo desvela");  
5     imprimir("Una pena extraordinaria");  
6 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Esto puede ser un poco perturbador 😊: JavaScript no diferencia funciones de procedimientos: todos pueden tener efectos y todos pueden o no tener retorno.

Vos sos responsable de escribir una `function` que tenga sentido y se comporte o bien como un procedimiento (sin retorno y con efecto) o bien como una función (con retorno y sin efecto).

Si empezás a mezclar funciones con retornos y efecto, funcionará, pero tu código se volverá de a poco más difícil de entender. Esto nos va a pasar mucho en JavaScript: que puedas hacer algo no significa que debas hacerlo 😊.



## ¿Y el program?

Ahora bien, más allá de que podamos consultar el resultado de una función a través de la consola, también aprendimos anteriormente que los programas tienen un punto de entrada: el `program`. ¿Dónde quedó?

La respuesta es tan simple como sorprendente: en JavaScript todo lo que escribamos fuera de una `function` será, implícitamente, dicho punto de entrada. Por ejemplo, si queremos un programa que imprime por pantalla el clásico "Hola, mundo!", lo podremos escribir así:

```
imprimir("Hola, mundo!");
```

Si queremos un programa que tire tres veces los dados e imprima sus resultados, podemos escribirlo así:

```
imprimir("Tirando dados");
imprimir("La primera tirada dio " + tirarDado());
imprimir("La segunda tirada dio " + tirarDado());
imprimir("La tercera tirada dio " + tirarDado());
```

Copíá y envíá este programa

[Solución](#) [Consola](#)

```
1 imprimir("Tirando dados");
2 imprimir("La primera tirada dio " + tirarDado());
3 imprimir("La segunda tirada dio " + tirarDado());
4 imprimir("La tercera tirada dio " + tirarDado());
```

[Enviar](#)

[¡Muy bien! Tu solución pasó todas las pruebas](#)

¿Ooooups, y el resultado? ¿Dónde está lo que imprimimos por pantalla? ¿Es que nuestro programa no anduvo?

No, para nada, es que simplemente no te estamos mostrando lo que sale por pantalla 😊.

¿Por qué? ¿Porque somos malvados? Bueno, quizás en parte 🤪, pero tenemos además una buena razón: cuando escribís programas reales, es muy, **muy** frecuente que no sea fácil ver lo que el `imprimir` imprime, por decenas de motivos. Entonces, como rara vez vas poder ver *a tiempo* lo que se imprime en la pantalla, terminan siendo una técnica poco útil.

Moraleja: en los ejercicios que quedan, **no uses `imprimir` salvo que te lo pidamos explícitamente**.

¡Nos vemos en el próximo ejercicio!



## Coerciones

6. Coerciones

Volvamos un momento al código anterior. ¿Notás algo extraño en esta expresión?

```
"La primera tirada dio " + primeraTirada
```

Utilizamos el operador `+` de una forma diferente, operando un string y un número, y lo que hizo fue concatenar al string con la representación textual del número. Es decir que:

- si operamos dos números con `+`, se suman
- si operamos dos strings con `+`, se concatenan
- si operamos un string y un número `+`, se convierte *implícitamente* el número a string, y luego se concatenan, al igual que antes

En JavaScript, estas conversiones implícitas, también llamadas *coerciones*, ocurren mucho.

;Quizás incluso [más de lo que nos gustaría](https://archive.org/details/wat_destroyallsoftware) ([https://archive.org/details/wat\\_destroyallsoftware](https://archive.org/details/wat_destroyallsoftware))! 😊

Veamos si queda claro, escribí una función `elefantesEquilibristas`, que tome un número de elefantes y devuelva una rima de una conocida canción:

```
elefantesEquilibristas(3)  
"3 elefantes se balanceaban"  
elefantesEquilibristas(462)  
"462 elefantes se balanceaban"
```

Solución  Consola

```
1 function elefantesEquilibristas(numero) {  
2   return numero + " elefantes se balanceaban"  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## El círculo de la vida

En programación buscamos que resolver nuestros problemas usando... programas ⓘ. Y entre los problemas que casi nadie quiere resolver están los matemáticos. Sobre todo aquellos que aparecen números como pi con infinitos decimales imposibles de recordar. 😱

Considerando al número pi igual a 3.14159265358979 (no es infinito pero lo suficientemente preciso para nuestros cálculos):

Definí las funciones `perimetroCirculo` y `areaCirculo` que reciben el radio de un círculo y a partir del mismo nos devuelven su perímetro y su área.

☒ ¡Dame una pista!

☒ Solución ☒ Consola

```
1 function perimetroCirculo(radio) {  
2   return radio * 2 * 3.14159265358979;  
3 }  
4  
5 function areaCirculo(radio) {  
6   return radio * radio * 3.14159265358979;  
7 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

Excelente, la precisión de nuestros cálculos es innegable ⓘ, pero tuvimos que escribir un número larguísimo. Pensemos que pi aparece en un montón de fórmulas matemáticas. ¿Es necesario escribir este número cada vez? ¿No podemos hacer algo más cómodo? 😊



## Plenso que así es más fácil

8. Plenso qu

Por suerte existe una herramienta que va a simplificar nuestra tarea de ahora en adelante: las *variables*. 😊

Las variables nos permiten nombrar y reutilizar *valores*. Similar a cómo los procedimientos y funciones nos permiten dar nombres y reutilizar soluciones a problemas más pequeños. Por ejemplo, si hacemos...

```
let primerMes = "enero"
```

...estamos *asignándole* el valor "enero" a la variable `primerMes`. En criollo, estamos dándole ese valor a la variable. ☺

Cambiá los lugares donde aparece `3.14159265358979` por la variable `pi` en las funciones que tenemos definidas.

☒ Solución ☒ Consola

```
1 let pi = 3.14159265358979;
2
3 function perimetroCirculo(radio) {
4   return radio * 2 * pi;
5 }
6
7 function areaCirculo(radio) {
8   return radio * radio * pi;
9 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Excelente! Gracias a la variable `pi` no tuvimos que escribir el número cada vez que teníamos que usarlo y ¡nuestro programa quedó mucho más entendible! 🎉



## Esto no tiene valor

Ya que vas entendiendo cómo se **asignan** las variables, te traemos algo para pensar: ¿qué pasa si intento **usar** una variable a la que nunca le asigné un valor? 🤔

Tenemos esta función definida:

```
function sumaSinSentido() {  
    return numero + 8;  
}
```

Probala en la consola y fíjate qué sucede.

```
ReferenceError: numero is not defined  
  sumaSinSentido(2)  
return numero + 8;  
^  
  
ReferenceError: numero is not defined  
  sumaSinSentido(2)
```



## VARIABLES GLOBALES

Entonces, ¿es necesario darle valor a nuestras variables antes de usarlas?

¡Sí! 😊 Cuando declarás una variable tenés que darle un valor inicial, lo cual se conoce como *inicializar* la variable.

¡Y sorpresa! Podemos declarar variables tanto directamente en el programa, como dentro de una `function`:

```
function cuentaLoca(unNumero) {
  let elDoble = unNumero * 2;
  if (elDoble > 10) {
    return elDoble;
  } else {
    return 0;
  }
}
```

Las variables declaradas dentro de una `function`, conocidas como *variables locales*, no presentan mayor misterio. Sin embargo, hay que tener un particular cuidado: sólo se pueden utilizar desde dentro de la `function` en cuestión. Si quiero referenciarla desde un programa:

```
let elCuadruple = elDoble * 4;
```

Kaboom, ¡se romperá! 💣

Sin embargo, las variables declaradas directamente en el programa, conocidas como *variables globales*, pueden ser utilizadas desde cualquier `function`. Por ejemplo:

```
let pesoMaximoEquipajeEnGramos = 5000;

function puedeLlevar(pesoEquipaje) {
  return pesoEquipaje <= pesoMaximoEquipajeEnGramos;
}
```

Veamos si queda claro: escribí una función `ascensorSobrecargado`, que toma una cantidad de personas y dice si entre todas superan la carga máxima de 300 kg.

Tené en cuenta que nuestra función va a utilizar dos variables globales:

- `pesoPromedioPersonaEnKilogramos`, la cual ya está declarada,
- `cargaMaximaEnKilogramos` que vas a tener que declarar.

✖ Solución ✖ Consola

```
1 let cargaMaximaEnKilogramos = 300;
2
3 function ascensorSobrecargado(cantidadpersonas) {
4   return cantidadpersonas * pesoPromedioPersonaEnKilogramos >= cargaMaximaEnKilogramos;
5 }
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





## Volviéndonos ricos

Las variables no serían tan interesantes si no se pudieran modificar. Afortunadamente, JavaScript nos da nuevamente el gusto y nos lo permite:

```
function pasarUnDiaNormal() {  
    diasSinAccidentesConVelociraptores = diasSinAccidentesConVelociraptores + 1  
}  
  
function tenerAccidenteConVelociraptores() {  
    diasSinAccidentesConVelociraptores = 0;  
}
```

¡Ahora vamos a hacer algo de dinero 💰!

Escribí un procedimiento `aumentarFortuna` que duplique el valor de la variable global `pesosEnMiBilletera`. No declares la variable, ya lo hicimos nosotros por vos (con una cantidad secreta de dinero 😊).

✖ ¡Dame una pista!

¿Cómo usar `aumentarFortuna`? Por ejemplo así:

```
// Consulto la variable a ver cuánto tiene:  
✖ pesosEnMiBilletera  
500 // ¡Ojo! Esto es a fines ilustrativos;  
// ¡Podría tener cualquier cantidad!  
// Aumento mi fortuna:  
✖ aumentarFortuna()  
// Consulto de nuevo mi fortuna:  
✖ pesosEnMiBilletera // ¡Aumentó!  
1000
```

✖ Solución

✖ Consola

```
1 function aumentarFortuna() {  
2     pesosEnMiBilletera = pesosEnMiBilletera * 2  
3 }  
4  
5
```

✖ Enviar

✖ ¡Muy bien! Tu solución pasó todas las pruebas

Actualizaciones como duplicar, triplicar, incrementar en uno o en una cierta cantidad son tan comunes que JavaScript presenta algunos atajos:

```
x += y; //equivalente a x = x + y;  
x *= y; //equivalente a x = x * y;  
x -= y; //equivalente a x = x - y;  
x++; //equivalente a x = x + 1;
```

¡Usalos cuando quieras! ☺

Esta guía fue desarrollada por Franco Bulgarelli bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





## ¿Y esto cuánto vale?

Vimos que una variable solo puede tener un valor, entonces cada vez que le asignamos uno nuevo, perdemos el anterior. Entonces, dada la función:

```
function cuentaLoca() {  
  let numero = 8;  
  numero *= 2;  
  numero += 4;  
  return numero;  
}
```

¿Qué devuelve cuenta\_loca?

- "numero"
- 8
- 16
- 20
- true

Enviar

¡La respuesta es correcta!



# ¡Que el último apague la luz!

Empecemos por algo sencillo, ¿te acordás del operador `!`? Se lo denomina negación, not o complemento lógico y sirve para negar un valor booleano.

Si tengo el booleano representado por `tieneHambre`, el complemento será `!tieneHambre`.

