*Abstracción*

Analicemos el siguiente código. Se declara una variable que, a través de un iterador, va *acumulando* la suma del contador:

| let total = 0  for (let i = 1; i <= 10; i++) {  total += i  }  console.log(total) // 55 |
| --- |

Ahora lo presentamos resumido en una función, y ocupa sólo una línea de código:

| console.log( sumarRango(1, 10) ) // 55 |
| --- |

El segundo caso es lo que denominamos una *abstracción*. Resumimos un grupo complejo de instrucciones bajo una palabra clave (función) que sugiere cuál es el problema a resolver por la misma. Las abstracciones ocultan detalles sobre la operación a resolver y nos permite “hablar” sobre los problemas en un nivel más alto (o mayor grado de abstracción).

El segundo ejemplo es *más corto y fácil de interpretar*, pero hay que tener más claro ciertos conceptos para poder aplicarlo efectivamente (funciones, parámetros, return, etc.) y trabajar en un nivel de abstracción superior, lo cual nos ahorra tiempo de desarrollo y claridad de escritura. Si no entendemos cómo puede funcionar una función de este tipo, es común que *desconfiemos* de su versatilidad y caigamos en la declaración paso a paso de la solución, como en el primer ejemplo.

Como desarrolladores constantemente estamos creando funciones y abstracciones para pensar en un nivel superior y poder construir soluciones complejas a los problemas que se nos presentan. En JS hay muchos métodos nativos incorporados que, como abstracciones, nos ofrecen soluciones a problemas recurrentes; sumado a la posibilidad de que nosotros podemos declarar las nuestras.

*Funciones de orden superior*

Una función de orden superior (también conocidas como *funciones de alto orden*, o *higher-order functions)* es aquella que **retorna una función**, o **recibe una función por parámetro**. Este tipo de funciones nos permiten abstraernos sobre *acciones* y no sólo valores. En esta clase nos concentraremos más en la segunda acepción.

***Retornar funciones***

En el primer caso, podremos tener una función que retorna una función, lo cual nos permitiría crear funciones con un esquema superior. Por ejemplo:

| function mayorQue(n) {  return (m) => m > n  }  let mayorQueDiez = mayorQue(10)  console.log( mayorQueDiez(12) ) // true  console.log( mayorQueDiez(8) ) // false |
| --- |

En este caso, *mayorQue(n)* retorna una función que compara un valor contra *n* y retorna true o false (porque es el resultado de la comparación). En *mayorQueDiez* se termina asignando la función que retorna el llamado de *mayorQue(10)*. Al ser llamada con el valor de 10, la asignación se resuelve de la siguiente forma:

***let mayorQueDiez = (m) => m > 10***

De ahí que luego podemos llamar a la función y compara siempre contra 10. Si lo creamos con un llamado a mayorQue(5), la función siempre compararía contra 5.

Otro ejemplo:

| function asignarOperacion(op) {  if (op == "sumar") {  return (a, b) => a + b  } else if (op == "restar") {  return (a, b) => a - b  }  }  let suma = asignarOperacion("sumar")  let resta = asignarOperacion("restar")  console.log( suma(4, 6) ) // 10  console.log( resta(5, 3) ) // 2 |
| --- |

Si bien este ejemplo es un poco redundante, sirve para entender la capacidad de *retornar funciones* que podemos desarrollar. En este caso, según el parámetro *op* se termina asignando un return de función u otro a las variables declaradas.

***Recibir funciones por parámetro***

La segunda acepción significa escribir funciones que puedan recibir funciones por parámetro. Para hacerlo más claro, empecemos con un ejemplo. Supongamos que quiero recorrer un array y *hacer algo* con cada uno de sus elementos.

| function porCadaUno(arr, fn) {  for (const el of arr) {  fn(el)  }  } |
| --- |

Esta función recibe un *array* por primer parámetro y una ***función***por el segundo, y se encarga de recorrer el array y por cada elemento del mismo hacer un llamado a la función mencionada enviando dicho elemento por parámetro.

| const numeros= [1, 2, 3, 4]  porCadaUno(numeros, console.log)  // 1  // 2  // 3  // 4 |
| --- |

