Tipos de variables

JavaScript es un lenguaje débilmente tipado. Esto quiere decir que no indicamos de qué tipo es cada variable que declaramos. Todas las variables admiten todos los tipos, y pueden ser reescritas. Es una de las cosas buenas y malas que tiene JavaScript.

Definición

Las variables son espacios de memoria donde almacenamos temporalmente datos desde los que podemos acceder en cualquier momento de la ejecución de nuestros programas. Tienen varios tipos y clases que veremos a continuación.

Para definir una variable en JavaScript, utilizamos la palabra reservada `var` y le damos un nombre, por ejemplo:

```
var miDato;
```

También le podemos asignar un valor en la misma línea que la declaramos, por ejemplo, a continuación a la variable `dato` le asignamos el valor `5`:

```
var dato = 5;
```

O podemos primero declarar la variable y más adelante, en otra línea, asignarle un valor:

```
var dato;
dato = 5;
```

Debemos intentar que los nombres de las variables sean lo más descriptivos posibles, de manera que con solo leerlos sepamos que contienen y así nuestros desarrollos serán más ágiles.

Los nombres de las variables siempre han de comenzar por una letra, el símbolo `$` o `_`, nunca pueden comenzar por números u otros caracteres especiales. JavaScript también distingue entre mayúsculas o minúsculas, por tanto no es lo mismo `miDato`

que `MiDato` o `miDato`, para JavaScript son nombres diferentes y las tratará de manera diferente.

Tipos

JavaScript tiene 4 tipos primitivos de datos para almacenar en variables. Estos son:

- number
- boolean
- string
- undefined

number

Sirve para almacenar valores numéricos. Son utilizados para contar, hacer cálculos y comparaciones. Estos son algunos ejemplos:

```
var miEntero = 1;
var miDecimal = 1.33;
```

boolean

Este tipo de dato almacena un bit que indica `true` o `false`. Los valores **booleanos** se utilizan para indicar estados. Por ejemplo, asignamos a una variable el estado `false` al inicio de una operación, y al finalizarla lo cambiamos a `true`. Después realizamos la comprobación necesaria.

```
var si = true;
var no = false;
```

string

Las variables de tipo *string* almacenan caracteres o palabras. Se delimitan entre comillas simples o dobles. Ejemplo:

```
var dato = "Esto es un string";
var otroDato = 'Esto es otro string';
```

undefined

Este tipo se utiliza cuando el valor de una variable no ha sido definido aún o no existe. Por ejemplo:

```
var dato; // su valor es undefined
var dato = undefined;
```

Otro tipo de almacenamiento de datos que tiene JavaScript son los Objetos. En JavaScript todo es un objeto, hasta las funciones. Todo *hereda* de la clase `Object`. Se pueden definir como una estructura donde se agregan valores. Dentro de las clases que heredan de `Object` tenemos `Array`, `Date`, etc... que veremos más adelante.

Operadores

Operadores aritméticos

JavaScript posee operadores para tipos y objetos. Estos operadores permiten formar expresiones. Las más comunes son las operaciones aritméticas.

- **Suma de números**: `5 + 2`
- **Resta**: `5 - 2`
- **Operaciones con paréntesis**: `(3 + 2) - 5`
- **Divisiones**: `3 / 3`
- **Multiplicaciones**: `6 * 3`

El operador suma `+` también puede usarse para concatenar *strings* de la siguiente manera: `"Hola " + "mundo" + "!"` tendrá como resultado `"Hola mundo!"`.

JavaScript también posee los operadores post y pre incremento y decremento que añaden uno o restan uno a la variable numérica en la que se aplican. Dependiendo si son pre o post, la variable es autoincrementada o decrementada antes o después de la sentencia. Veamos un ejemplo:

```
var x = 1;      // x=1
var y = ++x;    // x=2, y=2
var z = y++ + x;// x=2, y=3, z=4
```

Como vemos en el ejemplo, la sentencia `y = ++x` lo que hace es incrementar x, que valía 1 y pasa a tener el valor 2, y la asignación `y = ++x` hace que `y` valga lo que `x`, es decir 2.

En la última sentencia tenemos un postincremento, esto lo que hace es primero evaluar la expresión y después realizar el incremento. Es decir en el momento de la asignación `z = y++ + x`, `y` vale 2 y `x` también 2, por lo tanto `z` vale 4, y después de esta asignación `y` es incrementada pasando a tener el valor 3.

Operador `typeof`

El operador `typeof` es un operador especial que nos permite conocer el tipo que tiene la variable sobre la que operamos. Ejemplos:

```
typeof 5;         // number
typeof false;     // boolean
typeof "Carlos";  // string
typeof undefined; // undefined
```

Esto es muy util para conocer en un momento dado que tipo estamos utilizando y prevenir errores en el desarrollo.

Operadores booleanos

Los tipos booleanos sólo tienen dos valores posibles: `true` y `false` (Verdadero y Falso). Pero disponen de varios operadores que nos permiten transformar su valor.

Negación

Este operador convierte un valor booleando en su opuesto. Se representa con el signo `!`. Si se utiliza dos veces, nos devuelve el valor original.

```
!true = false
!false = true
!!true = true
```

Identidad o Igualdad

El operador de igualdad (o igualdad débil), se representa con `==` y el de identidad (o igualdad estricta), con `===`. Se recomienda el uso del operador de identidad (los 3 iguales) frente al de igualdad débil, ya que el coste de procesamiento de éste último es mucho mayor y sus resultados en ocasiones son impredecibles. Es una de las *partes malas* de JavaScript, pero si se tiene cuidado no tiene por qué darnos ningún problema.