¿Y esto para qué sirve? ☺ Por ejemplo, para modelar casos de alternancia como prender y apagar una luz ☺:

```
let lamparaPrendida = true;

function apretarInterruptor() {
    lamparaPrendida = !lamparaPrendida;
}
```

¡Ahora te toca a vos!

Definí el procedimiento `usarCierre` para que podamos abrir y cerrar el cierre de una mochila.

[Solución](#) [Consola](#)

```
1 let mochilaAbierta = true;
2
3 function usarCierre() {
4     mochilaAbierta = !mochilaAbierta;
5 }
```

[Enviar](#)

[¡Muy bien! Tu solución pasó todas las pruebas](#)



## Negar no cuesta nada

Por el momento no parece una idea muy interesante, pero nos puede servir para reutilizar la lógica de una función que ya tenemos definida.

Por ejemplo, si contamos con una función `esPar`, basta con negarla para saber si un número es impar.

```
function esImpar(numero) {  
    return !esPar(numero);  
}
```

¡Ahora te toca a vos! Definí `esMayorDeEdad`, que recibe una edad, y luego `esMenorDeEdad` a partir de ella.

[Solución](#) [Consola](#)

```
1 function esMayorDeEdad(edad) {  
2     return edad >= 18;  
3 }  
4  
5 function esMenorDeEdad(edad) {  
6     return !esMayorDeEdad(edad);  
7 }
```

[Enviar](#)

[¡Muy bien! Tu solución pasó todas las pruebas](#)

Cada una de las funciones representa **un estado de dos posibles**: ser mayor o ser menor de edad. No se puede ser ambos al mismo tiempo y tampoco se puede evitar pertenecer a alguno de los dos grupos. Es decir, ¡siempre sos uno u otro! 😊

Por eso decimos que son complementarios y que juntos forman el *conjunto universal*.



## Los peripatéticos

Otro de los operadores con el que ya te encontraste es la conjunción lógica (también llamada *y lógico*, o *and* por su nombre en inglés), que sólo retorna verdadero cuando todas las expresiones que opera son verdaderas.

Podemos encadenar varias de ellas mediante el operador `&&` y alcanza con que sólo una de ellas sea falsa para que toda la expresión resulte falsa.

Por ejemplo, si cuento con la función:

```
function esCantanteProlifico (cdsEditados, recitalesRealizados, graboAlgunDVD) {  
    return cdsEditados >= 10 && recitalesRealizados > 250 && graboAlgunDVD;  
}
```

y tenemos un cantante que no grabó un DVD, entonces no se lo considera [prolífico](http://dle.rae.es/?id=UKzl2xC) (<http://dle.rae.es/?id=UKzl2xC>), incluso aunque haya editado más de 10 CDs y dado más de 250 recitales.

Definí una función `esPeripatetico` que tome la profesión de una persona, su nacionalidad y la cantidad de kilómetros que camina por día. Alguien es peripatético cuando es un filósofo griego y le gusta pasear (camina más de 2 kilómetros por día). Ejemplo:

```
☒ esPeripatetico("filósofo", "griego", 5)  
true  
☒ esPeripatetico("profesor", "uruguayo", 1)  
false
```

Solución  Consola

```
1 function esPeripatetico(profesion, nacionalidad, kilometros) {  
2     return profesion === "filósofo" && nacionalidad === "griego" && kilometros > 2;  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



# La verdad detrás de la conjunción

En la lógica booleana, se puede definir el comportamiento de un operador con una *tabla de verdad* donde **A** y **B** son las expresiones o valores de verdad a ser operados y el símbolo **^** representa la conjunción. Cada celda tiene una V si representa verdadero o F si representa falso.

Por ejemplo, supongamos que una casa consume poca energía si se usa el aire acondicionado a 24 grados y tiene al menos 5 lamparitas bajo consumo. Podemos representar las expresiones de la siguiente forma:

- **A:** En la casa se usa el aire acondicionado a 24 grados
- **B:** La casa tiene al menos 5 lamparitas bajo consumo
- **A ^ B:** La casa consume poca energía

Como indicamos, la casa consume poca energía (**A ^ B**) cuando tanto **A** como **B** son verdaderos. Esto se puede representar mediante la siguiente tabla de verdad:

<b>A</b>	<b>B</b>	<b>A ^ B</b>
V	V	V
V	F	F
F	V	F
F	F	F

En el mundo de la lógica estas expresiones se llaman *proposiciones*. Pero... ¿qué cosas pueden ser una proposición? Sólo hace falta que porten un valor de verdad, es decir, cualquier expresión booleana puede ser una proposición.

¿No nos creés? Probá en la consola la función `consumePocaEnergia`, que recibe una temperatura y una cantidad de lamparitas, y comprobá si se comporta como en la tabla:

- `consumePocaEnergia(24, 5)`
- `consumePocaEnergia(24, 0)`
- `consumePocaEnergia(21, 7)`
- `consumePocaEnergia(18, 1)`

```

 consumePocaEnergia(24, 5)
=> true

 consumePocaEnergia(24, 0)
=> false

 consumePocaEnergia(21, 7)
=> false

 consumePocaEnergia(18, 1)
=> false
  
```



## ¡Juguemos al T.E.G.!

¿Y si basta con que una de varias condiciones se cumpla para afirmar que una expresión es verdadera? Podemos utilizar otro de los operadores que ya conocés, ¡la disyunción lógica! ☺

Recordá que se lo representa con el símbolo `||` y también se lo conoce como el operador `or`.

En el famoso juego [T.E.G.](https://es.wikipedia.org/wiki/TEG) (<https://es.wikipedia.org/wiki/TEG>), un jugador puede ganar de dos formas: cumpliendo su objetivo secreto o alcanzando el objetivo general de conquistar 30 países.

```
function gano(cumplioObjetivoSecreto, cantidadDePaisesConquistados) {  
    return cumplioObjetivoSecreto || cantidadDePaisesConquistados >= 30;  
}
```

Probá en la consola las siguientes expresiones:

- ☒ `gano(true, 25)`
- ☒ `gano(false, 30)`
- ☒ `gano(false, 20)`
- ☒ `gano(true, 31)`

¿Te animás a construir la tabla de verdad de la disyunción lógica?

```
☒ gano(true, 25)  
=> true  
  
☒ gano(false, 30)  
=> true  
  
☒ gano(false, 20)  
=> false  
  
☒ gano(true, 31)  
=> true
```



## Y ahora... ¿quién podrá ayudarnos?

Nuestra amiga Dory 🐟 necesitaba hacer algunos trámites en el banco, pero cuando llegó notó que estaba cerrado. ☹

Para evitar que le ocurra nuevamente, vamos a desarrollar una función que ayude a la gente despistada como ella.

Sabemos que el banco está cerrado cuando:

- Es feriado, o
- Es fin de semana, o
- No estamos dentro del horario bancario.

La función `dentroDeHorarioBancario` ya la definimos por vos: recibe un horario ⏰ (una hora en punto que puede ir desde las 0 hasta las 23) y nos dice si está comprendido en la franja de atención del banco.

Definí las funciones `esFinDeSemana` y `estaCerrado`.

Solución  Biblioteca  Consola

```
1 function esFinDeSemana(dia) {  
2   return dia === "sabado" || dia === "domingo";  
3 }  
4  
5 function estaCerrado(esFeriado, dia, horario) {  
6   return esFeriado || esFinDeSemana(dia) || !dentroDeHorarioBancario(horario);  
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## ¡Buen día!

Todos sabemos que el seguimiento de árboles genealógicos puede tornarse complicado cuando hay muchas personas y relaciones involucradas.

Por ejemplo, en la familia Buendía ocurre que:

- Arcadio es hijo de José Arcadio y de Pilar Ternera
- Aureliano José es hijo del Coronel Aureliano y Pilar Ternera
- Aureliano Segundo y Remedios son hijos de Arcadio y Sofía De La Piedad

Para empezar a analizar esta familia, nosotros ya definimos las funciones `madreDe` y `padreDe`:

```
☒ padreDe(aurelianoJose)
"Coronel Aureliano"
☒ madreDe(aurelianoSegundo)
"Sofía De La Piedad"
```

Ahora te toca a vos definir la función `sonMediosHermanos`. Recordá que los medios hermanos pueden compartir madre o padre pero no ambos porque... ¡en ese caso serían hermanos! 😊

☒ ¡Dame una pista!

Quizás te sirva definir las funciones `tienenLaMismaMadre` y `tienenElMismoPadre`.

☒ Solución ☒ Consola

```
1 function tienenLaMismaMadre(hijo1, hijo2) {
2   return madreDe(hijo1) === madreDe(hijo2);
3 }
4
5 function tienenElMismoPadre(hijo1, hijo2) {
6   return padreDe(hijo1) === padreDe(hijo2);
7 }
8
9 function sonMediosHermanos(hijo1, hijo2) {
10   return (tienenElMismoPadre(hijo1, hijo2) && !tienenLaMismaMadre(hijo1, hijo2)) ||
11   (tienenLaMismaMadre(hijo1, hijo2) && !tienenElMismoPadre(hijo1, hijo2))
12 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

Esta guía fue desarrollada por Jessica Saavedra bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





## La verdad es que no hay una verdad

Ahora pensemos cómo sería la tabla de verdad que representa el comportamiento de la función que acabás de hacer.

Las proposiciones serán `tienenLaMismaMadre` y `tienenElMismoPadre`, y los valores de verdad que porten dependerán de qué dos personas estén evaluando.

El booleano final resultará de operarlas mediante `sonMediosHermanos`:

<b>tienen la misma madre</b>	<b>tienen el mismo padre</b>	<b>son medios hermanos</b>
true	true	false
true	false	true
false	true	true
false	false	false

Probá tu función `sonMediosHermanos` con los siguientes valores y comprobá si se comporta como la tabla:

- ☒ `sonMediosHermanos(arcadio, aurelianoJose)`
- ☒ `sonMediosHermanos(aurelianoSegundo, remedios)`
- ☒ `sonMediosHermanos(aurelianoJose, remedios)`

☒ .;Dame una pista!

Recordá que en la familia Buendía:

- Arcadio es hijo de José Arcadio y de Pilar Ternera
- Aureliano José es hijo del Coronel Aureliano y Pilar Ternera
- Aureliano Segundo y Remedios son hijos de Arcadio y Sofía De La Piedad

```
☒ sonMediosHermanos(arcadio, aurelianoJose)
=> true
☒ sonMediosHermanos(aurelianoSegundo, remedios)
=> false
☒ sonMediosHermanos(aurelianoJose, remedios)
=> false
☒
```



# ¡Hola! Mi nombre es Xor

Ahora cambiemos las funciones `tienenLaMismaMadre` y `tienenElMismoPadre` por proposiciones genéricas **A** y **B**. Además, representemos la operación que realiza `sonMediosHermanos` con el símbolo  $\vee$ . Lo que obtenemos es... ¡una nueva tabla! 🎉

A	B	$A \vee B$
V	V	F
V	F	V
F	V	V
F	F	F

Este comportamiento existe como un operador dentro de la lógica y se lo denomina `xor` o disyunción lógica excluyente.

A diferencia del `and`, `or` y `not`, el `xor` no suele estar definido en los lenguajes. 😕 Sin embargo, ahora que sabés cómo funciona, si alguna vez lo necesitás podés definirlo a mano. 😊

Veamos si se entiende: definí la función genérica `xor`, que tome dos booleanos y devuelva el valor de verdad correspondiente.

Solución  Consola

```
1 function xor(a, b){  
2   return (a && !b) || (!a && b);  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## Precedencia

Cuando una expresión matemática tiene varios operadores, sabemos que las multiplicaciones y divisiones se efectuarán antes que las sumas y las restas:

```
5 * 3 + 8 / 4 - 3 = 14
```

Al igual que en matemática, cuando usamos operadores lógicos las expresiones se evalúan en un orden determinado llamado *precedencia*.