Al enviar *console.log* por parámetro, se entiende que se ejecuta esa función con cada elemento del array. Lo interesante de esto es que podemos enviar funciones diferentes en distintos llamados y ejecutar, en este caso, distintas acciones sobre los elementos del array, todo con la misma función.

| let total = 0  function acumular(num) {  total += num  }  porCadaUno(numeros, acumular)  console.log(total) // 10 |
| --- |

Algo usual es definir la función directamente sobre el parámetro como una función anónima aprovechando la sintaxis de arrow function. Esto evita tener que andar declarando funciones todo el tiempo y definir acciones más dinámicas:

| const duplicado = []  porCadaUno(numeros, (el)=> {  duplicado.push(el \* 2)  })  console.log(duplicado) // [2, 4, 6, 8] |
| --- |

*Métodos de búsqueda y transformación*

Javascript incorpora nativamente varias funciones de orden superior. Específicamente veremos a continuación métodos para operar sobre **arrays** que trabajan con esta lógica. Los siguientes métodos funcionan siempre *iterando* sobre el array correspondiente, y reciben una *función por parámetro*, la cual recibe a la vez el elemento del array que se está iterando. Cada uno de los métodos siguientes están pensados para solucionar problemas recurrentes con los arrays, como puede ser buscar un elemento bajo una condición, filtrar el array, transformarlo, reordenarlo, etc.

**For Each**

El método ***forEach()*** es similar al ejemplo *porCadaUno* anterior. Itera sobre el array de referencia y por cada elemento del array ejecuta la función que enviemos por parámetro, la cual recibe a su vez el elemento del array que se está recorriendo:

| const numeros = [1, 2, 3, 4, 5, 6]  numeros.forEach( (num)=> {  console.log(num)  } ) |
| --- |

**Find**

El método *find()* recibe una función de *comparación* por parámetro (es decir, una función que captura el elemento que se está recorriendo y retorne *true* o *false* según la comparación que ejecuta). El método ***retorna*** *el primer elemento del array que cumpla con esa condición*:

| const cursos = [  {nombre: 'Javascript', precio: 15000},  {nombre: 'ReactJS', precio: 22000},  ]  const resultado = cursos.find((el) => el.nombre === "ReactJS")  const resultado2 = cursos.find((el) => el.nombre === "DW")  console.log(resultado) // {nombre: 'ReactJS', precio: 22000}  console.log(resultado2) // undefined |
| --- |

Nótese que el find *retorna* el primer **elemento** del array que cumpla con la condición enviada, de ahí que podemos almacenarlo en una variable o usarlo de referencia para otro proceso. Si no hay ninguna coincidencia en el array, el método find retorna *undefined*.

**Filter**

El método ***filter()*** recibe, al igual que el find, una función comparadora por parámetro, y *retorna un nuevo array* con todos los elementos que cumplan esa condición. En caso de no haber coincidencias, retornará un array vacío:

| const cursos = [  {nombre: 'Javascript', precio: 15000},  {nombre: 'ReactJS', precio: 22000},  {nombre: 'AngularJS', precio: 22000},  {nombre: 'Desarrollo Web', precio: 16000},  ]  const resultado = cursos.filter((el) => el.nombre.includes('JS'))  const resultado2 = cursos.filter((el) => el.precio < 14000)  console.log(resultado)  /\* [  {nombre: 'ReactJS', precio: 22000},  {nombre: 'Angular', precio: 22000}  ] \*/  console.log(resultado2) // [] |
| --- |

**Some**

El método ***some()*** funciona igual que el *find()*, recibiendo una función de búsqueda, con la diferencia que en vez de retornar el *elemento* encontrado, el método *some* retorna *true* o *false* según el resultado de la iteración de búsqueda:

| console.log( cursos.some((el) => el.nombre == "Desarrollo Web"))  // true  console.log( cursos.some((el) => el.nombre == "VueJS"))  // false |
| --- |