La desigualdad estricta se representa con `!==`.

```
true === true   // true
true === false  // false
true !== false  // true
true !== true   // false
```

Comparación

Podemos comparar si dos valores son menores, mayores o iguales con los operadores de comparación representados por los símbolos `<`, `>`, `<=` y `>=`. El resultado de la comparación nos devuelve `true` o `false` dependiendo de si es correcto o no.

```
5 > 3   // true
5 < 3   // false
3 >= 3  // true
2 <= 1  // false
"a" < "b" // true
```

Aunque se pueden utilizar comparaciones entre booleanos, *strings* y objetos se recomienda no usarlos ya que el orden que siguen no es muy intuitivo.

Operadores lógicos

Operador AND

Es un operador lógico que devuelve `true` siempre que todos los valores comparados sean `true`. Si uno de ellos es `false`, devuelve `false`. Se representa con el símbolo `&&`. Veamos un ejemplo

```
true && true    // true
true && false   // false
false && true   // false
false && false  // false
```

Es muy utilizado para devolver valores sin que estos sean modificados, por ejemplo para comprobar si una propiedad existe, etc. La lógica que sigue es: Si el primer valor es `false` devuelve ese valor, si no, devuelve el segundo:

```
0 && true
// 0, porque el número 0 se considera
// un valor "false"
1 && "Hola"
// "Hola", porque el número 1 (o distinto de 0)
// se considera un valor "true"
```

En el ejemplo comparamos 0 y `true`, como 0 es un valor que retorna `false`, nos devuelve ese valor. En el segundo ejemplo 1 es un valor que retorna `true`, por lo que nos devolverá el segundo `"Hola"`.

Valores que devuelven `false`.

Hay ciertos valores en JavaScript que son evaluados como `false` y son: el número `0`, un *string* vacío `""`, el valor `null`, el valor `undefined` y `NaN`.

Operador OR

Es otro operador lógico que funciona a la inversa que AND. Devuelve `false` si los valores comparados son `false`. En el caso de que un valor sea `true` devolverá `true`. Se representa con el símbolo `||`.

```
true || true    // true
true || false   // true
false || true   // true
false || false  // false
```

También es muy utilizado para asignar valores por defecto en nuestras funciones. La lógica que sigue es: Si el primer valor es true, devuelve ese valor. Por ejemplo:

```
var port = process.env.PORT || 5000;
```

En este ejemplo, la variable `port` contendrá el valor de `process.env.PORT` siempre que esa variable esté definida, si no su valor será 5000.

Condicionales

Los condicionales son expresiones que nos permiten ejecutar una secuencia de instrucciones u otra diferente dependiendo de lo que estemos comprobando. Permiten establecer el flujo de ejecución de los programas de acuerdo a determinados estados.

Asignación condicional

Este tipo de asignación es también conocido como el *If simplificado* u *operador ternario*. un tipo de condicional que veremos más adelante. Sirve para asignar en una sola línea un valor determinado si la condición que se evalúa es `true` u otro si es `false`. La sintaxis es la siguiente:

```
condición ? valor_si_true : valor_si_false
```

Si la condición devuelve `true`, retornará el valor de `valor_si_true`, y si es `false` el valor devuelto será el de `valor_si_false`. Veamos unos ejemplos:

```
(true) 5 : 2; // Devuelve 5
(false) 5 : 2; // Devuelve 2
```

Sentencia IF

Como hemos visto antes, dependiendo del resultado de una condición, obtenemos un valor u otro. Si el resultado de la condición requiere más pasos, en lugar de utilizar la asignación condicional, es mejor emplear la sentencia `if`. Tenemos 3 formas de aplicarlo:

if simple

```
if (condicion) {
  bloque_de_codigo
}
```

Si se cumple la condición dentro del paréntesis, se ejecuta el bloque de código incluido entre las llaves `{ ... }`

if/else

```
if (condicion) {
  bloque_de_codigo_1
}
else {
  bloque_de_codigo_2
}
```

Con este tipo de sentencia, si se cumple la condición pasa como el anterior modelo, se ejecuta el bloque de código 1, y si la condición a evaluar no se cumple, se ejecuta el bloque de código 2.

if/else if

Y por último si queremos realizar varias comprobaciones, podemos concatenar varias sentencias if, else if, etc.. y se irán comprobando en orden:

```
if (condicion_1) {
  bloque_1
}
else if (condicion_2) {
  bloque_2
}
else if (condicion_3) {
  bloque_3
}
else {
  bloque_4
}
```

En el ejemplo anterior, se comprueba la condición 1, si se cumple se ejecuta el bloque 1 y si no, se comprueba si cumple la condición 2 y en ese caso se ejecutaría el bloque 2, y así sucesivamente hasta que encuentre una condición que se cumpla o se ejecute el bloque 4 del último `else`.

Sentencia Switch

Con Switch, podemos sustituir un conjunto de sentencias `if-else` de una manera más legible. Se comprueba la condición, y según el caso que devuelva, ejecutará un bloque u otro. Para poder separar los bloques, se utiliza la palabra `break` que permite salir de toda la sentencia. Tiene un bloque `default` que se ejecuta en el caso de que no se cumpla ningún caso. Veamos un ejemplo, esto sería un `switch` siguiendo el ejemplo anterior del `if-else`:

```
switch(condicion) {
  case condicion_1:
    bloque_1
    break;
  case condicion_2:
    bloque_2
    break;
  case condicion_3:
    bloque_3
    break;
  default:
    bloque_4
}
```

El bloque `default` no es obligatorio.

Clases Core y Módulos de JavaScript

Además de los tipos primitivos que vimos al principio de este libro, JavaScript tiene unas clases, llamadas `Core` que forman parte del lenguaje. Las que más se utilizan son `Object`, `Number`, `Array` y `String`. Todas ellan heradan de `Object`.