¿Cuál es ese orden? ¡Hagamos la prueba!

Teniendo definida la siguiente función, según la cual las tarjetas de débito ofrecen una única cuota, y las de crédito, seis:

```
function pagaConTarjeta(seCobraInteres, tarjeta, efectivoDisponible) {  
    return !seCobraInteres && cuotas(tarjeta) >= 3 || efectivoDisponible < 100;  
}
```

Probala en la consola con los valores:

- ☒ pagaConTarjeta(true, "crédito", 320)
- ☒ pagaConTarjeta(false, "crédito", 80)
- ☒ pagaConTarjeta(true, "débito", 215)
- ☒ pagaConTarjeta(true, "débito", 32)

```
☒ pagaConTarjeta(true, "crédito", 320)  
=> false  
☒ pagaConTarjeta(false, "crédito", 80)  
=> true  
☒ pagaConTarjeta(true, "débito", 215)  
=> false  
☒ pagaConTarjeta(true, "débito", 32)  
=> true
```



# Un ejercicio sin precedentes

Si prestaste atención a la función anterior, habrás notado que la operación con mayor precedencia es la negación `!`, seguida de la conjunción `&&` y por último la disyunción `||`. ¿Pero qué pasa si quiero alterar el orden en que se resuelven? 😊

Al igual que en matemática, podemos usar paréntesis para agrupar las operaciones que queremos que se realicen primero.

Escribí la función `puedeJubilarse` que recibe la edad y el sexo de una persona, además de los años de aportes jubilatorios que posee:

```
☒ puedeJubilarse(62, 'F', 34)
true
```

El mínimo de edad para realizar el trámite para las mujeres es de 60 años, mientras que para los hombres es 65. En ambos casos, se deben contar con al menos 30 años de aportes.

¡Intentá resolverlo en una única función! Despúes vamos a ver cómo quedaría si delegamos. 😊

Solución  Consola

```
1 /* primera opcion:
2 function puedeJubilarse(edad, sexo, aporte) {
3   return aporte >= 30 && ((sexo === "F" && edad >= 60) || (sexo === "M" && edad >= 65))
4 }
5
6 opcion delegando:
7 */
8
9 function cumpleEdadMinima(edad, sexo) {
10   return (sexo === "F" && edad >= 60) || (sexo === "M" && edad >= 65);
11 }
12
13 function tieneSuficientesAportes(aporte) {
14   return aporte >= 30;
15 }
16
17 function puedeJubilarse(edad, sexo, aporte) {
18   return cumpleEdadMinima(edad, sexo) && tieneSuficientesAportes(aporte);
19 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¿Y si delegamos? Podríamos separar la lógica de la siguiente manera:

```
function puedeJubilarse(edad, sexo, aniosAportes) {
  return cumpleEdadMinima(edad, sexo) && tieneSuficientesAportes(aniosAportes);
}
```

Al **delegar correctamente**, hay veces en las que no es necesario alterar el orden de precedencia, ¡otro punto a favor de la delegación! 🎉



## ¿Puedo subir?

En un parque de diversiones de la ciudad instalaron una nueva montaña rusa y nos pidieron ayuda para que le digamos a las personas si pueden subirse o no antes de hacer la fila. Los requisitos para subir a la atracción son:

- Alcanzar la altura mínima de 1.5m (o 1.2m si está acompañada por un adulto)
- No tener ninguna afección cardíaca

Definí la función de 3 parámetros `puedeSubirse` que recibe una altura de una persona en metros, si está acompañada por un adulto y si tiene alguna afección cardíaca. Ejemplo:

```
☒ puedeSubirse(1.7, false, true)
false // no puede subirse
    // porque aunque tiene mas de 1.5m,
    // tiene una afección cardíaca
```

Solución  Consola

```
1 function puedeSubirse(altura, acompañadaAdulto, afeccionCardiaca){
2     return ((altura >= 1.5) || (altura >= 1.2 && acompañadaAdulto)) && !afeccionCardiaca
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## Series favoritas

Supongamos que queremos representar al conjunto de nuestras series favoritas. ¿Cómo podríamos hacerlo?

```
let seriesFavoritasDeAna = ["Game of Thrones", "Breaking Bad", "House of Cards"];
let seriesFavoritasDeHector = ["En Terapia", "Recordando el Show de Alejandro Molina"]
```

Como ves, para representar a un conjunto de strings, colocamos todos esos strings que nos interesan, entre corchetes ( [ ] ) separados por comas. Fácil, ¿no?

Probá en la consola las siguientes consultas:

- seriesFavoritasDeAna
- seriesFavoritasDeHector
- ["hola", "mundo!"]
- ["hola", "hola"]

☒ Consola ☒ Biblioteca

```
☒ seriesFavoritasDeAna
=> ["Game of Thrones", "Breaking Bad", "House of Cards"]
☒ seriesFavoritasDeHector
=> ["En Terapia", "Recordando el Show de Alejandro Molina"]
☒ ["hola", "mundo!"]
=> ["hola", "mundo!"]
☒ ["hola", "hola"]
=> ["hola", "hola"]
```



## Y esto, es una lista

Lo que acabamos de ver es cómo modelar fácilmente conjuntos de cosas. Mediante el uso de `[]`, en JavaScript contamos con una manera simple de agrupar esos elementos en listas.

¿Acaso hay una cantidad máxima de elementos? ¡No, no hay límite! Las listas pueden tener cualquier cantidad de elementos.

Y no sólo eso, sino que además, el orden es importante. Por ejemplo, no es lo mismo `["hola", "mundo"]` que `["mundo", "hola"]`: ambos tienen los mismos elementos, pero en posiciones diferentes.

Probá en la consola las siguientes consultas:

- `listasIguales(["hola", "mundo"], ["mundo", "hola"])`
- `listasIguales(["hola", "mundo"], ["hola", "mundo"])`
- `listasIguales(["hola", "mundo"], ["hola", "todo", "el", "mundo"])`
- `listasIguales(["hola"], ["hola", "mundo"])`
- `["hola", "mundo"] === ["mundo", "hola"]`
- `personas`
- `["mara", "julian"] === personas`
- `personas === personas`

¿Qué conclusiones podés sacar? ☺

☒ Consola ☒ Biblioteca

```
☒ listasIguales(["hola", "mundo"], ["hola", "todo", "el", "mundo"])
=> false
☒ listasIguales(["hola"], ["hola", "mundo"])
=> false
☒ ["hola", "mundo"] === ["mundo", "hola"]
=> false
☒ personas
=> ["mara","julian"]
```

Esta guía fue desarrollada por Franco Bulgarelli, Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





## Juegos de azar

Pero, pero, ¿sólo podemos crear listas de strings? ¿Y si quiero, por ejemplo, representar los números de la lotería que salieron la semana pasada? ¿O las tiradas sucesivas de un dado? ¿O si salió cara o ceca en tiradas sucesivas de una moneda?

```
let numerosDeLoteria = [2, 11, 17, 32, 36, 39];
let tiradasDelDado = [1, 6, 6, 2, 2, 4];
let salioCara = [false, false, true, false];
```

Como ves, también podemos representar conjuntos de números o booleanos, de igual forma: escribiéndolos entre corchetes y separados por comas. Podemos tener listas de números, de strings, de booleanos, etc. ¡Incluso podríamos tener listas de listas!

Veamos si queda claro. Probá en la consola las siguientes consultas:

- numerosDeLoteria
- salioCara
- [[1, 2, 3], [4, 5, 6]]

☒ Consola ☒ Biblioteca

```
☒ numerosDeLoteria
=> [2,11,17,32,36,39]
☒ salioCara
=> [false,false,true,false]
☒ [[1, 2, 3], [4, 5, 6]]
=> [[[1,2,3],[4,5,6]]]
```



## Listas vacías

Genial, ¡parece que una lista puede contener cualquier tipo de elemento! Podemos tener listas de booleanos, de números, de strings, de listas...

Y no sólo eso, sino que además pueden contener cualquier cantidad de elementos: uno, dos, quince, cientos.

¿Podremos entonces tener listas vacías, es decir, que no tengan elementos? ¡Por supuesto!

```
let unaListaVacia = []
```

Probá escribir en la consola `unaListaVacia`

Consola  Biblioteca

```
unaListaVacia
```

```
=> []
```



## ¿Cuántos elementos tenés?

Por el momento ya sabemos qué cosas podemos representar con listas, y cómo hacerlo. Pero, ¿qué podemos hacer con ellas?

Empecemos por lo fácil: saber cuántos elementos hay en la lista. Esto lo podemos hacer utilizando la función `longitud`, de forma similar a lo que hacíamos con los strings.

Realizá las siguientes consultas en la consola:

- `longitud([])`
- `longitud(numerosDeLoteria)`
- `longitud([4, 3])`

☒ Consola ☒ Biblioteca

```
☒ longitud([])
=> 0
☒ longitud(numerosDeLoteria)
=> 6
☒ longitud([4, 3])
=> 2
☒
```



## Agregando sabor

Las listas son muy útiles para contener múltiples elementos. ¡Pero hay más! También podemos agregarle elementos en cualquier momento, utilizando la función `agregar`, que recibe dos parámetros: la lista y el elemento. Por ejemplo:

```
let pertenencias = ["espada", "escudo", "antorchas"];
//longitud(pertenencias) devuelve 3;

agregar(pertenencias, "amuleto mágico");
//ahora longitud(pertenencias) devuelve 4
```

Como vemos, `agregar` suma un elemento a la lista, lo cual hace que su tamaño aumente. ¿Pero en qué parte de la lista lo agrega? ¿Al principio? ¿Al final? ¿En el medio?

Averigualo vos mismo: inspeccioná en la consola qué elementos contiene `pertenencias`, agregale una "ballesta" y volvé a inspeccionar `pertenencias`.

Además existe un procedimiento `remover`, que sólo recibe la lista por parámetro. Investigá en la consola qué hace. ☺

☒ Consola ☒ Biblioteca

```
☒ pertenencias
=> ["espada", "escudo", "antorchas"]
☒ agregar(pertenencias, "ballesta")
=> 4
☒ pertenencias
=> ["espada", "escudo", "antorchas", "ballesta"]
☒ remover(pertenencias)
=> "ballesta"
```



## Trasladar

Bueno, ya hablamos bastante; ¡es hora de la acción 📲!

Declará un procedimiento `trasladar`, que tome dos listas, saque el último elemento de la primera y lo agregue a la segunda.