**Map**

El método ***map()*** se utiliza para crear un nuevo array con todos los elementos del array original, transformados según las operaciones de la función enviada por parámetro. El nuevo array obtenido tiene la misma cantidad de elementos que el array original, pero los elementos que se almacenan son el *return* de la función enviada por parámetro:

| const cursos = [  {nombre: 'Javascript', precio: 15000},  {nombre: 'ReactJS', precio: 22000},  {nombre: 'AngularJS', precio: 22000},  {nombre: 'Desarrollo Web', precio: 16000},  ]  const nombres = cursos.map((el) => el.nombre)  console.log(nombres)  // [ 'Javascript', 'ReactJS', 'AngularJS', 'Desarrollo Web' ] |
| --- |

En este ejemplo los elementos del nuevo array son el *return* de la función enviada al map: la función retorna la propiedad nombre de cada elemento y eso es lo que se almacena en el nuevo array *nombres*.

Este método se utiliza mucho para *transformación* de arrays, ya que genera un nuevo array recorriendo el primero. Si quisiera aumentar el precio de todos los cursos en este ejemplo, puedo mapear y retornar una copia de los elementos con el precio modificado:

| const actualizado = cursos.map((el) => {  return {  nombre: el.nombre,  precio: el.precio \* 1.25  }  })  console.log(actualizado)  /\* [  { nombre: 'Javascript', precio: 18750 },  { nombre: 'ReactJS', precio: 27500 },  { nombre: 'AngularJS', precio: 27500 },  { nombre: 'Desarrollo Web', precio: 20000 }  ] \*/ |
| --- |

**Reduce**

El método *reduce()* nos permite obtener un único valor tras iterar sobre el array. Es decir, funciona como un método que *resume* el array a un único valor de retorno. Se utiliza mucho cuando, por ejemplo, queremos acumular la suma de alguna propiedad de los elementos almacenados, u obtener algún resultado general sobre todo el array.

A diferencia de los métodos anteriores, el método *reduce* recibe dos parámetros. El primero es la función que ordena qué queremos resumir del array, y recibe primero un parámetro que funciona como *acumulador* y el segundo es el *elemento* del array que iteramos.

| const numeros = [1, 2, 3, 4, 5, 6]  const total = numeros.reduce((acumulador, elemento) => acumulador + elemento, 0)  console.log(total) // 21 |
| --- |

En este ejemplo, en el *acumulador* sumamos cada elemento del array, y al terminar la iteración nos devuelve ese resultado. El segundo parámetro del reduce, que aquí se ve como 0, es el *valor inicial del acumulador*.

Con esto podría, por ejemplo, pensando en un simulacro de compra, sumar el precio de todos los productos elegidos:

| const miCompra = [  { nombre: 'Desarrollo Web', precio: 20000 },  { nombre: 'Javascript', precio: 18750 },  { nombre: 'ReactJS', precio: 27500 }  ]  const total = miCompra.reduce((acc, el) => acc + el.precio, 0)  console.log(total) // 66250 |
| --- |

**Sort**

El método *sort()* nos permite reordenar un array según un criterio que definamos. Este método es *destructivo,* es decir que modifica el array sobre el cual se llama. El método recibe una *función de comparación* por parámetro que recibe dos elementos del array a la vez. La función debe retornar un valor numérico (1, -1, 0) que indica qué elemento se posiciona antes o después.

Para ordenar números, basta con restar uno al otro, y el orden indica si será ordenado de forma ascendente o descendente:

| const numeros = [ 40, 1, 5, 200 ];  numeros.sort((a, b) => a - b); // [ 1, 5, 40, 200 ]  numeros.sort((a, b) => b - a); // [ 200, 40, 5, 1 ] |
| --- |

Para ordenar un array por algún string, debemos definir una función comparadora que retorne un valor numérico de referencia, según queramos el orden ascendente o descendente:

| const items = [  { name: 'Pikachu', price: 21 },  { name: 'Charmander', price: 37 },  { name: 'Pidgey', price: 45 },  { name: 'Squirtle', price: 60 }  ]  items.sort((a, b) => {  if (a.name > b.name) {  return 1;  }  if (a.name < b.name) {  return -1;  }  // a es igual a b  return 0;  }) |
| --- |

**Ejemplo aplicado**

Veamos a continuación un ejemplo aplicado, donde se utilizan los métodos presentados en una solución más aproximada al escenario de la tienda virtual.