Object

Un objeto es una colección de variables y funciones agrupadas de manera estructural. A las variables definidas dentro de un objeto se las denomina propiedades, y las funciones, métodos. Veamos un ejemplo de objeto que recoje los datos de un libro:

```
var libroAngular = {
  titulo: 'Desarrollo web ágil con AngularJS',
  autor: 'Carlos Azaustre',
  paginas: 64,
  formatos: ["PDF", "ePub", "Mobi"],
  precio: 2.79,
  publicado: false
};
```

Como podemos ver, las propiedades son pares *clave-valor*, separados por comas, y podemos acceder a ellas de forma independiente de varias formas, con la notación punto o con la notación array:

```
libroAngular.titulo; // Desarrollo web ágil con AngularJS
libroAngular['paginas']; // 64
```

También podemos modificarlas de la misma manera:

```
libroAngular.precio = 1.95;
libroAngular['publicado'] = true;
```

Con la notación array, podemos acceder a las propiedades con variables. Ejemplo:

```
var propiedad = "autor";
libroAngular[propiedad]; // "Carlos Azaustre"
```

Pero no funciona con la notación punto:

```
var propiedad = "autor";
libroAngular.propiedad; // undefined
```

Como hemos dicho antes, si el objeto contiene funciones se les llama métodos. En el siguiente capítulo veremos como se inicializan e invocan funciones más en detalle. Si queremos crearlas dentro de un objeto e invocarlas, sería así:

```
var libroAngular = {
  paginas: 64,
  leer: function () {
    console.log("He leído el libro de AngularJS");
  }
};

libroAngular.leer(); // Devuelve: "He leído el libro de AngularJS"
```

Para crear un objeto podemos hacerlo con la notación de llaves `{...}` o creando una nueva instancia de clase:

```
var miObjeto = { propiedad: "valor" };
var miObjeto = new Object({ propiedad: "valor" });
```

Anidación

Un objeto puede tener propiedades y estas propiedades tener en su interior más propiedades. Sería una representación en forma de árbol y podemos acceder a sus propiedades de la siguiente manera:

```
var libro = {
  titulo: "Desarrollo Web ágil con AngularJS",
  autor: {
    nombre: "Carlos Azaustre",
    nacionalidad: "Española",
    edad: 30,
    contacto: {
      email: "carlosazaustre@gmail.com",
      twitter: "@carlosazaustre"
    }
  },
  editorial: {
    nombre: "carlosazaustre.es Books",
    web: "https://carlosazaustre.es"
  }
};
// Podemos acceder con notación punto, array, o mixto.
libro.autor.nombre; // "Carlos Azaustre"
libro['autor']['edad']; // 30
libro['editorial'].web; // "https://carlosazaustre.es"
libro.autor['contacto'].twitter; // "@carlosazaustre
```

Igualdad entre objetos

Para que dos objetos sean iguales al compararlos, deben tener la misma referencia. Debemos para ello utilizar el operador identidad `===`. Si creamos dos objetos con el mismo contenido, no serán iguales a menos que compartan la referencia. Veamos un ejemplo:

```
var coche1 = { marca: "Ford", modelo: "Focus" };
var coche2 = { marca: "Ford", modelo: "Focus"};
coche1 === coche2; // Devuelve false, no comparten referencia
coche1.modelo === coche2.modelo; // Devuelve true porque el valor es el mismo.
var coche3 = coche1;
coche1 === coche3; // Devuelve true, comparten referencia
```

Number

Es la clase del tipo primitivo `number`. Se codifican en formato de coma flotante con doble precisión (Es decir, con 64 bits / 8 bytes) y podemos representar números enteros, decimales, hexadecimales, y en coma flotante. Veamos unos ejemplos:

```
// Número entero, 25
25
// Número entero, 25.5. Los decimales se separan de la parte entera con punto `.`
25.5
// Número hexadecimal, se representa con 0x seguido del número hexadecimal
0x1F  // 31 decimal
0xFF  // 255 decimal
0x7DE // 2014
// Coma flotante, separamos la mantisa del exponente con la letra `e`
5.4e2 // Representa 5.4 * 10 elevado a 2 = 540
```

La clase `Number` incluye los números `Infinity` y `-Infinity` para representar números muy grandes:

```
1/0 = Infinty
-1/0 = -Infinity
1e1000 = Infinity
-1e1000 = -Infinity
```

El rango real de números sobre el que podemos operar es ~1,797x10^308 --- 5x10^-324`.

También disponemos del valor `NaN` (*Not A Number*) para indicar que un determinado valor no representa un número:

```
"a"/15 = NaN
```

Para crear un número podemos hacerlo con la forma primitiva o con la clase `Number`. Por simplicidad se utiliza la forma primitiva.

```
var numero = 6;
var numero = new Number(6);
```

Funciones de Number

JavaScript tiene 2 funciones interesantes para convertir un string en su número equivalente.

`parseInt()`

Devuelve el número decimal equivalente al string que se pasa por parámetro. Si se le indica la base, lo transforma en el valor correspondiente en esa base, si no, lo devuelve en base 10 por defecto. Veamos unos ejemplos:

```
parseInt("1111");     // Devuelve 1111
parseInt("1111", 2);  // Devuelve 15
parseInt("1111", 16); // Devuelve 4369
```

`parseFloat()`

Función similar a `parseInt()` que analiza si es un número de coma flotante y devuelve su representación decimal

```
parseFloat("5e3");    // Devuelve 5000
```

`number.toFixed(x)`

Devuelve un string con el valor del numero `number` redondeado al alza, con tantos decimales como se indique en el parámetro `x`.

```
var n = 2.5674;
n.toFixed(0); // Devuelve "3"
n.toFixed(2); // Devuelve "2.57"
```