Ejemplo:

```
let unaLista = [1, 2, 3];
let otraLista = [4, 5];

trasladar(unaLista, otraLista);

unaLista //debería ser [1, 2]
otraLista //debería ser [4, 5, 3]
```

✖ ¡Dame una pista!

✖ Solución ✎ Biblioteca ✎ Consola

```
1 function trasladar(unaLista, otraLista) {
2   return agregar(otraLista, remover(unaLista));
3 }
```

✖ Enviar

✖ ¡Muy bien! Tu solución pasó todas las pruebas

¡Felicitaciones! 🎉

Hasta ahora anduvimos agregando, quitando y consultando longitudes. ¿Qué más podemos hacer con las listas? ¡Seguinos!



## ¿Y dónde está?

Otra cosa que queremos hacer con las listas es saber en qué posición se encuentra un elemento. Para ello utilizamos la función `posicion` de la siguiente manera:

```
posicion(["a", "la", "grande", "le", "puse", "cuca"], "grande"); //devuelve 2  
let diasLaborales = ["lunes", "martes", "miércoles", "jueves", "viernes"]  
posicion(diasLaborales, "lunes"); //devuelve 0
```

Como ves, lo curioso de esta función es que pareciera devolver siempre uno menos de lo esperado. Por ejemplo, la palabra "grande" aparece tercera, no segunda; y "lunes" es el primer día laboral, no el cero. ¿Es que los creadores de JavaScript se equivocaron? 😐

¡No! Se trata de que en JavaScript, al igual que en muchos lenguajes, las posiciones de las listas arrancan en 0: el primer elemento está en la posición 0, el segundo en la 1, el tercero en la 2, y así.

¿Y qué sucede si le pasás por parámetro a `posicion` un elemento que no tiene? ¡Averigualo vos mismo!

Probá lo siguiente:

```
posicion(diasLaborales, "osvaldo")
```

☒ Consola ☒ Biblioteca

```
☒ posicion(diasLaborales, "osvaldo")  
=> -1  
☒
```



## Contiene

¡Ahora te toca a vos!

Escribí la función `contiene` que nos diga si una lista contiene un cierto elemento.

```
☒ contiene([1, 6, 7, 6], 7)
true
☒ contiene([1, 6, 7, 6], 6)
true
☒ contiene([], 7)
false
☒ contiene([8, 5], 7)
false
```

☒ ¡Dame una pista!

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function contiene(lista, numero) {
2   return posicion(lista, numero) >= 0;
3 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Bien hecho!

Si venís prestando atención a los ejemplos de consulta, habrás notado que las listas también pueden tener elementos duplicados: [1, 2, 1], ["hola", "hola"], etc.

Por tanto, `posicion` en realidad devuelve la posición de la *primera aparición* del elemento en la lista. Por ejemplo:

```
☒ posicion(["qué", "es", "eso", "eso", "es", "queso"], "es")
1 //devuelve 1 porque si bien "es" también está en la posición 4, aparece primero en la posición 1.
```



## Enésimo elemento

Así como existe una función para averiguar en qué posición está un elemento, también puede ocurrir que queramos saber lo contrario: qué elemento está en una cierta posición. 😊

Para averiguarlo podemos usar el **operador de indexación**, escribiendo después de la colección y entre corchetes `[]` la posición que queremos para averiguar:

```
mesesDelAnio[0]
"enero"
["ese", "perro", "tiene", "la", "cola", "peluda"][1]
"perro"
```

¡Ojo! El número que le pases, formalmente llamado **índice**, debe ser menor a la longitud de la lista, o cosas malas pueden suceder. 😬

Probalo vos mismo en la consola: ¿qué sucede si le pedís el elemento 0 a una lista vacía? ¿O si le pedís el elemento 48 a una lista de 2 elementos?

Consola   Biblioteca

```
[] [0]
=> undefined
```



## Más premios

Si le pedís un elemento en una posición igual o mayor al tamaño de la lista, vas a obtener `undefined`. No parece algo terrible, pero el problema es que con `undefined` no podés hacer nada realmente útil.

Así que la advertencia es: ¡no te pases de índice!

Teniendo esto en cuenta, va un desafío: escribí nuevamente la función `medallaSegunPuesto`, pero esta vez usando como máximo un único `if`. Quizás las listas te pueden ser útiles acá 😊.

Te recordamos qué hace la función: tiene que devolver la medalla que le corresponde a los primeros puestos de una competencia.

```
☒ medallaSegunPuesto(1)
"oro"
☒ medallaSegunPuesto(2)
"plata"
☒ medallaSegunPuesto(3)
"bronce"
☒ medallaSegunPuesto(4)
"nada"
☒ medallaSegunPuesto(5)
"nada"
```

☒ ¡Dame una pista!

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 let medalla=["oro", "plata", "bronce"]
2 function medallaSegunPuesto(puesto) {
3   if (puesto >= 4) {
4     return "nada"
5   } else {
6     return medalla[puesto - 1]
7   }
8 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## No te olvides de saludar

Vamos a conocer una manera de recorrer los elementos de una lista con un nuevo amigo: el `for`.

Imaginémonos que tenemos una lista con los precios de los productos que compramos en el supermercado y queremos restar cada uno de ellos a `plataEnBilletera`. Usando `for` podemos hacerlo así:

```
for(let precio of [10, 100, 87]) {
  plataEnBilletera = plataEnBilletera - precio
}
```

donde `plataEnBilletera` es una variable que se va modificando a medida que recorremos los `precios`.

Si teníamos \$500 en nuestra billetera, después del `for` nos van a quedar \$303 porque:

- Al principio `plataEnBilletera` era 500 y el primer `precio` de la lista es 10. Luego de hacer  $500 - 10$ , `plataEnBilletera` es 490.
- A los 490 que quedaron en nuestra billetera, le restamos el segundo `precio` de la lista: 100. Ahora `plataEnBilletera` es 390.
- El último `precio` a restar es 87, por lo que, al hacer  $390 - 87$ , la variable `plataEnBilletera` terminará siendo 303.

Completá la función `saludar` que recibe una lista de personas e imprime un saludo para cada una de ellas.

```
saludar(["Don Pepito", "Don Jose"])
hola Don Pepito
hola Don Jose

saludar(["Elena", "Hector", "Tita"])
hola Elena
hola Hector
hola Tita
```

¡Dame una pista!

Tené en cuenta que el saludo que imprimimos siempre es "hola" y luego el nombre de la persona. ¡No te olvides del espacio!

Solución Biblioteca Consola

```
1 let personas = ["Cintia", "Matias", "Julieta"]
2
3 function saludar(personas) {
4   for(let persona of personas) {
5     imprimir ("hola" + " " + persona);
6   }
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## Los primeros registros

Una historiadora está recopilando información acerca de distintos monumentos a lo largo y ancho del mundo 🌎. En principio solo quiso saber el nombre, ubicación, y año de construcción de cada monumento. 📈

Para eso almacenó cada dato en una variable:

```
nombreEstatuaDeLaLibertad = "Estatua de la Libertad";
ubicacionEstatuaDeLaLibertad = "Nueva York";
anioDeConstruccionEstatuaDeLaLibertad = "1886";
nombreCristoRedentor = "Cristo Redentor";
ubicacionCristoRedentor = "Rio De Janeiro";
anioDeConstruccionCristoRedentor = "1931";
```

Ahí es cuando se dio cuenta que no era conveniente 😔: si bien la información entre las variables estaba relacionada, la estaba almacenando por separado. Entonces pensó: ¿no existirá alguna forma de representar las distintas características o propiedades de una misma cosa de forma agrupada?

Luego de investigar un poco, encontró una mejor manera para guardar la información de los monumentos. Probá en la consola escribiendo:

```
estatuaDeLaLibertad
cristoRedentor
torreEiffel
tajMahal
coliseo
```

☒ Consola ☒ Biblioteca

```
☒ estatuaDeLaLibertad
=> {nombre:"Estatua de la Libertad",ubicacion:"Nueva York, Estados Unidos de América",anioDeConstruccion:1886}
☒ cristoRedentor
=> {nombre:"Cristo Redentor",ubicacion:"Rio de Janeiro, Brasil",anioDeConstruccion:1931}
☒ torreEiffel
=> {nombre:"Torre Eiffel",ubicacion:"París, Francia",anioDeConstruccion:1889}
☒ tajMahal
=> {nombre:"Taj Mahal",ubicacion:"Agra, India",anioDeConstruccion:1653}
```



## Tu propio monumento

Los monumentos que probaste en el ejercicio anterior están representados como *registros*, y cada una de sus características (nombre, ubicación, año de construcción) son *campos* del registro. Por cierto, ¡podemos crear registros de cualquier cosa, con los campos que querramos!

Por ejemplo, podríamos almacenar un libro de modo que cada campo del registro fuese alguna característica: su título, su autor, su fecha de publicación, y más. 📖

¡Es tu momento del monumento! Guardá en las variables `torreAzadi` y `monumentoNacionalALaBandera` registros de esos monumentos, oriundos de las ciudades de Teherán, Irán y Rosario, Argentina respectivamente. ¿Te animás a investigar en qué año se terminaron de construir para completar ese campo? 😊

☒ ¡Dame una pista!

Quizá te sea útil ver cómo declaramos algún monumento en el ejercicio anterior.

Por ejemplo, esta es la Torre Eiffel:

```
let torreEiffel = { nombre: "Torre Eiffel", locacion: "París, Francia", anioDeConstruccion: 1889 };
```

☒ Solución ☒ Biblioteca ☒ Consola

```
1 let torreAzadi = { nombre: "Torre Azadi", locacion: "Teherán, Irán", anioDeConstruccion: 1971 };
2 let monumentoNacionalALaBandera = { nombre: "Monumento Nacional a la bandera", locacion: "Rosario, Argentina", anioDeConstruccion: 1957 };
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas

¡Buenas habilidades de búsqueda! 🎉 😊 Los registros, al igual que las listas, son una *estructura de datos* porque nos permiten organizar información. Pero ¿en qué se diferencia un registro de una lista?

En las listas podemos guardar muchos elementos de un mismo tipo que representen una misma cosa (por ejemplo todos números, o todos strings). No existen límites para las listas: pueden tener muchos elementos, ¡o ninguno!

En un registro vamos a guardar información relacionada a una única cosa (por ejemplo **un** monumento o **una** persona), pero los tipos de los campos pueden cambiar. Por ejemplo, el nombre y la ubicación de un monumento son strings, pero su año de construcción es un número.



## Accediendo al campo

Cuando consultaste los registros existentes, se veía algo parecido a lo siguiente:

```
✉ tajMahal
{ nombre: "Taj Mahal", locacion: "Agra, India", anioDeConstruccion: 1653 }
```

Esa consulta era porque estábamos viendo al registro `tajMahal` completo, incluyendo todos sus campos. ¡Pero también se puede consultar por un campo particular! Mirá ☺:

```
✉ tajMahal.locacion
"Agra, India"
✉ tajMahal.anioDeConstruccion
1653
```

Declaramos los planetas `mercurio`, `marte` y `saturno` como registros con la siguiente información: `nombre`, `temperaturaPromedio` y si `tieneAnillos`. ¡Probalos en la consola!

