

Funciones y tipos de datos

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

```
10/2  
5  
2/10  
0.2
```

✅ ¡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.

🔍 Solución ➡ Consola

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

▶ Enviar

Siguiente Ejercicio: Funciones, uso ➤

Ejercicio 3: Funciones, uso

JS

¿Y esto con qué se come? Digo, ehm.... ¿cómo se usan estas funciones? ¿Cómo hago para pasarles parámetros 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) {
```

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; escribi 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 }
```

☒ Solución [>_Consola](#)

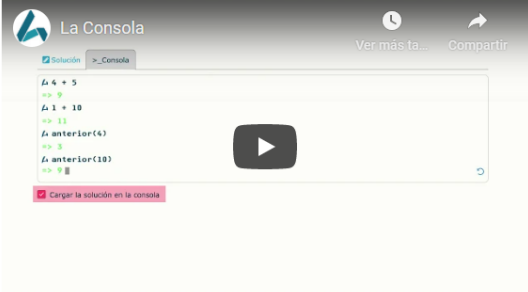
```
1 function anterior(numero) {  
2   return numero - 1;  
3 }  
4  
5 function triple(numero) {  
6   return numero*3;  
7 }  
8 function anteriorDelTriple(numero) {  
9   return anterior(triple(numero));  
10 }
```

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

La console 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 chirimbolito , que se llama *prompt*.

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



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

MATH FUNCIONES

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 *apenas* 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 console las siguientes expresiones:

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

💡 ¡Dame una pista!



 Solución  _Console

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

¡Veamos 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 }
```

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:


 Solución

 _Consola

```
1 function estaEntre (numero1, numero2, numero3) {
2   return numero1 > numero2 && numero1 < numero3;
3 }
4
5 function estaFueraDeRango (numero1, numero2, numero3) {
6   return numero1 < numero2 || numero1 > numero3;
7 }
8
```

 Enviar

Strings

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 *cadena 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

todoElMundo === "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"`.

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!

Solución >_Consola

```
1 function esFinDeSemana (dia) {
2
3   return dia == "domingo" || dia == "sábado";
4
5 }
```

Longitud

comienzaCon

¿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"
"aabb"
"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
```

Solución </>Biblioteca >_Consola

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

▶ Enviar

convertirEnMayuscula

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:

```
function gritar(nombre) {
  return "!" + convertirEnMayuscula (nombre) + "!"
}
```

▶ Enviar

If condicional

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
4     return numero1 ;
5
6   } else {
7
8     return numero2;
9   }
10 }
```

▶ Enviar

Signos iguales (diferencias)

Hola Fede, trato de ayudarte!

= se usa para ALMACENAR un valor en una variable..

EJEMPLO

```
num=1
```

== se usa para comparar valores que no suelen ser del mismo tipo

EJEMPLOS

```
if(1 == "1")
```

esto daría VERDADERO

al igual que

```
if(1==1)
```

=== se usa para comparar EXACTAMENTE (incluso de que tipo es)

EJEMPLOS

```
if(1 === "1")
```

Esto daría FALSO

```
if(1===1)
```

esto daría VERDADERO

Mayormente trabajando con funciones y IF suele necesitarse que haya al menos 1 return extra por cada IF porque al poner

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
3   if (numero===0) {
4     return 0;
5   }
6
7   if (numero>0) {
8     return 1; }
9   else {
10    return -1;
11  }
12
13
14 }
15
```

Otra forma

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

Escribi 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
3   if (numero===0) {
4     return 0;
5   }
6   else {
7     if (numero>0) {
8       return 1; }
9     else {
10      return -1;
11    }
12  }
13
14 }
15
16
```

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<15 || numero>15 && numero<100;
3 }
```

▶ Enviar

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"
```

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