`number.toExponential(x)`

Devuelve un string redondeando la base o mantisa a `x` decimales. Es la función complementaria a `parseFloat`

```
var n = 2.5674;
n.toExponential(2); // Devuelve "2.56e+0"
```

`number.toString(base)`

Devuelve un string con el número equivalente `number` en la base que se pasa por parámetro. Es la función complementaria a `parseInt`

```
(1111).toString(10); // Devuelve "1111"
(15).toString(2); // Devuelve "1111"
(4369).toString(16) // Devuelve "1111"
```

Módulo Math

`Math` es una clase propia de JavaScript que contiene varios valores y funciones que nos permiten realizar operaciones matemáticas. Estos son los más utilizados:

```
Math.PI // Número Pi = 3.14159265...
Math.E  // Número e = 2.7182818...
Math.random() // Número aleatorio entre 0 y 1, ej: 0.45673858
Math.pow(2,6) // Potencia de 2 elevado a 6 = 64;
Math.sqrt(4)  // raiz cuadrada de 4 = 2
Math.min(4,3,1) // Devuelve el mínimo del conjunto de números = 1
Math.max(4,3,1) // Devuelve el máximo del conjunto de números = 4
Math.floor(6.4) // Devuelve la parte entera más próxima por debajo, en este caso 6
Math.ceil(6.4)  // Devuelve la parte entera más próxima por encima, en este caso 7
Math.round(6.4) // Redondea a la parte entera más próxima, en este caso 6
Math.abs(x);  // Devuelve el valor absoluto de un número

// Funciones trigonométricas
Math.sin(x);  // Función seno de un valor
Math.cos(x);  // Función coseno de un valor
Math.tan(x);  // Función tangente de un valor
Math.log(x);  // Función logaritmo
...
```

Existen muchos más, puedes consultarlo en la documentación de Mozilla: link

Array

Es una colección de datos que pueden ser números, strings, objetos, otros arrays, etc... Se puede crear de dos formas con el literal `[...]` o creando una nueva instancia de la clase Array

```
var miArray = [];
var miArray = new Array();
```

```
var miArray = [1, 2, 3, 4]; // Array de números
var miArray = ["Hola", "que", "tal"]; // Array de Strings
var miArray = [ {propiedad: "valor1" }, { propiedad: "valor2" }]; // Array de objetos
var miArray = [[2, 4], [3, 6]]; // Array de arrays, (Matriz);
var miArray = [1, true, [3,2], "Hola", {clave: "valor"}]; // Array mixto
```

Se puede acceder a los elementos del array a través de su índice y con `length` conocer su longitud.

```
var miArray = ["uno", "dos", "tres"];
miArray[1]; // Devuelve: "dos"
miArray.length; // Devuelve 3
```

Si accedemos a una posición que no existe en el array, nos devuelve `undefined`.

```
miArray[8]; // undefined
```

Métodos

`Array` es una clase de JavaScript, por tanto los objetos creados a partir de esta clase heredan todos los métodos de la clase padre. Los más utilizados son:

```
var miArray = [3, 6, 1, 4];
miArray.sort(); // Devuelve un nuevo array con los valores ordenados: [1, 3, 4, 6]
miArray.pop(); // Devuelve el último elemento del array y lo saca. Devuelve 6 y miArr
miArray.push(2); // Inserta un nuevo elemento en el array, devuelve la nueva longitud
miArray.reverse(); // Invierte el array, [2,4,3,1]
```

Otro método muy útil es `join()` sirve para crear un string con los elementos del array uniéndolos con el "separador" que le pasemos como parámetro a la función. Es muy usado para imprimir strings, sobre todo a la hora de crear templates. Ejemplo:

```
var valor = 3;
var template = [
  "<li>",
  valor
  "</li>"
].join("");

console.log(template); // Devuelve: "<li>3</li>"
```

Lo cual es mucho más eficiente en términos de procesamiento, que realizar lo siguiente, sobre todo si estas uniones se realizan dentro de bucles.

```
var valor = 3;
var template = "<li>" + valor + "</li>";
```

Si queremos aplicar una misma función a todos los elementos de un array podemos utilizar el método `map`. Imaginemos el siguiente array de números `[2, 4, 6, 8]` y queremos conocer la raíz cuadrada de cada uno de los elementos podríamos hacerlo así:

```
var miArray = [2, 4, 6 ,8];
var raices = miArray.map(Math.sqrt);
});

// raices: [ 1.4142135623730951, 2, 2.449489742783178, 2.8284271247461903 ]
```

O algo más específico:

```
var miArray = [2, 4, 6, 8];
var resultados = miArray.map(function(elemento) {
  return elemento *= 2;
});

// resultados: [ 4, 8, 12, 16 ]
```

Otra función interesante de los arrays es la función `filter`. Nos permite "filtrar" los elementos de un array dada una condición sin necesidad de crear un bucle (que veremos más adelante) para iterarlo. Por ejemplo, dado un array con los números del 1 al 15, obtener un array con los números que son divisibles por 3:

```
var miArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
var resultado = miArray.filter(function(elemento) {
  return elemento % 3 === 0;
});

// resultados: [ 3, 6, 9, 12, 15 ]
```

Si queremos obtener una parte del array, podemos emplear la función `slice` pasándole por parámetro el índice a partir del que queremos cortar y el final. Si no se indica el parámetro de fin, se hará el "corte" hasta el final del array, si no, se hará hasta la posición indicada y si se pasa un número negativo, contará desde el final del array hacia atrás.