✉ Consola ✉ Biblioteca

```
✉ mercurio
=> {nombre:"Mercurio",temperaturaPromedio:67,tieneAnillos:false}
✉ mercurio.temperaturaPromedio
=> 67
✉ marte.nombre
=> "Marte"
✉ saturno.tieneAnillos
=> true
```



## Temperatura de planeta

Ahora que agregamos registros de planetas, ¡trabajemos un poco con ellos! 🚀

Desarrollá una función `temperaturaDePlaneta` que reciba por parámetro un registro de planeta y devuelva un string que indica su nombre y su temperatura promedio. ¡Tiene que funcionar para cualquier planeta! 🌎 Por ejemplo:

```
☒ temperaturaDePlaneta(mercurio)  
"Mercurio tiene una temperatura promedio de 67 grados"  
☒ temperaturaDePlaneta(saturno)  
"Saturno tiene una temperatura promedio de -139 grados"  
☒ temperaturaDePlaneta(venus)  
"Venus tiene una temperatura promedio de 462 grados"
```

☒ ¡Dame una pista!

¡Prestá atención a los strings que devuelven los ejemplos! Sólo la parte correspondiente a cada planeta varía, como el `nombre` y la `temperaturaPromedio`. Además, tenés que dejar espacios entre las palabras que rodean a `nombre` y `temperaturaPromedio`. Mirá...

nombre tiene una temperatura promedio de temperaturaPromedio grados

☒ Solución   ☒ Biblioteca   ☒ Consola

```
1 function temperaturaDePlaneta(planeta) {  
2   return planeta.nombre + " tiene una temperatura promedio de " + planeta.temperaturaPromedio + " grados";  
3 }  
4
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Moviendo archivos

Por el momento estuvimos creando y consultando registros. ¿No sería interesante poder... modificarlos? 😊

La sintaxis para modificar campos de registros es muy similar a lo que hacemos para cambiar los valores de las variables. Por ejemplo, para cambiar la temperatura de un planeta:

```
saturno.temperaturaPromedio = -140;
```

Ahora imaginá que tenemos un registro para representar un archivo, del que sabemos su ruta (dónde está guardado) y su fecha de creación. Si queremos cambiar su ruta podemos hacer...

```
leeme
{ ruta: "C:\leeme.txt", creacion: "23/09/2004" }

moverArchivo(leeme, "C:\documentos\leeme.txt") }
```

Luego el registro `leeme` tendrá modificada su ruta:

```
leeme
{ ruta: "C:\documentos\leeme.txt", creacion: "23/09/2004" }
```

¡Es tu turno! Desarrollá el procedimiento `moverArchivo`, que recibe un registro y una nueva ruta y modifica el archivo con la nueva ruta.

Solución  Biblioteca  Consola

```
function moverArchivo(archivo, nuevaruta) {
  archivo.ruta = nuevaruta
}
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## Registros de dos milenios

En el ejercicio anterior modificamos la ruta del registro, pero no utilizamos su fecha de creación. ¡Usémosla! Queremos saber si un archivo es del milenio pasado, lo que ocurre cuando su año es anterior a 2000. [MÁS]

Desarrollá la función `esDelMilenoPasado`, que recibe un archivo y devuelve un booleano.

```
☒ esDelMilenoPasado({ ruta: "D:\fotonacimiento.jpg", creacion: "14/09/1989" })
true
```

☒ ¡Dame una pista!

Quizá te pueda servir la función `anio`:

```
☒ anio("04/11/1993")
1993
```

Solución  Biblioteca  Consola

```
1 function esDelMilenoPasado(archivo) {
2   return anio(archivo.creacion) < "2000";
3 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Postres complejos

Unos ejercicios atrás te contamos la diferencia entre listas y registros. ¡Pero eso no significa que no podamos usar ambas estructuras a la vez!



Por ejemplo, una lista puede ser el campo de un registro. Mirá estos registros de postres, de los cuales sabemos cuántos minutos de cocción requieren y sus ingredientes:

```
let flanCasero = { ingredientes: ["huevos", "leche", "azúcar", "vainilla"], tiempoDeCoccion: 50 }
let cheesecake = { ingredientes: ["queso crema", "framboresas"], tiempoDeCoccion: 80 }
let lemonPie = { ingredientes: ["jugo de limón", "almidón de maíz", "leche", "huevos"], tiempoDeCoccion: 65 }
```

Creá una función `masDificilDeCocinar`, que recibe dos registros de postres por parámetros y devuelve el que tiene más ingredientes de los dos.

```
masDificilDeCocinar(flanCasero, cheesecake)
{ ingredientes: ["huevos", "leche", "azúcar", "vainilla"], tiempoDeCoccion: 50 }
```

¡Dame una pista!

Solución  Biblioteca  Consola

```
1 function masDificilDeCocinar(postre1, postre2) {
2   if (longitud(postre1.ingredientes) > longitud(postre2.ingredientes)) {
3     return postre1;
4   } else {
5     return postre2;
6   }
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



## Listas de registros

En el ejercicio anterior te mostramos que un registro puede tener una lista entre sus campos. ¿Y al revés? ¿Podemos tener una lista de registros? 🤔

¡Sí! Así como trabajamos con listas de números, booleanos, strings o más listas, también podemos listar registros. Se puede hacer todo lo que hacías antes, como por ejemplo `remover`, saber su longitud o preguntar por el elemento de cierta posición utilizando los corchetes `[]`.

Probá en la consola las listas `postresFavoritos` y `monumentosDeAmerica`. Hay un postre que no mostramos antes, ¿te das cuenta cuál es solamente leyendo sus ingredientes? 😊

☒ Consola ☒ Biblioteca

```
☒ postresFavoritos
=> [{ingredientes:["galletitas","dulce de leche","crema"],tiempoDeCoccion:20},{ingredientes:
["huevos","leche","azúcar","vainilla"],tiempoDeCoccion:50},{ingredientes:["queso crema","framboesas"],tiempoDeCoccion:80},
{ingredientes:["jugo de limón","almidón de maíz","leche","huevos"],tiempoDeCoccion:65}]

☒ monumentosDeAmerica
=> [{nombre:"Monumento Nacional a la Bandera",locacion:"Rosario, Argentina",anioDeConstruccion:1957},{nombre:"Estatua de la
Libertad",locacion:"Nueva York, Estados Unidos de América",anioDeConstruccion:1886},{nombre:"Cristo Redentor",locacion:"Rio de
Janeiro, Brasil",anioDeConstruccion:1931}]
```



## 60 dulces minutos

A veces no sólo queremos comer algo rico, sino que queremos comerlo lo antes posible. 😊 🎂

Desarrollá el procedimiento `agregarAPostresRapidos`, que recibe una lista con postres rápidos y un postre por parámetro. Si el tiempo de cocción es de una hora o menos, se agrega el registro a la lista.

☒ ¡Dame una pista!

¡Recordá que `tiempoDeCoccion` está expresado en minutos! Por lo tanto, si queremos que sea cocine en una hora o menos, tenés que fijarte que ese `tiempoDeCoccion` sea menor a 60 minutos. 😊

Además, como `agregarAPostresRapidos` es un procedimiento, no tiene que devolver nada. Sólo tenés que `agregar` (¿te acordás de este procedimiento?) el postre a la lista si es rápido.

☒ Solución ☒ Biblioteca ☒ Consola

```
1 function agregarAPostresRapidos(postresRapidos, postre) {  
2   if (postre.tiempoDeCoccion <= 60) {  
3     agregar(postresRapidos, postre);  
4   }  
5 }
```

☒ Enviar

☒ ¡Muy bien! Tu solución pasó todas las pruebas



## Hay un registro en mi registro

¿Te acordás cuando vimos que una lista podía estar compuesta por otras listas? ¡Con los registros aplica la misma idea! 🤔 Si tenemos alguna estructura de datos compleja, puede ocurrir que no alcance con representarla únicamente mediante strings, números, booleanos y listas, sino que necesitemos *otro* registro dentro.

¡No se puede vivir a base de postres! Bueno, quizás sí, pero mantengamos una alimentación saludable 🍗. Mediante un registro queremos modelar un menú completo: consiste en un plato principal 🍜, los vegetales de la ensalada que acompaña 🥗, y un postre 🍰 como lo veníamos trabajando, es decir, sigue siendo un registro.

Por ejemplo, el siguiente es un menú con bife de lomo como plato principal, una ensalada de papa, zanahoria y arvejas como acompañamiento y un cheesecake de postre. Como el registro es un poco extenso, y para que sea más legible, lo vamos a escribir de la siguiente forma:

```
let menuDelDia = {  
  platoPrincipal: "bife de lomo",  
  ensalada: ["papa", "zanahoria", "arvejas"],  
  postre: { ingredientes: ["queso crema", "framboresas"], tiempoDeCoccion: 80 }  
};
```

Averiguá qué devuelve el campo `ingredientes` del campo `postre` del registro `menuInfantil`. ¡Está un registro adentro del otro! La sintaxis es la siguiente:

```
menuInfantil.postre.ingredientes
```

☒ Consola ☒ Biblioteca

```
☒ menuInfantil.postre.ingredientes  
=> ["galletitas", "dulce de leche", "crema"]  
☒
```



## ¡Azúcar!

Para terminar, trabajemos una vez más con los menús.

Definí un procedimiento `endulzarMenu`, que recibe un registro menú y le agrega azúcar a los ingredientes de su postre. Si ya tiene azúcar, no importa... ¡le agrega más! 😊

[☰ ¡Dame una pista!](#)

Recordá que cada menú tiene un `postre` y que cada postre tiene `ingredientes`. 🍪

[☰ Solución](#) [☰ Biblioteca](#) [☰ Consola](#)

```
1 function endulzarMenu(menuDelDia) {  
2   agregar(menuDelDia.postre.ingredientes, "azúcar")  
3 }
```

[☰ Enviar](#)

[☰ ¡Muy bien! Tu solución pasó todas las pruebas](#)



# Las ganancias semestrales

• (/argentina-practica-de-listas-y-registros-y-el-resto-de-registros-todas-las-ganancias) (/argentina-practica-de-listas-y-registros-nos-visita-un-viejo-amigo) (/argentina-practica-de-listas-y-registros-cuentas-claras) (/argentina-practica-de-listas-y-registros-la-ganancia-promedio)

```
//En julio ganó $50, en agosto perdió -$12, etc
let balancesUltimoSemestre = [
    { mes: "julio", ganancia: 50 },
    { mes: "agosto", ganancia: -12 },
    { mes: "septiembre", ganancia: 1000 },
    { mes: "octubre", ganancia: 300 },
    { mes: "noviembre", ganancia: 200 },
    { mes: "diciembre", ganancia: 0 }
];
```

Y nos acaba de preguntar: "¿puedo saber la ganancia de todo un semestre?"