```
Solución  Biblioteca  _Consola

1 function medallaSegunPuesto (puesto) {
2
3   if (puesto===1) {
4
5     return "oro";
6   } else {
7
8     if(puesto===2) {
9
10      return "plata" ;
11    } else {
12
13      if(puesto===3 ) {
14
15        return "bronce";
16      } else {
17
18        if (puesto>=4 || puesto===0) {
19
20          return "nada";
21        }
22      }
23    }
24  }
25 }
```

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	"hola"	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)

¡Terminaste Funciones y tipos de datos!

¡Excelente! 🎉

Eso significa que ya estás entrando en calor en un nuevo lenguaje: pudiste aplicar alternativas condicionales, funciones y expresiones. Y por si fuera poco, aprendiste en el camino algunas herramientas nuevas: los strings.

¿Querés seguir aprendiendo? ¡Acompañanos a la siguiente lección!

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.

```
Solución  Biblioteca  _Consola

1 function meConviene (pulgadas,memoria) {
2   return pulgadas>=32 && memoria>=8 &&
3     cuantoCuesta(pulgadas,memoria)<6000 ;
4 }
```

Enviar

¡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.

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"
```

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 }
```

▶ Enviar

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

▶ Enviar

Escribir cartelito

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. 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
3 escribirCartelito(titulo,nombre,apellido,booleano) {
4   if(booleano) {
5     return titulo+" "+apellido;
6   }
7   else {
8     return titulo+" "+nombre+" "+apellido;
9   }
10 }
11
12
```

▶ Enviar

Guillermo Ignacio Benítez hace 8 minutos ★

¡Hola Antonella! Antes de ver el código repasemos el enunciado. Este dice que la función tiene que recibir cuatro parámetros, siendo el cuarto un booleano, el cual nos indica si hay que escribir un cartel corto o largo. Fijate que en los ejemplos cuando este booleano es `true` se tiene que escribir el cartel corto y cuando es `false` se tiene que escribir el largo.

Ahora bien, el parámetro **booleano** puede tomar el valor `true` o `false`, pero **No** es necesario preguntar cuál es, sino que podemos utilizar directamente su valor. Acordate que cuando se usa un `if` se evalúa que la condición sea cierta o falsa. En el caso de que sea cierta, se ejecuta el código que tenga este `if` y en el caso de ser falsa se ejecuta el código que está en el `else` (o no se ejecuta nada si es que no hay un `else`). Entonces al escribir `if (booleano)` lo que estamos haciendo es usar directamente el valor de ese booleano, por lo que si es `true` esto equivale a decir `if (true)` y se va a ejecutar el código del `if`. En cambio si el parámetro **booleano** es `false` no se va a ejecutar el `if` y se va a ejecutar el `else`.

Con esto en mente, si el parámetro **booleano** es `true` se tiene que escribir el cartel corto, es decir: `return (titulo + " " + apellido);`, y si no (`else`), se tiene que escribir el largo: `return (titulo + " " + nombre + " " + apellido);`.

Guillermo Ignacio Benítez hace 1 minuto ★

¡Buenísimo! Perdón que sea repetitivo, pero acordate que no hay que comparar el parámetro **booleano** con un valor para usarlo, sino que hay que usarlo directamente:

```
function escribirCartelito (titulo,nombre,apellido,booleano) {  
  if (booleano) {  
    ...  
  } else {  
    ...  
  }  
}
```

Acá cuando ejecutes la función y le des el valor `true` al parámetro **booleano** se va a ejecutar el `if` y cuando le pases el valor `false` se va a ejecutar el `else`.

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!

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 escribirCartelitoOptimo  
  (titulo,nombre,apellido) {  
2  
3    return escribirCartelito (titulo, nombre,  
    apellido,longitud(nombre+" "+apellido) >=15) ;  
4  
5 }  
6
```

Enviar

[Solución](#) [Biblioteca](#) [_Consola](#)

```
1 function decisionConMoneda (coc,segundo,tercero) {  
2   if (coc=="cara") {  
3     return segundo  
4   }  
5   else {  
6     return tercero  
7   }  
8 }  
9
```

Enviar

Queremos saber el valor de las **cartas de truco** 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
```