El método devuelve un nuevo array sin transformar sobre el que se está invocando la función. Veamos unos ejemplos:

```
var miArray = [4, 8, 15, 16, 23, 42];
miArray.slice(2); // [15, 16, 23, 42]
miArray.slice(2, 4); // [15, 16] (la posición de fin no es inclusiva)
miArray.slice(2, -1); // [15, 16, 23]
miArray.slice(2, -2); // [15, 16]
```

String

Como vimos al principio de este libro, los strings son un tipo de variable primitivo en JavaScript, pero también, al igual que con `Number` tienen su clase propia y métodos.

Un string se comporta como un *Array*, no es más que un conjunto de caracteres, con índices que van desde el 0 para el primer carácter hasta el último. Veamos algunos ejemplos de cómo acceder a los caracteres y los métodos que posee esta clase

```
// Supongamos el string con el texto "javascript"
"javascript"[2] // Acceso como array, devuelve el tercer carácter "v", ya que la prim
"javascript".length() // Devuelve 10
"javascript".charCodeAt(2) // Devuelve el caracter en formato UNICODE de "v", el 118
"javascript".indexOf("script"); // Devuelve el índice donde comienza el string "scrip
"javascript".substring(4,10); // Devuelve la parte del string comprendida entre los i
```

Para crear un string podemos hacerlo con notación de tipo o creando un nuevo objeto. Por simplicidad se utiliza la forma primitiva.

```
var texto = "Hola Mundo";
var texto = new String("Hola Mundo");
```

Un string puede ser transformado en array con el método `split()` pasándole como parámetro el delimitador que queramos que separe los elementos. Por ejemplo:

```
var fecha = new Date();
fecha = fecha.toString(); // "Wed May 20 2015 20:16:25 GMT+0200 (CEST)"
fecha = fecha.split(" "); // ["Wed", "May", "20", "2015", "20:16:25", "GMT+0200", "(C
fecha[4]; // "20:16:25"
```

Funciones

Las funciones en JavaScript son bloques de código ejecutable, a los que podemos pasar parámetros y operar con ellos. Nos sirven para modular nuestros programas y estructurarlos en bloques que realicen una tarea concreta. De esta manera nuestro código es más legible y mantenible.

Las funciones normalmente, al acabar su ejecución devuelven un valor, que conseguimos con el parámetro `return`. Se declaran con la palabra reservada `function` y a continuación suelen llevar un nombre, para poder invocarlas más adelante. Si no llevan nombre se les llama funciones anónimas.

Veamos un ejemplo de función:

```
function saludar (nombre) {
  return ("Hola " + nombre + "!");
}

saludar("Carlos"); // Devuelve "Hola Carlos!"
```

La función del ejemplo se llama `saludar`, y se le pasa un único parámetro, entre paréntesis `(...)`, que es `nombre`. Ese parámetro funciona como contenedor de una variable que es utilizada dentro del bloque de código delimitado por las llaves `{...}`. El comando `return` devolverá el String que concatena texto con el valor que contiene el parámetro `nombre`.

Si no pasásemos ningún valor por parámetro, obtendríamos el valor `undefined`.

```
function saludar (nombre) {
  return ("Hola " + nombre + "!");
}

saludar(); // Devuelve "Hola undefined!"
```

También podemos acceder a los parámetros que se pasan por argumento a través del array `arguments` sin indicarlo en en la defínción de la función, aunque esta opción no es muy utilizada. Ejemplo:

```
function saludar () {
  var tipo = arguments[0];
  var nombre = arguments[1];
  return (tipo + ", " + nombre + "!");
}

saludar("Adios", "Carlos"); // Devuelve "Adios, Carlos!"
```

Parámetros por defecto

Una buena práctica para evitar errores o que se tome el valor `undefined` sin que podamos controlarlo, es utilizar algunos de los operadores booleanos que vimos en capítulos anteriores. Si tomamos el operador OR `||` podemos asignar un valor por defecto si no está definido. Veamos un ejemplo:

```
function saludar (tipo, nombre) {
  var tipo = tipo || "Hola";
  var nombre = nombre || "Carlos";
  return (tipo + ", " + nombre + "!");
}

saludar();                      // "Hola, Carlos!"
saludar("Adios");               // "Adios, Carlos!"
saludar("Hasta luego", "Pepe"); // "Hasta luego, Pepe!"
```

Ámbito de una función.

Por defecto, cuando declaramos una variable con `var` la estamos declarando de forma global y es accesible desde cualquier parte de nuestra aplicación. Tenemos que tener cuidado con los nombres que elegimos ya que si declaramos a una variable con el mismo nombre en varias partes de la aplicación estaremos sobrescribiendo su valor y podemos tener errores en su funcionamiento.

Si declaramos una variable dentro de una función, esta variable tendrá un ámbito local al ámbito de esa función, es decir, solo será accesible de la función hacia adentro. Pero si la definimos fuera de una función, tendrá un ámbito global.

En la versión 6 de ECMAScript tenemos los tipos de variable `let` y `const` en lugar de `var` y definen unos ámbitos específicos. `const` crea una constante cuyo valor no cambia durante el tiempo y `let` define el ámbito de la variable al ámbito donde ha sido

definida (por ejemplo en una función).

Con un ejemplo lo veremos más claro:

```
var valor = "global";

function funcionlocal () {
  var valor = "local";
  return valor;
}

console.log(valor);           // "global"
console.log(funcionLocal());  // "local"
console.log(valor);           // "global"
```

Aunque tenemos definida fuera de la función la variable `valor`, si dentro de la función la declaramos y cambiamos su valor, no afecta a la variable de fuera porque su ámbito (o *scope*) de ejecución es diferente. Una definición de variable local tapa a una global si tienen el mismo nombre.

Closures

Los *Closures* o funciones cierre son un patrón de diseño muy utilizado en JavaScript y son una de las llamadas *Good parts*. Para poder comprender su funcionamiento veamos primero unos conceptos.