"Obvio, solo tenemos que sumar las ganancias de todos los balances", dijimos, y escribimos el siguiente código:

```
function gananciaSemestre(balances) {
    return balances[0].ganancia + balances[1].ganancia +
        balances[2].ganancia + balances[3].ganancia +
        balances[4].ganancia + balances[5].ganancia;
}
```

"Gracias ☺", nos dijo Ana, y se fue calcular las ganancias usando la función que le pasamos. Pero un rato más tarde, volvió contándonos que también había registrado los balances del primer trimestre de este año:

```
//En enero la empresa ganó $80, en febrero, $453, en marzo $1000
let balancesPrimerTrimestre = [
    { mes: "enero", ganancia: 80 },
    { mes: "febrero", ganancia: 453 },
    { mes: "marzo", ganancia: 1000 }
];
```

Y nos preguntó: "¿Podría usar esta función que me dieron para calcular las ganancias del primer trimestre?". ☺

¿Tiene algún problema la función gananciaSemestre que escribimos anteriormente? ¿Funcionará con los balances trimestrales? ¿Y con los cuatrimestrales?

¡Probala en la consola!

>\_Consola </> Biblioteca

```
↳ gananciaSemestre
=> <function>
```



# ¿Y el resto de las ganancias?

- (/argentina- • (/argentina- (/argentina- (/argentina- (/argentina- (/argentina-  
programacion-imperativa- programacion-imperativa- programacion-imperativa- programacion-imperativa- programacion-imperativa- programacion-imperativa- progra  
1. Es muy repetitiva y tediosa de escribir. ¡Tenemos que hacer muchas sumas a mano!  
practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- pr  
2. No es genérica, como bien dice su nombre, sólo sirve para sumar las ganancias de 6 balances:  
registros-las-ganancias- registros-todas-las- registros-nos-visita-un- registros-cuentas-claras) registros-la-ganancia- regis  
• si tiene más de seis balances, sólo suma los primeros (yo-amigo) promedio)  
• si tiene menos, no funciona (¿te acordás cuando te dijimos que si te ibas de índice cosas malas podían ocurrir? 😊)

Lo que nos gustaría es poder sumar las ganancias de todos los balances de una lista, sin importar cuántos haya realmente; queremos una función `gananciaTotal`, que pueda sumar balances de cualquier período de meses: semestres, cuatrimestres, trimestres, etc. ¡Qué difícil!

¡Relajá! Ya tenemos nuestra versión; probala con las siguientes consultas:

```
↳ gananciaTotal([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 3 }
])
↳ gananciaTotal([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 3 },
  { mes: "marzo", ganancia: 1 },
  { mes: "abril", ganancia: 8 },
  { mes: "mayo", ganancia: 8 },
  { mes: "junio", ganancia: -1 }
])
↳ gananciaTotal([])
```

Después seguinos para contarte cómo la hicimos. ☺

```
↳ gananciaTotal([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 3 }
])
=> 5
↳ gananciaTotal([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 3 },
  { mes: "marzo", ganancia: 1 },
  { mes: "abril", ganancia: 8 },
  { mes: "mayo", ganancia: 8 },
  { mes: "junio", ganancia: -1 }
])
=> 19
```



# Todas las ganancias, la ganancia

¿Y si tuviera exactamente 1 elemento? Sería... 0.... ¿más ese elemento? ¡Exacto! 😊

```
function gananciaTotal1(balancesDeUnPeriodo) {  
    let sumatoria = 0;  
    sumatoria = sumatoria + balancesDeUnPeriodo[0].ganancia;  
    return sumatoria;  
}
```

¿Y si tuviera 2 elementos? ☺

```
function gananciaTotal2(balancesDeUnPeriodo) {  
    let sumatoria = 0;  
    sumatoria = sumatoria + balancesDeUnPeriodo[0].ganancia;  
    sumatoria = sumatoria + balancesDeUnPeriodo[1].ganancia;  
    return sumatoria;  
}
```

¿Y si tuviera 3 elementos? ☺

```
function gananciaTotal3(balancesDeUnPeriodo) {  
    let sumatoria = 0;  
    sumatoria = sumatoria + balancesDeUnPeriodo[0].ganancia;  
    sumatoria = sumatoria + balancesDeUnPeriodo[1].ganancia;  
    sumatoria = sumatoria + balancesDeUnPeriodo[2].ganancia;  
    return sumatoria;  
}
```

¿Empezás a ver un patrón? Tratá de escribir `gananciaTotal14` que funcione para 4 elementos.

 Solución > Consola

```
1 function gananciaTotal4(balancesDeUnPeriodo) {  
2     let sumatoria = 0;  
3     sumatoria = sumatoria + balancesDeUnPeriodo[0].ganancia;  
4     sumatoria = sumatoria + balancesDeUnPeriodo[1].ganancia;  
5     sumatoria = sumatoria + balancesDeUnPeriodo[2].ganancia;  
6     sumatoria = sumatoria + balancesDeUnPeriodo[3].ganancia;  
7     return sumatoria;  
8 }
```



## ✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Bien hecho! 🎉

¿Y si la lista tuviera *cualquier* cantidad de elementos? Si seguimos repitiendo este patrón, veremos que una sumatoria de una lista siempre arranca igual, con `let sumatoria = 0`, y termina igual, devolviendo la variable local `sumatoria` (`return sumatoria`).

```
function gananciaTotalN(unPeriodo) {  
  let sumatoria = 0; // esto siempre está  
  //... etc  
  return sumatoria; //esto siempre está  
}
```

Lo que cambia son las acumulaciones (`sumatoria = sumatoria + ...`); necesitamos una por cada elemento de la lista. Dicho de otra forma, tenemos que *visitar* cada elemento del mismo, sin importar cuántos tenga. Pero, ¿cómo hacerlo? ¿No te suena conocida esta idea de *repetir algo muchas veces?* 🔄

Esta guía fue desarrollada por Gustavo Trucco, Franco Bulgarelli, Felipe Calvo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





# Nos visita un viejo amigo

```
(/argentina-          (/argentina-          (/argentina-      •      (/argentina-          (/argentina-
programas/exercis.../7270/programas/exercis.../7281/programas/exercis.../7282/programas/exercis.../7284/programas/exercis.../7285/programa
(practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- pr
practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- pr
funciones-gananciaTotal(balancesDeUnPeriodo)
registros-todas-las- registros-todas-las- registros-cuentas-claras) registros-la-ganancia- regis
registros-todas-las- registros-todas-las- registros-cuentas-claras) registros-la-ganancia- regis
let sumatoria = 0;
for(several balance of balancesDeUnPeriodo) {
    sumatoria = sumatoria + balance.ganancia;
}
return sumatoria;
}
```

Como ves, el `for...of` nos permite visitar y hacer algo con cada elemento de una lista; en este caso, estaremos visitando cada `balance` de `balancesDeUnPeriodo`.

¿Aún no te convenciste? Nuevamente, probá las siguientes expresiones en la consola:

```
↳ gananciaTotal([])
↳ gananciaTotal([
  { mes: "noviembre", ganancia: 5 }
])
↳ gananciaTotal([
  { mes: "marzo", ganancia: 8 },
  { mes: "agosto", ganancia: 10 }
])
↳ gananciaTotal([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 10 },
  { mes: "marzo", ganancia: -20 }
])
↳ gananciaTotal([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 10 },
  { mes: "marzo", ganancia: -20 },
  { mes: "abril", ganancia: 0 },
  { mes: "mayo", ganancia: 10 }
])
```

>\_Consola </> Biblioteca

```
↳ gananciaTotal([])
=> 0
↳ gananciaTotal([
  { mes: "noviembre", ganancia: 5 }
])
=> 5
↳ gananciaTotal([
  { mes: "marzo", ganancia: 8 },
```



# Cuentas claras

(/argentina-practica-de-listas-y-registros-turisticos-en-los-semestrales)      (/argentina-las-ganancias)      (/argentina-ganancias-la-ganancia)      (/argentina-viejo-amigo)      •      (/argentina-promedio)  
? ¡Dame una pista!

Lo importante en este ejercicio es pensar cual es el valor inicial de cantidad y cuando incrementa ese valor.

 Solución >\_Consola

```
1 function cantidadDeBalancesPositivos(balancesDeUnPeriodo) {  
2     let cantidad = 0;  
3     for (let balance of balancesDeUnPeriodo) {  
4         if (balance.ganancia > 0) {  
5             cantidad = cantidad + 1;  
6         }  
7     }  
8     return cantidad;  
9 }  
10
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

cantidad es lo que en programación se conoce como *contador*, una variable que se incrementa cada vez que hacemos algo dentro de un `for...of` o solo aquellas veces que se cumpla una condición (como en este caso 😊).



# La ganancia promedio

```
(/argentina-          (/argentina-          (/argentina-          (/argentina-          (/argentina-          (/argentina-
Programas de lenguajes/7259/proyecto-de-avisos/7260/programacion-exercis.../7282/programacion-exercis.../7283/programacion-exercis.../7284/programa-
prolog/la-promedio-dos@liva- programacion-imperativa- programacion-imperativa- programacion-imperativa- programacion-imperativa- progra-
practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- pr
registros-ganancia-un-registro-y-una-lista-de-balances-registros-todas-las-promedios-registros-nos-visita-un- registros-cuentas-claras) regis
semestrales) las-ganancias) ganancias-la-ganancia) viejo-amigo)
  ↳ gananciaPromedio([
    { mes: "marzo", ganancia: 8 },
    { mes: "agosto", ganancia: 10 }
  ])
  9
```

💡 ¡Dame una pista!

¿Qué es un promedio? 😊 Veámoslo con un ejemplo:

En un grupo de 3 personas, Felipe de 3, Delfina de 8 y Eli de 10 años, la edad promedio es de 7 años. Porque  $3 + 8 + 10$  es igual a 21, dividido la cantidad de gente en el grupo (3 personas) da 7.

Para esto contás con las funciones `gananciaTotal` y `longitud`. 😊

Solución

Consola

```
1 function gananciaPromedio(balances) {
2   return gananciaTotal(balances) / longitud(balances)
3 }
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Perfecto! 😊 Vamos a complicar un poco las cosas. 😺



# Quién gana, quién pierde

(/argentina-registro-exercism/7279-practica-de-listas-y-registros-las-ganancias-semestrales) (/argentina-registro-exercism/7281-practica-de-listas-y-registros-y-el-resto-de-gananciaPositiva, qué es la suma de las ganancias de los balances positivos semestrales) (/argentina-registro-exercism/7282-practica-de-listas-y-registros-todas-las-ganancias-la-ganancia) (/argentina-registro-exercism/7283-practica-de-listas-y-registros-nos-visita-un-viejo-amigo) (/argentina-registro-exercism/7284-practica-de-listas-y-registros-cuentas-claras) (/argentina-registro-exercism/7285-practica-de-listas-y-registros-nos-visita-un-viejo-amigo) (/argentina-registro-exercism/7286-practica-de-listas-y-registros-cuentas-claras)

💡 [Dame una pista!](#)

gananciaPositiva es muy similar a cantidadDeBalancesPositivos , solo que ahora necesitas una sumatoria en vez de un contador. 😊