Solución [Biblioteca](#) [_Consola](#)

```
1 function valorEnvido (numero) {
2   if(numero<7 && numero>0) {
3     return numero ;
4   }
5   else {
6     return 0
7   }
8 }
9
```

▶ Enviar

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

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

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

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

Solución [Biblioteca](#) [_Consola](#)

```
1 function puntosDeEnvidoTotales
2 (numero1,palo1,numero2,palo2) {
3   if(palo1===palo2) {
4     return (valorEnvido(numero1)+valorEnvido(numero2)+20)
5   }
6   else {
7     return Math.max(numero1,numero2);
8   }
9 }
```

▶ Enviar

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

Solución [Biblioteca](#) [_Consola](#)

```
1 function valorCantoTruco (canto) {
2   if (canto==="truco") {
3     return 2
4   }
5   else{
6     if(canto==="retruco") {
7       return 3
8     }
9     else {
10      return 4
11    }
12  }
13 }
14 }
```

▶ Enviar

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

Imprimir

¿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!

Ah, ¿y cómo se hace para imprimir varias líneas? ¡Llamando a `imprimir`!

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 }
```

▶ Enviar

Procedimientos en JavaScript

Funciones vs 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");  
}
```

Solución >_Consola

```
1 ''  
[  
  [  
    [  
      [  
        [  
          [  
            [  
              [  
                [  
                  [  
                    [  
                      [  
                        [  
                          [  
                        [  
                      [  
                    [  
                  [  
                [  
              [  
            [  
          [  
        [  
      [  
    [  
  ]  
]
```

▶ Enviar

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 😊.

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!* 😊

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

Solución [_Consola](#)

```
1 ...escribi tu solución acá...
```

▶ Enviar

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!* 😊

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 elefantesEquilibristas (numero) {
2
3   return numero + " " + "elefantes" + " " + "se" + " "
4   + "balanceaban";
5 }
```

▶ Enviar

Solución [_Consola](#)

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

▶ Enviar

Variables let

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 3.14159265358979*2*radio;
5 }
6 function areaCirculo (radio) {
7   return 3.14159265358979*(radio*radio);
8 }
```

▶ Enviar

¡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:

```
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 function ascensorSobrecargado (personas) {  
2  
3   let cargaMaximaEnKilogramos = 300;  
4  
5   return cargaMaximaEnKilogramos < personas*pesoPromedioPersonaEnKilogramos;  
6  
7 }  
~
```

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 `augmentarFortuna` 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 🤖).

🔍 Solución

📄 _Consola

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

▶ Enviar

💡 ¡Dame una pista!

¿Cómo usar `augmentarFortuna`? Por ejemplo así:

Atajos

```
// ¡podría tener cualquier cantidad!
// Aumento mi fortuna:
aumentarFortuna()
// Consulto de nuevo mi fortuna:
pesosEnMiBilletera // ¡Aumentó!
1000
```

✓ ¡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! 😊

+

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

=! Negación

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!

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

✓ Solución >_Consola

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

Otro de los operadores con el que ya te encuentres 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*, 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=="filosofo" && nacionalidad=="griego" && kilometros>2;  
3  
4 }  
}
```

Proposiciones

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:

A	B	A ^ 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

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
```


¿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.`, un jugador puede ganar de dos formas: cumpliendo su objetivo secreto o alcanzando el objetivo general de conquistar 30 países.

```
function gano(cumplioObjetivoSecreto, cantidadDePaísesConquistados) {  
  return cumplimientoObjetivoSecreto || cantidadDePaísesConquistados >= 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?