| const productos = [{ id: 1, producto: "Arroz", precio: 125 },  { id: 2, producto: "Fideo", precio: 70 },  { id: 3, producto: "Pan" , precio: 50},  { id: 4, producto: "Flan" , precio: 100}]  const buscado = productos.find(producto => producto.id === 3)  console.log(buscado) //{id: 3, producto: "Pan", precio: 50}  const existe = productos.some(producto => producto.nombre === “Harina”)  console.log(existe ) // false  const baratos = productos.filter(producto => producto.precio < 100)  console.log(baratos)  // [{id: 2,producto:"Fideo",precio:70},{id:3,producto:"Pan",precio: 50}]  const listaNombres = productos.map(producto => producto.nombre)  console.log(listaNombres);  //[“Arroz”, “Fideo”, “Pan”, “Flan”] |
| --- |

***El objeto Math***

Es normal que en programación realicemos operaciones matemáticas regularmente, y para eso Javascript nos provee el objeto Math que funciona como un contenedor de herramientas para realizar dichas operaciones.

El objeto Math contiene una serie de métodos que nos permiten realizar algunas operaciones matemáticas más complejas. Veremos a continuación algunas de las funciones que se desprenden de este objeto, aunque el repertorio completo lo pueden ver en su documentación: [Math - JavaScript | MDN](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math)

**Propiedades**

Se puede acceder a algunas constantes matemáticas a través del objeto Math, como pueden ser el número PI o la constante de Euler:

| console.log( Math.E ) // 2.718281828459045  console.log( Math.PI ) // 3.141592653589793 |
| --- |

**Min y Max**

Los métodos de Math.min() y Math.max() reciben una *serie de argumentos numéricos* y devuelven aquel que tenga el valor máximo o mínimo, según corresponda:

| console.log( Math.max(55, 13, 0, -25, 93, 4) ) // 93  console.log( Math.min(55, 13, 0, -25, 93, 4) ) // -25 |
| --- |

También se pueden referenciar los valores de *infinito* positivo o negativo a través de la variable global **Infinity**, de tipo number.

| console.log( Math.max(55, Infinity, 0, -25, 93, 4) ) // Infinity  console.log( Math.min(55, 13, 0, -Infinity, 93, 4) ) // -Infinity |
| --- |

**Ceil, Floor y Round**

Estos métodos sirven para *redondear* algún valor numérico a un número entero cercano.

| const pi = Math.PI // 3.141592653589793  // CEIL: devuelve el entero mayor o igual más próximo  console.log( Math.ceil(pi) ) // 4  // FLOOR: devuelve el entero más cercano redondeado hacia abajo  console.log( Math.floor(pi) ) // 3  // ROUND: retorna el valor de un número redondeado al entero más cercano  console.log( Math.round(pi) ) // 3 |
| --- |

**Square Root**

El método Math.sqrt() retorna la *raíz cuadrada* de un número. Si se le envía un número negativo, el método retorna *NaN*.

| Math.sqrt(9) // 3  Math.sqrt(2) // 1.414213562373095  Math.sqrt(1) // 1  Math.sqrt(-1) // NaN |
| --- |

**Random**

El método *Math.random()* genera un número pseudo-aleatorio entre 0 y 1, siendo el 0 límite inclusivo y el 1 exclusivo.

| console.log( Math.random() ) // 0.6609867980868442  console.log( Math.random() ) // 0.09291446900104305  console.log( Math.random() ) // 0.6597817047013095 |
| --- |

Para generar números aleatorios dentro de un rango deseado, distinto de 0-1, podemos multiplicar su resultado por el rango esperado. A la vez podemos sumar el límite inferior si lo necesitamos:

// números entre 0 y 10

console.log( Math.random() \* 10 )

// números entre 0 y 50

console.log( Math.random() \* 50)

// números entre 20 y 50

console.log( Math.random() \* 30 + 20 )

Nótese que en el último ejemplo quiero generar números entre 20 y 50. Por eso, el *rango* de números es de 30 a partir del número 20 (límite inferior adicionado). Pero todos los números siguen conteniendo una larga serie de decimales.