Solución > Consola

```
1 function gananciaPositiva(balancesDeUnPeriodo) {  
2   let sumatoria = 0;  
3   for (let balance of balancesDeUnPeriodo) {  
4     if(balance.ganancia > 0){  
5       sumatoria = sumatoria + balance.ganancia;  
6     }  
7   }  
8   return sumatoria;  
9 }  
10  
11 function promedioGananciasPositivas(balancesDeUnPeriodo) {  
12   return gananciaPositiva(balancesDeUnPeriodo) / cantidadDeBalancesPositivos(balancesDeUnPeriodo);  
13 }
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Como podés ver todos los promedios se basan en el mismo principio 🤓. Sumar una cantidad determinada elementos y dividir el resultado por esa cantidad. Si quisieramos realizar una función promedio genérica sería algo así:

```
function promedio(listaDeNumeros) {  
  return sumatoria(listaDeNumeros) / longitud(listaDeNumeros);  
}  
  
function sumatoria(listaDeNumeros) {  
  let sumatoria = 0;  
  for (let numero of listaDeNumeros) {  
    sumatoria = sumatoria + numero;  
  }  
  return sumatoria;  
}
```

¡Pero nosotros no tenemos una lista de números sino de registros! 🤔 ¿Y entonces? 🤔



# Soy el mapa, soy el mapa

```
(/argentina-          (/argentina-          (/argentina-          (/argentina-          (/argentina-          (/argentina-
programas/exercis.../7259/programas/exercis.../7261/programas/exercis.../7262/programas/exercis.../7263/programas/exercis.../7264/program
ganacias-en-la-practica. Programa para practicar la programacion imperativa en ordenacion-imperativa- programacion-imperativa- program
practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- pract
registros-de-ganancias-registros-de-listas-de-balances-registros-de-listas-que-mostramos-las-ganancias-de-los-registros-de-cuentas-claras) regist
semestrales)      las-ganancias)   ganancias-la-ganancia)  viejo-amigo)
  ↳ ganancias([
    { mes: "enero", ganancia: 40 },
    { mes: "febrero", ganancia: 12 },
    { mes: "marzo", ganancia: 8 }
  ])
  [40, 12, 8]
```

💡 ¡Dame una pista!

Solución  Consola

```
1 function ganancias(balancesDeUnPeriodo) {
2   let ganancias = [];
3   for (let balance of balancesDeUnPeriodo) {
4     agregar(ganancias, balance.ganancia)
5   }
6   return ganancias;
7 }
8
9 /*function promedio(listaDeNumeros) {
10   return sumatoria(listaDeNumeros) / longitud(listaDeNumeros);
11 }
12
13 function sumatoria(listaDeNumeros) {
14   let sumatoria = 0;
15   for (let numero of listaDeNumeros) {
16     sumatoria = sumatoria + numero;
17   }
18   return sumatoria;
19 }
20 */
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Excelente! 🎉 Ahora ya sabemos cómo transformar cada elemento de una lista para obtener una lista nueva 🎉. De esta manera podemos usar `promedio` con nuestra lista de balances. Pero, ¿se puede utilizar la función `promedio` solo para los balances positivos? 🤔



A filtrar, a filtrar cada cosa en su lugar

 ¡Dame una pista!

`balances` es una lista que contiene justamente eso, `balances` . Pero no todos, tienen que cumplir una condición.

 Solución | >\_Consola

```
1 function balancesPositivos(balancesDeUnPeriodo) {  
2     let balances = [];  
3     for (let balance of balancesDeUnPeriodo) {  
4         if (balance.ganancia > 0) {  
5             agregar(balances, balance);  
6         }  
7     }  
8     return balances;  
9 }
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Muy bien! 🎉 Ahora ya sabemos cómo filtrar una lista. En criollo, aprendimos a obtener los elementos de una lista que cumplen una condición determinada. En este caso obtuvimos una nueva lista con los balances que presentaban una ganancia positiva. 💰



# Un promedio más positivo

 ¡Dame una pista!

 Solución  Biblioteca  Consola

```
1 function gananciasDeBalancesPositivos(balancesDeUnPeriodo) {  
2   return ganancias(balancesPositivos(balancesDeUnPeriodo)) ;  
3 }  
4  
5 function promedioDeBalancesPositivos(balancesDeUnPeriodo) {  
6   return promedio(gananciasDeBalancesPositivos(balancesDeUnPeriodo));  
7 }
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas



# Esto es lo máximo

Usando esta nueva función, definí la función `maximaGanancia` que nos diga cuál es la ganancia más alta entre los balances de un período de tiempo.

```
↳ maximaGanancia([
    { mes: "enero", ganancia: 87 },
    { mes: "febrero", ganancia: 12 },
    { mes: "marzo", ganancia: 8 }
])
87
```

 Dame una pista!

Podés usar la función ganancias que hiciste antes. ☺

Solución Biblioteca Consola

```
1 function maximaGanancia(balancesDeUnPeriodo) {  
2     return maximo(ganancias(balancesDeUnPeriodo));  
3 }
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Si hay una función para calcular el máximo de una lista también hay una para calcular el mínimo. ¿Te imaginás como se llama? ☺



## Como mínimo

 Dame una pista!

Para poder hacer esta nueva función probablemente te sirva tanto `minimo` como `gananciasDeBalancesPositivos`. ☺

minimo se utiliza como maximo:

```
↳ minimo([5, 8, 10, 42, 87, 776])  
5
```

Solución >\_Consola

```
1 function minimaGananciaPositiva(balancesDeUnPeriodo) {  
2     return minimo(ganancias(balancesDeUnPeriodo));  
3 }
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

¡Muy bien! 🎉 Solo queda un ejercicio por delante. 🎉



# Los mejores meses del año

Para eso vamos a hacer las siguientes funciones:

```
practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y-  
registros-siguiendo-una-regla-registros-registro-de-que-devuelve-un-listado-de-datos; registros-nos-visita-un- registros-cuentas-claras) registros-  
• afortunados) que filtra aquellos registros que tuvieron una ganancia mayor a $1000 (amigo)  
• mesesAfortunados , devuelve aquellos meses que fueron afortunados.
```

```
↳ meses([
  { mes: "enero", ganancia: 870 },
  { mes: "febrero", ganancia: 1000 },
  { mes: "marzo", ganancia: 1020 },
  { mes: "abril", ganancia: 2300 },
  { mes: "mayo", ganancia: -10 }
])
["enero", "febrero", "marzo", "abril", "mayo"]

↳ afortunados([
  { mes: "enero", ganancia: 870 },
  { mes: "febrero", ganancia: 1000 },
  { mes: "marzo", ganancia: 1020 },
  { mes: "abril", ganancia: 2300 },
  { mes: "mayo", ganancia: -10 }
])
[ { mes: "marzo", ganancia: 1020 }, { mes: "abril", ganancia: 2300 }]

↳ mesesAfortunados([
  { mes: "enero", ganancia: 870 },
  { mes: "febrero", ganancia: 1000 },
  { mes: "marzo", ganancia: 1020 },
  { mes: "abril", ganancia: 2300 },
  { mes: "mayo", ganancia: -10 }
])
["marzo", "abril"]
```

Definí las funciones `meses` , `afortunados` , `mesesAfortunados` .

 ¡Dame una pista!

Solución Biblioteca Consola

```
1 function meses(balancesDeUnPeriodo) {  
2     let todosLosMeses = [];  
3     for (let balance of balancesDeUnPeriodo) {  
4         agregar(todosLosMeses, balance.mes);  
5     }  
6     return todosLosMeses;  
7 }  
8  
9 function afortunados(balancesDeUnPeriodo) {  
10    let mejoresMeses = [];  
11    for (let balance of balancesDeUnPeriodo)  
12        if (balance.ganancia > 1000) {  
13            agregar(mejoresMeses, balance);  
14        }  
15    }  
16    return mejoresMeses;  
17 }  
18  
19 function mesesAfortunados(balancesDeUnPeriodo) {
```



# Publicaciones muy especiales

(/argentina-  
programas/exclases/72721/proyectos-excepcion/12101/programas-excepcion/75201/programas-aplicaciones/7103-solucion-publicaciones/7284/program  
proyectos-imperativa- programacion-imperativa- programacion-imperativa- programacion-imperativa- programacion-imperativa- program  
practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- practica-de-listas-y- pract  
{ apodo: "foli", texto: "tengo sueño"}, registros-y-el-resto-de- registros-todas-las- registros-nos-visita-un- registros-cuentas-claras) regist  
semestrales) las-ganancias) ganancias-la-ganancia) viejo-amigo)  
Las publicaciones por sí mismas no son muy interesantes, por eso en general contamos con hilos, que no son más que listas de publicaciones:

```
let hiloDeEjemplo = [  
  { apodo: "jor", mensaje: "Aguante la ciencia ficción" },  
  { apodo: "tere", mensaje: "A mi no me gusta mucho" },  
  { apodo: "jor", mensaje: "¡Lee fundación!" }  
]
```

¿Y qué tienen de especiales estas publicaciones? 😊 Aunque como siempre podés enviar tu solución las veces que quieras, no la vamos a evaluar automáticamente por lo que el ejercicio quedará en color celeste. 😊 Si querés verla en funcionamiento, ¡te invitamos a que la pruebes en la consola!

Como último desafío, definí una función `publicacionesCortasDelHilo` que tome un apodo por parámetro y un hilo, y retorne las publicaciones de esa persona que tengan menos de 20 caracteres. Por ejemplo:

```
↳ publicacionesCortasDelHilo("jor", hiloDeEjemplo)  
=> [{ apodo: "jor", mensaje: "¡Lee fundación!" }]
```

Solución  Biblioteca  Consola

```
1 let hiloDeEjemplo = [  
2   { apodo: "jor", mensaje: "Aguante la ciencia ficción" },  
3   { apodo: "tere", mensaje: "A mi no me gusta mucho" },  
4   { apodo: "jor", mensaje: "¡Lee fundación!" }  
5 ]  
6  
7 function publicacionesCortasDelHilo(apodo, hilo) {  
8   let publicacionesCortas = [];  
9   for (let comentario of hilo) {  
10     if (comentario.apodo === apodo && longitud(comentario.mensaje) < 20) {  
11       agregar(publicacionesCortas, comentario)  
12     }  
13   }  
14   return publicacionesCortas;  
15 }
```

▶ Enviar

¡Gracias por enviar tu solución!