```
function gano(cumplioObjetivoSecreto, cantidadDePaísesConquistados) {  
  return cumplimientoObjetivoSecreto || cantidadDePaísesConquistados >= 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  
false  
=> false  
gano(false, 20)  
=> false
```

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 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  
2  
3  
4 function estaCerrado(esFeriado, dia, horario) {  
5  
6   return esFeriado || esFinDeSemana (dia) ||  
   dentroDeHorarioBancario(horario) ;  
7 }  
8  
9 function esFinDeSemana (dia) {  
10   return dia=="domingo" || dia=="sábado";  
11 }  
12  
13  
14
```

▶ Enviar

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! 😊

[Solución](#) [_Consola](#)

```
1 function tienenElMismoPadre (nom1,nom2) {  
2   return padreDe(nom1)==padreDe(nom2)  
3 }  
4 function tienenLaMismaMadre (nom1,nom2) {  
5   return madreDe(nom1)==madreDe(nom2)  
6 }  
7  
8 function sonMediosHermanos (nom1,nom2) {  
9  
10  return ((tienenElMismoPadre(nom1,nom2) &&  
   !tienenLaMismaMadre(nom1,nom2)) ||  
   (!tienenElMismoPadre(nom1,nom2) &&  
   tienenLaMismaMadre(nom1,nom2)))  
11 }  
12
```

▶ Enviar

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`:

tienen la misma madre	tienen el mismo padre	son medios hermanos
true	true	false
true	false	true
false	true	true
false	false	false

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

Siguiente Ejercicio: ¡Hola! Mi nombre es Xor >

Xor

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

A	B	$A \nabla B$
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.

Solución >_Consola

```
function xor(A,B) {
  return (!A && B) || (A && !B)
}
```

▶ Enviar

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

A	B	$A \nabla B$
V	V	F
V	F	V
F	V	V
F	F	F

Solución >_Consola

```
ReferenceError: A is not defined
xor(true,false)
=> true
xor(true,true)
=> false
```

✓ Cargar la solución en la consola

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;
}
```

Próbalos en la consola con los valores:

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

```
>> false
pagaConTarjeta(false, "crédito", 80)
-> true
pagaConTarjeta(true, "débito", 215)
-> false
pagaConTarjeta(true, "débito", 32)
-> true
pagaConTarjeta(true, "débito", 32)
```

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! Después vamos a ver cómo quedaría si delegamos. 😊

Solución [_Consola](#)

```
1 function puedeJubilarse (edad, sexo, años) {
2
3   return (((edad>=60 && sexo=="F") || (edad>=65 &&
4     sexo=="M")) && años>=30);
5 }
```

▶ Enviar

Usando delegación

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! Después vamos a ver cómo quedaría si delegamos. 😊

Solución [_Consola](#)

```
1 function cumpleEdadMinima(edad, sexo) {
2   return (edad>=60 && sexo=="F") || (edad>=65 &&
3     sexo=="M");
4 }
5 function tieneSuficientesAportes (aportes) {
6   return aportes>=30;
7 }
8
9 function puedeJubilarse (años, sexo, aportes){
10  return cumpleEdadMinima(años, sexo) &&
11    tieneSuficientesAportes(aportes);
12 }
```

▶ Enviar

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

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

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

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,acompañada,afeccion) {
2   return ((altura>=1.5 && !acompañada) || (altura>=1.2
3     && acompañada )) && !afeccion
}
```

▶ Enviar

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 alturaMax (altura,acompañada) {
2   return (altura>=1.5 && !acompañada) || (altura >=1.2
3     && altura<1.5 && acompañada) ;
4 }
5
6
7 function puedeSubirse (altura,acompañada,afeccion) {
8
9   return alturaMax(altura,acompañada) && !afeccion ;
10 }
```

▶ Enviar

Listas

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 (`[]` y `]`) separados por comas. Fácil, ¿no?

Probá en la consola las siguientes consultas:

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

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? 🗨

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]]`

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

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

```
↵ unaListaVacía
unaListaVacía;
^
ReferenceError: unaListaVacía is not defined
↵ longitud([])
=> 0
↵
```

Función agregar

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", "antorcha"];
//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?

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

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

Trasladar

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

Declararé 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!

¿Tenés dudas sobre cómo quitar y agregar elementos? Repasemos las siguientes funciones:

Solución </> Biblioteca >_Consola

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

▶ Enviar

Posición

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", "miercoles", "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.

¡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? ¡Averigüelo vos mismo!

Probá lo siguiente:

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

>_Consola </> Biblioteca

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

```
=> -1
```

```
↩
```

Si no lo contiene, me da un numero negativo

¡Ahora te toca a vos!

Solución [Biblioteca](#) [_Consola](#)

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
```

```
1
2
3 function contiene (lista, numero) {
4   return posicion(lista, numero) >= 0 ;
5 }
```

Es un booleano, si cumple la condición, es true

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.
```

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:

```
mesesDelAño[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?