Funciones como objetos

Las funciones en JavaScript son objetos, ya que todo en JavaScript es un objeto, heredan sus propiedades de la clase `Object`. Entonces pueden ser tratadas como tal. Podemos guardar una función en una variable y posteriormente invocarla con el operador paréntesis `()`. Ejemplo:

```
var saludar = function (nombre) {
  return "Hola " + nombre;
};

saludar("Carlos"); // "Hola Carlos"
```

Si a la variable que guarda la función no la invocamos con el operador paréntesis, el resultado que nos devolverá es el código de la función

```
saludar; // Devuelve 'function(nombre) { return "Hola " + nombre };'
```

Funciones anidadas

Las funciones pueden tener otras funciones dentro de ellas, produciendo nuevos ámbitos para las variables definidas dentro de cada una. Y para acceder desde el exterior a las funciones internas, tenemos que invocarlas con el operador doble paréntesis `()()`. Veamos un ejemplo

```
var a = "OLA";

function global () {
  var b = "K";

  function local () {
    var c = "ASE";
    return a + b + c;
  }

  return local;
}

global();   // Devuelve la función local: "function local() { var c ="ASE"..""
global()(); // Devuelve la ejecución de la función local: "OLAKASE"

var closure = global();
closure(); // Devuelve lo mismo que global()(): "OLAKASE"
```

Vistos estos conceptos ya podemos definir lo que es un `closure`.

Función cierre o closure

Un *Closure* es una función que encapsula una serie de variables y definiciones locales que únicamente serán accesibles si son devueltas con el operador `return`. JavaScript al no tener una definición de clases como tal (como por ejemplo en Java, aunque con la versión ECMAScript6 esto cambia un poco) este patrón de creación de closures, hace posible modularizar nuestro código y crear algo parecido a las clases.

Veamos un ejemplo de closure con la siguiente función. Creamos una función que tiene un variable local que guarda el valor de un numero que será incrementado o decrementado según llamemos a las funciones locales que se devuelven y acceden a

esa variable. la variable local `_contador` no puede ser accesible desde fuera si no es a través de esas funciones:

```
var miContador = (function () {
  var _contador = 0; // Por convención, a las variables "privadas" se las llama con u

  function incrementar () {
    return _contador++;
  }

  function decrementar () {
    return _contador--;
  }

  function valor () {
    return _contador;
  }

  return {
    incrementar: incrementar,
    decrementar: decrementar,
    valor: valor
  }
})();

miContador.valor(); // 0
miContador.incrementar();
miContador.incrementar();
miContador.valor(); // 2
miContador.decrementar();
miContador.valor(); // 1
```

Funciones como clases

Un closure es muy similar a una clase, la principal diferencia es que una clase tendrá un constructor que cumple el mismo cometido que el closure. Al crear un objeto a partir de una clase debemos usar el parámetro `new` y si es un closure, al inicializar un nuevo objeto, se le pasa lo que le devuelve la función cierre.

Veamos un ejemplo de la misma función, codificada como clase y como closure, y como se crearían sus objetos.

```
function inventario (nombre) {
  var _nombre = nombre;
  var _articulos = {};

  function add (nombre, cantidad) {
    _articulos[nombre] = cantidad;
  }

  function borrar (nombre) {
    delete _articulos[nombre];
  }

  function cantidad (nombre) {
    return _articulos[nombre];
  }

  function nombre () {
    return _nombre;
  }

  return {
    add: add,
    borrar: borrar,
    cantidad: cantidad,
    nombre: nombre
  }
}
```

Una vez construido la closure, podemos usar sus métodos como vemos a continuación:

```
var libros = inventario("libros");
libros.add("AngularJS", 3);
libros.add("JavaScript", 10);
libros.add("NodeJS", 5);
libros.cantidad("AngularJS"); // 3
libros.cantidad("JavaScript"); // 10
libros.borrar("JavaScript");
libros.cantidad("JavaScript"); // undefined
```

Ahora veamos como sería esto mismo pero codificado como Clase:

```
function Inventario (nombre) {
  this.nombre = nombre;
  this.articulos = [];

  this.add = function (nombre, cantidad) {
    this.articulos[nombre] = cantidad;
  }

  this.borrar = function (nombre) {
    delete this.articulos[nombre];
  }

  this.cantidad = function (nombre) {
    return this.articulos[nombre];
  }

  this.getNombre = function () {
    return this.nombre;
  }
}
```

Una vez definida la clase, crear objetos a partir de ella e invocar a sus métodos sería así:

```
var libros = new Inventario("Libros");
libros.add("AngularJS", 3);
libros.add("JavaScript", 10);
libros.add("NodeJS", 5);
libros.cantidad("AngularJS"); // 3
libros.cantidad("JavaScript"); // 10
libros.borrar("JavaScript");
libros.cantidad("JavaScript"); // undefined
```

Esta forma de codificar las funciones como clases se conoce como *Factory Pattern* o *Template functions*.