# Biblioteca

```
/**/  
  
function longitud(unString) /* ... */  
// Retorna cuan largo es un string  
//  
// Por ejemplo:  
//  
// ✘ longitud("hola")  
// 4  
  
function convertirEnMayuscula(unString) /* ... */  
// Convierte una palabra en mayúsculas  
//  
// Por ejemplo:  
//  
// ✘ convertirEnMayuscula("hola")  
// "HOLA"  
  
function comienzaCon(unString, otroString) /* ... */  
// Retorna un booleano que nos dice si unString empieza con otroString  
//  
// Por ejemplo:  
//  
// ✘ comienzaCon("hola todo el mundo", "hola todo")  
// true  
  
/**/  
  
/**/  
  
function imprimir(unString) /* ... */  
// Imprime por pantalla unString  
//  
// Por ejemplo:  
//  
// ✘ imprimir("¡estoy imprimiendo!")  
// ¡estoy imprimiendo!  
  
function tirarDado() /* ... */  
// Retorna un número al azar entre 1 y 6  
//  
// Por ejemplo:  
//  
// ✘ tirarDado()  
// 5  
// ✘ tirarDado()  
// 1  
// ✘ tirarDado()  
// 2  
  
/**/  
  
function listasIguales(unArray, otroArray) /* ... */  
// Retorna un booleano que nos dice si dos listas son iguales  
//  
// Por ejemplo:  
//  
// ✘ listasIguales([1, 2, 3], [1, 2, 3])  
// true  
// ✘ listasIguales([1, 2, 3], [4, 5, 3])  
// false  
  
function longitud(unStringOLista) /* ... */  
// Retorna el largo de un string o una lista  
//  
// Por ejemplo:  
//  
// ✘ longitud("hola")  
// 4  
// ✘ longitud([5, 6, 3])  
// 3  
  
function agregar(unArrayLista, unElemento) /* ... */  
// Inserta unElemento al final de unArrayLista.  
// Este es un procedimiento que no retorna nada pero modifica a unArrayLista:  
//
```

```

// ✘ let cancionesFavoritas = ["La colina de la vida", "Zamba por vos"]
// ✘ agregar(cancionesFavoritas, "Seminare")
// ✘ cancionesFavoritas
// [ "La colina de la vida", "Zamba por vos", "Seminare"]

function remover(unalista) /* ... */
// Quita el último elemento de unaLista y lo retorna.
//
// ✘ let listaDeCompras = ["leche", "pan", "arroz", "aceite", "yerba"]
// ✘ remover(listaDeCompras)
// "yerba"
// ✘ listaDeCompras
// [ "leche", "pan", "arroz", "aceite"]

function posicion(unalista, unElemento) /* ... */
// Retorna la posición se encuentra un elemento.
// Si el elemento no está en la lista, retorna -1
//
// ✘ let premios = ["dani", "agus", "juli", "fran"]
// ✘ posicion(premios, "dani")
// 0
// ✘ posicion(premios, "juli")
// 2
// ✘ posicion(premios, "feli")
// -1

/**/

```



# Apéndice

## Referencia rápida del lenguaje JavaScript

El lenguaje JavaScript es utilizado ampliamente para construir software en todo el mundo, siendo una de las principales tecnologías de la Web. En Mumuki sólo usamos una muy pequeña parte del mismo, que listamos a continuación:

### Declaración de Funciones

A partir de la [Lección 1: Funciones y tipos de datos](#) ([../..//guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos](#))

Las funciones en JavaScript se declaran mediante la *palabra clave* `function`, y su cuerpo va entre llaves `{ y }`:

```
function nombreDeLaFuncion(parametro1, parametro2, parametro3) {  
    return ...;  
}
```

Toda función debe tener al menos un retorno, que se expresa mediante `return`.

### Operadores matemáticos

A partir de la [Lección 1: Funciones y tipos de datos](#) ([../..//guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos](#))

```
4 + 5  
10 - 5  
8 * 9  
10 / 5
```

### Operadores lógicos

A partir de la [Lección 1: Funciones y tipos de datos](#) ([../..//guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos](#))

```
true && false  
true || false  
! false
```

### Comparaciones

A partir de la [Lección 1: Funciones y tipos de datos](#) ([../..//guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos](#))

```
// para cualquier tipo de dato  
"hola" === "hola"  
"hola" !== "chau"  
  
// para números  
4 >= 5  
4 > 5  
4 <= 5  
4 < 5
```

## Alternativa Condicional

A partir de la [Lección 1: Funciones y tipos de datos](#) (./../guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos)

Los `if`s en JavaScript encierran la condición entre paréntesis y su cuerpo entre llaves:

```
if (hayPersonasEnEspera()) {  
  llamarSiguientePersona();  
}
```

Además, los `if`s pueden opcionalmente tener un `else`:

```
if (hayPersonasEnEspera()) {  
  llamarSiguientePersona();  
} else {  
  esperarSiguientePersona();  
}
```

Por último, podemos combinar varios `if`s para tomar decisiones ante múltiples condiciones:

```
if (hayPersonasEnEspera()) {  
  llamarSiguientePersona();  
} else if (elPuestoDebeSeguirAbierto()) {  
  esperarSiguientePersona();  
} else {  
  cerrarPuesto();  
}
```

## Variables

A partir de la [Lección 3: Variables y procedimientos](#) (./../guides/fbulgarelli/mumuki-guia-fundamentos-javascript-variables-y-procedimientos)

Las variables nos permiten *recordar* valores y se declaran mediante la palabra reservada `let` y se les da un valor inicial usando `=`:

```
let pesosEnMiBilletera = 100;  
let diasQueFaltanParaElVerano = 10;
```

La mismas se asignan mediante `=`:

```
pesosEnMiBilletera = 65;  
diasQueFaltanParaElVerano = 7;
```

En ocasiones las asignaremos usando el valor anterior:

```
pesosEnMiBilletera = pesosEnMiBilletera * 2;
diasQueFaltanParaElVerano = diasQueFaltanParaElVerano - 1;
```

La asignación anterior se puede compactar combinando el signo `=` y la operación:

```
pesosEnMiBilletera *= 2;
diasQueFaltanParaElVerano -= 1;
```

## Repetición indexada

A partir de la [Lección 7: Recorridos](#) ([..../guides/mumukiproject/mumuki-guia-javascript-practica-de-listas-y-registros](#))

Las listas pueden ser *recorridas*, visitando y haciendo algo con cada uno de sus elementos. Para ello contamos con la estructura de control `for...of`, que encierra su generador entre paréntesis (`( y )`) y su cuerpo entre llaves (`{ y }`):

```
let patrimoniosDeLaHumanidad = [
  {declarado: 1979, nombre: "Parque nacional Tikal", pais: "Guatemala"},
  {declarado: 1983, nombre: "Santuario histórico de Machu Picchu", pais: "Perú"},
  {declarado: 1986, nombre: "Parque nacional do Iguaçu", pais: "Brasil"},
  {declarado: 1995, nombre: "Parque nacional de Rapa Nui", pais: "Chile"},
  {declarado: 2003, nombre: "Quebrada de Humahuaca", pais: "Argentina"}
]

let cantidadPatrimoniosDeclaradosEnEsteSiglo = 0;
for (let patrimonio of patrimoniosDeLaHumanidad) {
  if (patrimonio.declarado >= 2000) {
    cantidadPatrimoniosDeclaradosEnEsteSiglo += 1;
  }
}
```

## Biblioteca simplificada

Dentro de Mumuki usamos una biblioteca de funciones inspirada en la que ya viene con JavaScript, pero simplificada para que sea más sencilla y segura de usar. A continuación listamos las principales funciones que se pueden usar, indicando el equivalente *real* en JavaScript cuando corresponda.

### longitud(unString)

A partir de la [Lección 1: Funciones y tipos de datos](#) ([..../guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos](#))

Versión simplificada de `length`

([https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/String/length](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/String/length))

Uso:

```
longitud("holá")
4
```

### convertirEnMayuscula(unString)

A partir de la [Lección 1: Funciones y tipos de datos](#) ([..../guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos](#))

Versión simplificada de `toUpperCase`

([https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/String/toUpperCase](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/String/toUpperCase))

Convierte un `unString` en mayúsculas:

```
↳ convertirEnMayuscula("holá")
"HOLA"
```

## comienzaCon(unString, otroString)

A partir de la [Lección 1: Funciones y tipos de datos](#) ([..../guides/fbulgarelli/fundamentos-javascript-funciones-tipos-de-datos](#))

Versión simplificada de `startsWith`

([https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/String/startsWith](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/String/startsWith))

Dice si `unString` empieza con `otroString`:

```
↳ comienzaCon("aprendiendo a programar", "aprendiendo")
true

↳ comienzaCon("aprendiendo a programar", "aprend")
true

↳ comienzaCon("aprendiendo a programar", "programar")
false

↳ comienzaCon("aprendiendo a programar", "tomar el té")
false
```

## imprimir(unString)

A partir de la [Lección 3: Variables y procedimientos](#) ([..../guides/fbulgarelli/mumuki-guia-fundamentos-javascript-variables-y-procedimientos](#))

Versión simplificada de `console.log` (<https://developer.mozilla.org/es/docs/Web/API/Console/log>)

Imprime por pantalla `unString`:

```
↳ imprimir("¡estoy imprimiendo!")
¡estoy imprimiendo!
```

## tirarDado()

A partir de la [Lección 3: Variables y procedimientos](#) ([..../guides/fbulgarelli/mumuki-guia-fundamentos-javascript-variables-y-procedimientos](#))

Devuelve al azar un número entre 1 y 6:

```
↳ tirarDado()  
5  
↳ tirarDado()  
1  
↳ tirarDado()  
2
```

## listasIguales

A partir de la [Lección 5: Listas](#) ([../../guides/fbulgarelli/mumuki-guia-fundamentos-javascript-vectores](#))

```
↳ listasIguales(unLista, otraLista)
```

## longitud(unLista)

A partir de la [Lección 5: Listas](#) ([../../guides/fbulgarelli/mumuki-guia-fundamentos-javascript-vectores](#))

- `length` de listas

([https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Array/length](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/length))

Nos dice cuan largo es `unLista`:

```
↳ longitud([true, false, false, true])  
4  
↳ longitud([5, 6, 3])  
3
```

## agregar(unLista, unElemento)

A partir de la [Lección 5: Listas](#) ([../../guides/fbulgarelli/mumuki-guia-fundamentos-javascript-vectores](#))

Versión simplificada de `push`

([https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Array/push](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/push))

Inserta `unElemento` al final de `unLista`. Este es un procedimiento que no devuelve nada pero modifica a `unLista`:

```
↳ let cancionesFavoritas = ["La colina de la vida", "Zamba por vos"]  
// agrega el elemento "Seminare" a la lista cancionesFavoritas  
↳ agregar(cancionesFavoritas, "Seminare")  
// ahora la lista tiene un elemento más:  
↳ cancionesFavoritas  
["La colina de la vida", "Zamba por vos", "Seminare"]
```

## remover(unLista)

A partir de la [Lección 5: Listas](#) (<https://guides.flbulgarelli/mumuki-guia-fundamentos-javascript-vectores>)

Versión simplificada de `pop`

([https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Array/pop](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/pop))

Quita el último elemento de unaLista. Este es un procedimiento que no devuelve nada pero modifica a unaLista:

```
↳ let listaDeCompras = ["leche", "pan", "arroz", "aceite", "yerba"]
// removemos el último elemento
↳ remove(listaDeCompras)
// la "yerba" ya no está en lista de compras
↳ listaDeCompras
["leche", "pan", "arroz", "aceite"]
```

`posicion(unaLista, unElemento)`

A partir de la [Lección 5: Listas](#) (<https://guides.flbulgarelli/mumuki-guia-fundamentos-javascript-vectores>)

Versión simplificada de `indexof`

([https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos\\_globales/Array/indexOf](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/indexOf))

Nos dice en qué posición se encuentra `unElemento` dentro de `unaLista`. Si el elemento no está en la lista, devuelve `-1`

```
↳ let premios = ["dani", "agus", "juli", "fran"]
↳ posicion(premios, "dani")
0
↳ posicion(premios, "juli")
2
↳ posicion(premios, "feli")
-1
```

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)