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:

```
mesesDelAño[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
["1", "2"][48]
=> undefined
```

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"
```

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

```
1 function medallaSegunPuesto(numero)
2 {
3   if ( numero===1 ) {
4     return "oro"
5   }
6   if(numero===2) {
7     return "plata"
8   }
9 }
10
11 if (numero===3) {
12   return "bronce"
13 }
14 else {
15   return "nada"
16 }
17 }
```

▶ Enviar

For

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

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

💡 Dame una pista!

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

```
1 function saludar(personas) {
2   for(let persona of personas) {
3     imprimir("hola"+" "+persona);
4   }
5 }
6
7
```

```
1 function saludar(lista) {
2   for(let personas of lista) {
3
4     imprimir("hola"+" "+personas);
5   }
6 }
7
```



```
saludar(["anto", "paulo"])
hola anto
hola paulo
```

Registros

Los monumentos que probaste en el ejercicio anterior están representados como *registros*, y cada una de sus características (nombre, locación, 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.

como *registros*, y cada una de sus características (nombre, locación, 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

[Solución](#) [Biblioteca](#) [_Consola](#)

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

▶ Enviar

[Solución](#) [Biblioteca](#) [_Consola](#)

```
torreAzadi
=> {nombre:"Torre Azadi",locacion:"Teherán, Irán",anioDeConstruccion:1971}
```

¡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.

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

```
torreAzadi.locacion
"Agra, India"
torreAzadi.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"
mercurio.temperaturaPromedio
=> 67
mercurio.tieneAnillos
=> false
```

Registros

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



Desarrolla 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"
```

[Solución](#)

[Biblioteca](#)

[_Consola](#)

```
1
2
3 function temperaturaDePlaneta (planeta) {
4   return planeta.nombre + " "+ "tiene una temperatura
promedio de" + " "+ planeta.temperaturaPromedio + " "+
"grados"
5 }
```

Mover

Modificación de un registro

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.

```
1 function moverArchivo (registro, nuevaRuta) {
2
3   registro.ruta =nuevaRuta
4
5 }
```

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. 📅

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

```
↳ esDelMilenioPasado({ ruta: "D:\\fotonacimiento.jpg", crea
cion: "14/09/1989" })
true
```

🔗 ¡Dame una pista!

Quizá te pueda servir la función `anio`:

```
↳ anio("04/11/1993")
1993
```

[Solución](#)

[Biblioteca](#)

[_Consola](#)

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

▶ Enviar

Lista campo de un registro

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", "frambuesas"], tiempoDeCoccion: 80 }
let lemonPie = { ingredientes: ["jugo de limón", "almidón de maíz", "leche", "huevos"], tiempoDeCoccion: 65 }
```

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

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

```
1 function masDifícilDeCocinar (postre1, postre2) {
2
3   if ((longitud(postre1.ingredientes) >= longitud(postre2.ingredientes))) {
4
5     return postre1; }
6
7   else {
8
9     return postre2;
10  }
11 }
12 }
```

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

Desarrollá el procedimiento `agregarAPostresRápidos`, 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 `agregarAPostresRápidos` 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.

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!

[Solución](#) [Biblioteca](#) [_Consola](#)

```
1 function agregarAPostresRápidos (lista, postre) {
2   if (postre.tiempoDeCoccion <= 60)
3     agregar(lista, postre)
4 }
```

▶ Enviar

[Solución](#) [Biblioteca](#) [_Consola](#)