Esto se suele combinar con las funciones de *redondeo* para obtener números enteros aleatoriamente, que suelen ser de uso más común:

| const generadorNumero = () => {  return Math.round( Math.random() \* 100 )  }  console.log( generadorNumero() ) |
| --- |

Al usar Math.round, esta función retornará números aleatorios en el rango de 0-100 inclusive. Si usara *Math.ceil* los números irían de 1 a 100, ya que siempre redondeará hacia arriba; y si usa *Math.floor* el rango sería de 0 a 99.

***La clase DATE***

Seguramente en algún momento necesitemos manipular *fechas* dentro de los datos que manejamos. Para ésto, javascript posee una **clase** diseñada para representar fechas.

Instanciar un objeto Date nos genera la fecha y tiempo actual:

| console.log( new Date() )  // Fri Dec 17 2021 11:35:08 GMT-0300 (hora estándar de Argentina) |
| --- |

**Constructor**

El constructor de la clase Date nos permite crear objetos date con fechas diferentes. Puede recibir parámetros en el orden *año, mes, día*, *hora, minutos, segundos, milisegundos* (todos tipo number). Hay que tener en cuenta que la convención con la que trabaja Javascript para construir fechas cuenta los meses a partir del 0 (0 = enero, 11 = diciembre) y los días a partir del 1:

| console.log(new Date(2020, 1, 15))  // Sat Feb 15 2020 00:00:00 GMT-0300 (hora estándar de Argentina)  const casiNavidad = new Date(2021, 11, 25, 23, 59, 59)  console.log(casiNavidad)  // Sat Dec 25 2021 23:59:59 GMT-0300 (hora estándar de Argentina) |
| --- |

El constructor también puede crear una fecha a partir de un *string* con formato específico:

| const casiNavidad = new Date("December 25, 2021 23:59:59")  console.log(casiNavidad)  // Sat Dec 25 2021 23:59:59 GMT-0300 (hora estándar de Argentina) |
| --- |

**Obtener un valor singular de la fecha**

Instanciado un objeto Date, podemos aplicar distintos métodos que nos devuelven determinados valores de la misma.

* *getFullYear()* nos devolverá el number que representa el año creado.
* *getMonth()* nos retornará el number que representa el mes (entre 0 y 11).
* *getDay()* nos retorna el number que representa el día creado (1 = lunes, 7 = domingo).

| const hoy = new Date("December 17, 2021")  console.log(hoy.getFullYear()) // 2021  console.log(hoy.getMonth()) // 11 (diciembre)  console.log(hoy.getDay()) // 5 (viernes) |
| --- |

**Presentación**

La clase también tiene distintos métodos que presentan la fecha con distintos formatos posibles de tipo *string*. Según la utilidad que queramos presentar, podemos optar por alguna de las siguientes opciones, aunque hay más disponibles que pueden investigar en la documentación ([Date - JavaScript | MDN](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Date))

| const hoy = new Date("December 17, 2021")  console.log( hoy.toDateString() ) // Fri Dec 17 2021  console.log( hoy.toLocaleString() ) // 17/12/2021 00:00:00  console.log( hoy.toLocaleDateString() ) // 17/12/2021  console.log( hoy.toTimeString() ) // 00:00:00 GMT-0300 (hora estándar de Argentina) |
| --- |

**Diferencias**

Es posible calcular diferencias entre fechas pero entendiendo que los resultados se generan en *milisegundos*. Si quisiera calcular la diferencia de días entre dos fechas habría que generar cálculos adicionales sobre esta diferencia en milisegundos, lo cual no resulta muy agradable. Por suerte existen librerías que solucionan estos problemas de forma eficiente y rápida, pero las trabajaremos en clases posteriores.

Algunos ejemplos de medir diferencias:

| const navidad = new Date("December 25, 2021")  const hoy = new Date("December 17, 2021")  console.log( navidad - hoy ) // 691200000  const milisegundosPorDia = 86400000  console.log( (navidad - hoy ) / milisegundosPorDia) // 8 |
| --- |

| const inicio = new Date()  for (let i = 0; i < 1000; i++) {  console.log("haciendo tiempo")  }  const final = new Date()  console.log("El proceso tardó: " + (final - inicio) + " milisegundos")  // El proceso tardó: 396 milisegundos |
| --- |