Uso de Prototype

Un problema importante que tiene este tipo de estructura, es que cuando creamos un nuevo objeto a partir de esta clase, reservará espacio en memoria para toda la clase incluyendo atributos y métodos. Con un objeto solo creado no supone mucha desventaja, pero imaginemos que creamos varios objetos:

```
var libros = new Inventario("Libros");
var discos = new Inventario("discos");
var juegos = new Inventario("juegos");
var comics = new Inventario("comics");
...
```

Esto supone que las funciones de la clase, `add`, `borrar`, `cantidad` y `getNombre` están siendo replicadas en memoria, lo que hace que sea ineficiente.

```
> libros
◀ ▼ Inventario {nombre: "libros", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "libros"
    ▶ __proto__: Inventario
> discos
◀ ▼ Inventario {nombre: "discos", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "discos"
    ▶ __proto__: Inventario
> comics
◀ ▼ Inventario {nombre: "comics", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "comics"
    ▶ __proto__: Inventario
> juegos
◀ ▼ Inventario {nombre: "juegos", articulos: Array[0]}
    ▶ add: function (nombre, cantidad)
    ▶ articulos: Array[0]
    ▶ borrar: function (nombre)
    ▶ cantidad: function (nombre)
    ▶ getNombre: function ()
      nombre: "juegos"
    ▶ __proto__: Inventario
>
```

Para solucionar esto podemos hacer uso del objeto `Prototype` que permite que objetos de la misma clase compartan métodos y no sean replicados en memoria de manera ineficiente. La forma correcta de implementar la clase `Inventario` sería la siguiente:

```
function Inventario (nombre) {
  this.nombre = nombre;
  this.articulos = [];
};

Inventario.prototype = {
  add: function (nombre, cantidad) {
    this.articulos[nombre] = cantidad;
  },

  borrar: function (nombre) {
    delete this.articulos[nombre];
  },

  cantidad: function (nombre) {
    return this.articulos[nombre];
  },

  getNombre: function () {
    return this.nombre;
  }
};
```

De esta manera, si queremos crear un nuevo objeto de la clase `Inventario` y usar sus métodos, lo podemos hacer como veníamos haciendo hasta ahora, sólo que internamente será más eficiente el uso de la memoria por parte de JavaScript y obtendremos una mejora en el rendimiento de nuestras aplicaciones.

Creando de nuevo los objetos `libros`, `discos`, `juegos` y `comics`, su espacio en memoria es menor (Ya no tienen replicados los métodos):


```
var libros = new Inventario('libros');
libros.getNombre();
libros.add("AngularJS", 3);
...
var comics = new Inventario('comics');
comics.add("The Walking Dead", 10);
...
```

Clases en ECMAScript 6

Con la llegada de la nueva versión del estándar de JavaScript (ECMAScript 6 o ECMAScript 2015) la definición de una función como clase ha cambiado. ES6 aporta un *azúcar sintáctico* para declarar una clase como en la mayoría de los lenguajes de programación orientados a objetos, pero por *debajo* sigue siendo una función prototipal.

El ejemplo anterior del `Inventario`, transformado a ES6 sería tal que así

```
class Inventario {
  constructor(nombre) {
    this.nombre = nombre;
    this.articulos = [];
  }

  add (nombre, cantidad) {
    this.articulos[nombre] = cantidad;
  }

  borrar (nombre) {
    delete this.articulos[nombre]
  }

  cantidad (nombre) {
    return this.articulos[nombre]
  }

  getNombre () {
    return this.nombre;
  }
}
```

Utilizando la palabra reservada `class` creamos una clase que sustituye a la función prototipal de la versión anterior.

El método especial `constructor` sería el que se definía en la función constructora anterior. Después los métodos `add`, `borrar`, `cantidad` y `getNombre` estarían dentro de la clase y sustituirían a las funciones prototipales de la versión ES5.

Su utilización es igual que en la versión anterior

```
var libros = new Inventario("Libros");

libros.add("AngularJS", 3);
libros.add("JavaScript", 10);
libros.add("NodeJS", 5);

libros.cantidad("AngularJS"); // 3
libros.cantidad("JavaScript"); // 10
libros.borrar("JavaScript");
libros.cantidad("JavaScript"); // undefined
```

Con esta nueva sintáxis podemos implementar herencia de una forma muy sencilla. Imagina que tienes una clase `Vehículo` de la siguiente manera:

```
class Vehiculo {
  constructor (tipo, nombre, ruedas) {
    this.tipo = tipo;
    this.nombre = nombre;
    this.ruedas = ruedas
  }

  getRuedas () {
    return this.ruedas
  }

  arrancar () {
    console.log(`Arrancando el ${this.nombre}`)
  }

  aparcar () {
    console.log(`Aparcando el ${this.nombre}`)
  }
}
```

Y quieres crear ahora una clase `Coche` que herede de vehículo para poder utilizar los métodos que esta tiene. Esto lo podemos hacer con la clase reservada `extends`

y con `super()` llamamos al constructor de la clase que hereda

```
class Coche extends Vehiculo {
  constructor (nombre) {
    super('coche', nombre, 4)
  }
}
```

Si ahora creamos un nuevo objeto `Coche` podemos utilizar los métodos de la clase `Vehiculo`

```
let fordFocus = new Coche('Ford Focus')
fordFocus.getRuedas() // 4
fordFocus.arrancar() // Arrancando el Ford Focus
```

Bucles

En ocasiones nos interesa que determinados bloques de código se ejecuten varias veces mientras se cumpla una condición. En ese caso tenemos los bucles para ayudarnos. Dependiendo de lo que necesitemos usaremos uno u otro. A continuación veremos cuales son y algunos ejemplos prácticos para conocer su uso.

Existen 3 elementos que controlan el flujo del bucle. La **inicialización** que fija los valores con los que iniciamos el bucle. La condición de **permanencia** en el bucle y la **actualización** de las variables de control al ejecutarse la iteración.