```
1 function endulzarMenu (menu) {
2
3   agregar(menu.postre.ingredientes, "azúcar")
4 }
```

¡Terminaste Registros!

Durante la lección aprendiste cuál es la utilidad de esta estructura de datos llamada registro, cómo acceder a sus campos y modificarlos, y hasta viste que pueden *anidarse* (es decir, que haya un registro dentro de otro). ¡Felicitaciones! 🍷

Registros dentro de listas

Ana, contadora de una conocida empresa 🏢, tiene registros para representar los balances de cada mes y una lista para guardarlos. Por ejemplo, para el último semestre del año pasado registró los siguientes:

```
//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:

```
//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 habíamos registrado los balances del primer trimestre de este año:

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. 😊

Ahora que sabemos la función que necesitamos (`gananciaTotal`), razonemos cómo hacerla...

Vamos de a poquito 🐼: si la lista no tuviera elementos, ¿cuánto debería ser la sumatoria? ¡0!

```
function gananciaTotal0(balancesDeUnPeriodo) {  
  let sumatoria = 0;  
  return sumatoria;  
}
```

¿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 `gananciaTotal4` 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 }
```

¡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*? 🐼

Lo que tenemos que hacer, entonces, es repetir la operación de acumular varias veces, una por cada elemento de la lista. ¡Digamos hola 🗨️ (nuevamente) al `for...of`!

```
function gananciaTotal(balancesDeUnPeriodo) {
  let sumatoria = 0;
  for (let 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 },
  { mes: "agosto", ganancia: 10 }
])
=> 18
```

```
gananciaTotal([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 10 },
  { mes: "marzo", ganancia: -20 }
])
=> -8
```

contadores

¡Ana tiene nuevos requerimientos! Ahora nos pidió lo siguiente: "Quiero saber cuántos balances fueron positivos, es decir, aquellos en los que la ganancia fue mayor a cero".

Completá la función `cantidadDeBalancesPositivos`. Si prestás atención, notarás que tiene una estructura similar al problema anterior. 😊

¿Dame una pista!

Solución

Consola

```
1 function
2 cantidadDeBalancesPositivos(balancesDeUnPeriodo) {
3   let cantidad = 0 ;
4   for (let balance of balancesDeUnPeriodo) {
5     if (balance.ganancia > 0) {
6
7       cantidad = cantidad + 1;
8     }
9   }
10  return cantidad ;
11
12
13
14 }
15
```

¡Ana tiene nuevos requerimientos! Ahora nos pidió lo siguiente: "Quiero saber cuántos balances fueron positivos, es decir, aquellos en los que la ganancia fue mayor a cero".

Completá la función `cantidadDeBalancesPositivos`. Si prestás atención, notarás que tiene una estructura similar al problema anterior. 😊

💡 ¡Dame una pista!

[Solución](#) [>_Consola](#)

```
1 function
  cantidadDeBalancesPositivos(balancesDeUnPeriodo) {
2   let cantidad = 0 ;
3   for (let balance of balancesDeUnPeriodo) {
4
5     if (balance.ganancia > 0) {
6       /*pongo cantidad ++ porque le sumo 1 */
7       cantidad++;
8     }
9   }
10
11   return cantidad ;
12
13 }
14
15
```

¡Ana tiene nuevos requerimientos! Ahora nos pidió lo siguiente: "Quiero saber cuántos balances fueron positivos, es decir, aquellos en los que la ganancia fue mayor a cero".

Completá la función `cantidadDeBalancesPositivos`. Si prestás atención, notarás que tiene una estructura similar al problema anterior. 😊

💡 ¡Dame una pista!

[Solución](#) [>_Consola](#)

```
{mes:"marzo",ganancia:-20}]
cantidadDeBalancesPositivos([
  { mes: "enero", ganancia: 2 },
  { mes: "febrero", ganancia: 10 },
  { mes: "marzo", ganancia: -20 }
])
=> 2
```

Promedios

Pasemos al siguiente requerimiento de Ana. Ya podemos calcular una sumatoria de ganancias y también crear contadores, ahora vamos a calcular promedios. 😊

Ana quisiera saber dado un conjunto cualquiera de balances cuál es su `gananciaPromedio`.

```
gananciaPromedio([
  { mes: "marzo", ganancia: 8 },
  { mes: "agosto", ganancia: 10 }
])
9
```

[Solución](#) [>_Consola](#)

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

Viendo que podemos hacer todo lo que nos pide, Ana quiere saber la ganancia promedio de los balances positivos. 🤖

Definí las funciones:

- `gananciaPositiva`, que es la suma de las ganancias de los balances positivos
- `promedioGananciasPositivas` utilizando `gananciaPositiva` y `cantidadDeBalancesPositivos`.

💡 ¡Dame una pista!

[Solución](#) [>_Consola](#)

```
1 function gananciaPositiva (balancesDeUnPeriodo) {
2
3   let sumatoria=0;
4
5   for (let balance of balancesDeUnPeriodo) {
6
7     if (balance.ganancia>0 ) {
8
9       sumatoria = sumatoria + (balance.ganancia)
10    }
11  }
12  return sumatoria;
13
14 }
15
16
17 function promedioGananciasPositivas
  (balancesDeUnPeriodo) {
18
19   return gananciaPositiva
    (balancesDeUnPeriodo)/cantidadDeBalancesPositivos(balancesDeUnPeriodo)
20 }
21
```

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 quisiéramos 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? 🗨

Devolver listas

Lamentablemente no se puede usar la función `promedio` con nuestra lista de registros 😞. Lo que necesitamos es una lista que tenga solo las ganancias de cada balance. Para ello debemos transformar, o *mapear*, cada elemento de la lista.

Completá la función `ganancias` que toma una lista de balances y devuelve una lista que solo posea solo las ganancias de cada uno.

```
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
5     agregar(ganancias, balance.ganancia)
6   }
7   return ganancias;
8 }
9
```

▶ Enviar

Con la programación se puede hacer cualquier cosa, o casi 😞. Ya hicimos una función para poder saber la cantidad de balances positivos (`cantidadDeBalancesPositivos`), ahora vamos a ver cómo podemos hacer para saber cuáles son esos balances. 🗨

Completá la función `balancesPositivos` que toma los balances de un período y devuelve una lista con aquellos cuya ganancia fue mayor a cero.

💡 Dame una pista!

Solución >_Consola

```
1 function balancesPositivos(balancesDeUnPeriodo) {
2   let balances = [];
3
4   for (let balance of balancesDeUnPeriodo) {
5
6     if(balance.ganancia > 0) {
7
8       agregar(balances, balance)
9     }
10  }
11  return balances;
12 }
13
```

Con la programación se puede hacer cualquier cosa, o casi 😞. Ya hicimos una función para poder saber la cantidad de balances positivos (`cantidadDeBalancesPositivos`), ahora vamos a ver cómo podemos hacer para saber cuáles son esos balances. 🗨

Completá la función `balancesPositivos` que toma los balances de un período y devuelve una lista con aquellos cuya ganancia fue mayor a cero.

💡 Dame una pista!

Solución >_Consola

```
1 function balancesPositivos(balancesDeUnPeriodo) {
2   let balances = [];
3
4   for (let registro of balancesDeUnPeriodo) {
5
6     if(registro.ganancia > 0) {
7
8       agregar(balances, registro)
9     }
10  }
11  return balances;
12 }
13
```


Con la programación se puede hacer cualquier cosa, o casi 😊. Ya hicimos una función para poder saber la cantidad de balances positivos (`cantidadDeBalancesPositivos`), ahora vamos a ver cómo podemos hacer para saber cuáles son esos balances. 31

Completé la función `balancesPositivos` que toma los balances de un período y devuelve una lista con aquellos cuya ganancia fue mayor a cero.

💡 ¡Dame una pista!

✓ Solución >_Consola

```
{ mes: "enero", ganancia: 40 },
{ mes: "febrero", ganancia: 12 },
{ mes: "marzo", ganancia: 8 },
{ mes: "abril", ganancia: -5 }
})
=> [{mes:"enero",ganancia:40},{mes:"febrero",ganancia:12},
{mes:"marzo",ganancia:8}]
```

Ahora que tenemos la función `ganancias` y `balancesPositivos` podemos utilizar la función `promedio` genérica para saber cuál es el promedio de ganancia de los balances positivos.

Definí la función `gananciasDeBalancesPositivos` y luego usala junto a `promedio` para definir `promedioDeBalancesPositivos`.

💡 ¡Dame una pista!

✓ Solución </> Biblioteca >_Consola

```
1 function
  gananciasDeBalancesPositivos(balancesDeUnPeriodo) {
2   return
    ganancias(balancesPositivos(balancesDeUnPeriodo));
3 }
4
5
6 function promedioDeBalancesPositivos
  (balancesDeUnPeriodo) {
7
8   return promedio(gananciasDeBalancesPositivos
    (balancesDeUnPeriodo)) ;
9 }
```

Máximo

Vamos a conocer una nueva función, `maximo`, que nos permite conocer cuál es el mayor valor en una lista de números. Por ejemplo:

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

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
```

✓ Solución </> Biblioteca >_Consola

```
1 function maximaGanancia (balances) {
2
3   return maximo (ganancias(balances));
4
5 }
```

▶ Enviar

Vamos a conocer una nueva función, `maximo`, que nos permite conocer cuál es el mayor valor en una lista de números. Por ejemplo:

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

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
```

✓ Solución </> Biblioteca >_Consola

```
1 function maximaGanancia (registro) {
2
3   return maximo (ganancias(registro));
4
5 }
```

▶ Enviar

¡Vamos a terminar esta lección con todo! 🍌

Para eso vamos a hacer las siguientes funciones:

- `meses`, la cual dada una lista con registros devuelve una lista de meses `31`;
- `afortunados`, que filtra aquellos registros que tuvieron una ganancia mayor a \$1000 `32`;
- `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 },
```

```
1 function meses (mesesDelMes) {
2   let meses=[];
3   for(let registros of mesesDelMes) {
4     agregar(meses,registros.mes)
5   }
6   return meses;
7 }
8
9 function afortunados(mesesDelMes) {
10  let meses=[];
11  for(let registros of mesesDelMes) {
12    if (registros.ganancia > 1000) {
13      agregar(meses,(registros));
14    }
15  }
16  return meses;
17 }
18 function mesesAfortunados (mesesDelMes) {
19   return meses(afortunados(mesesDelMes))
20 }
```

En una conocida red social trabajamos con publicaciones, que son registros que tienen el apodo de quien sube la publicación y el texto que contiene la misma:

```
{ apodo: "feli", texto: "tengo sueño" }
```

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 mí 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`

[Solución](#) [Biblioteca](#) [_Consola](#)

```
1
2 function publicacionesCortasDelHilo
3   (apodo,hiloDeEjemplo) {
4
5   let hilo=[];
6   let aux=0;
7   for (let registro of hiloDeEjemplo) {
8
9     aux=longitud(registro.apodo)
10    +longitud(registro.mensaje) ;
11
12    if(aux <20 && apodo===registro.apodo) {
13
14      agregar(hilo,registro)
15    }
16  }
17  return hilo;
18 }
19
```

▶ Enviar

En una conocida red social trabajamos con publicaciones, que son registros que tienen el apodo de quien sube la publicación y el texto que contiene la misma:

```
{ apodo: "feli", texto: "tengo sueño" }
```

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 mí 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`

[Solución](#)

[Biblioteca](#)

[Consola](#)

```
1
2 function publicacionesCortasDelHilo
  (apodo,hiloDeEjemplo) {
3
4   let hilo=[];
5   for (let registro of hiloDeEjemplo) {
6
7     if(longitud(registro.mensaje) <20 &&
      apodo==registro.apodo) {
8
9       agregar(hilo,registro)
10    }
11  }
12  }
13  return hilo;
14 }
15
16
```

▶ Enviar