Bucle while

La sintaxis de un bucle `while` es la siguiente, el bloque de código dentro del `while` se ejecutará mientras se cumpla la condición.

```
var condicion; // Inicialización

while (condicion) { // Condición de permanencia
  bloque_de_codigo // Código a ejecutar y actualización de la variable de control
}
```

Por ejemplo si queremos mostrar por consola los números del 1 al 10, con un bucle while sería así:

```
var i = 1; // Inicialización
while (i < 11) { // Condición de permanencia
  console.log(i); // Código a ejecutar
  i++; // Actualización de la variable de control
}

// Devuelve: 1 2 3 4 5 6 7 8 9 10
```

Bucle Do/While

El bucle `do-while` es similar al `while` con la salvedad de que ejecutamos un bloque de codigo dentro de `do` por primera vez y despues se comprueba la condición de permanencia en el bucle. De esta manera nos aseguramos que al menos una vez el bloque se ejecute

```
var i = 1;
do {
  console.log(i);
  i++;
} while (i < 11);

// Devuelve: 1 2 3 4 5 6 7 8 9 10
```

Bucle For

Por último el bucle `for` es una sentencia especial pero muy utilizada y potente. Nos permite resumir en una línea la forma de un bucle while. Su sintaxis es la siguiente:

```
for(inicializacion; condición de permanencia; actualizacion) {
  bloque_de_codigo
}
```

Los elementos de control se definen entre los paréntesis `(...)` y se separan por punto y coma `;` . Los bucles anteriores en formato `for` serían así:

```
for (var i=1; i < 11; i++) {
  console.log(i);
}

// Devuelve: 1 2 3 4 5 6 7 8 9 10
```

Buenas prácticas en bucles For.

Es común que en nuestros desarrollos utilicemos éste tipo de bucle a menudo, por ejemplo para recorrer arrays. Si tomamos unas consideraciones en cuenta, podemos hacer programas más eficientes.

Por ejemplo, un bucle de este tipo:

```
var objeto = {
  unArray: new Array(10000);
};

for(var i=0; i<objeto.unArray.length; i++) {
  objeto.unArray[i] = "Hola!";
}
```

Tiene varios puntos, donde perdemos rendimiento. El primero de ellos es comprobar la longitud del array dentro de la definición del bucle `for`. Esto hace que en cada iteracción estemos comprobando la longitud y son pasos que podemos ahorrarnos y que harán más eficiente la ejecución. Lo ideal es *cachear* este valor en una variable, ya que no va a cambiar.

```
var longitud = objeto.unArray.length;
```

Y esta variable la podemos incluir en el bucle `for` en la inicialización, separada por una coma `,`, quedando de la siguiente manera:

```
for (var i=0, longitud=objeto.unArray.length; i<longitud; i++) {
  ...
}
```

Otra optimización que podemos hacer es *cachear* también el acceso al array dentro del objeto `grandesCitas`. De ésta manera nos ahorramos un paso en cada iteracción al acceder al interior del bucle. A la larga son varios milisegundos que salvamos y afecta al rendimiento:

```
var unArray = objeto.unArray;
for (var i=0, longitud=unArray.length; i<longitud; i++) {
  unArray[i] = "Hola!";
}
```

Si utilizamos los comandos `console.time` y `console.timeEnd` podemos ver cuanto tiempo llevó la ejecución:

```
> console.time('Test');
  for(var i=0; i<objeto.unArray.length; i++) {
    objeto.unArray[i] = "Hola";
  }
  console.timeEnd('Test');
  Test: 23.081ms
<· undefined
> console.time('Test');
  var unArray = objeto.unArray;
  for(var i=0, longitud=unArray.length; i<longitud; i++) {
    unArray[i] = "Hola";
  }
  console.timeEnd('Test');
  Test: 17.437ms
```

Como se puede comprobar, es más rápido de la segunda manera.

Bucle For/Each

Este tipo de bucle fue una novedad que introdujo ECMAScript5. Pertenece a las funciones de la clase `Array` y nos permite iterar dentro de un array de una manera secuencial. Veamos un ejemplo:

```
var miArray = [1, 2, 3, 4];
miArray.forEach(function (elemento, index) {
    console.log("El valor de la posición " + index + " es: " + elemento);
});

// Devuelve lo siguiente
// El valor de la posición 0 es: 1
// El valor de la posición 1 es: 2
// El valor de la posición 2 es: 3
// El valor de la posición 3 es: 4
```

¿Quieres ver una utilidad del `forEach`? Imagina que quieres recorrer los valores y propiedades de un objeto de este tipo:

```
var libro = {
  titulo: "Aprendiendo JavaScript",
  autor: "Carlos Azaustre",
  numPaginas: 64,
  editorial: "carlosazaustre.es",
  precio: "2.95"
};
```

Con `forEach` no puedes, porque es un método de la clase `Array`, por tanto necesitas que las propiedades del objeto sean un array. Esto lo puedes conseguir haciendo uso de las funciones de la clase `Object`: `getOwnPropertyNames` que devuelve un array con todas las propiedades del objeto y con `getOwnPropertyDescriptor` accedes al valor. Veamos un ejemplo:

```
var propiedades = Object.getOwnPropertyNames(libro);
propiedades.forEach(function(name) {
  var valor = Object.getOwnPropertyDescriptor(libro, name).value;
  console.log("La propiedad " +name+ " contiene: " +valor );
});

// Devuelve:
// La propiedad titulo contiene: Aprendiendo JavaScript
// La propiedad autor contiene: Carlos Azaustre
// La propiedad numPaginas contiene: 64
// La propiedad editorial contiene: carlosazaustre.es
// La propiedad precio contiene: 2.95
```

Bucle For/In

Además de `ForEach`, tenemos el bucle `ForIn` Con este tipo de bucle podemos iterar entre las propiedades de un objeto de una manera más sencilla que la vista anteriormente. La sintaxis es `for( key in object )` siendo `key` el nombre de la propiedad y `object[key]` el valor de la propiedad. Como siempre, veamos un ejemplo práctico:

```
var libro = {
  titulo: "Aprendiendo JavaScript",
  autor: "Carlos Azaustre",
  numPaginas: 64,
  editorial: "carlosazaustre.es",
  precio: "2.95"
};

for(var prop in libro) {
  console.log("La propiedad " +prop+ " contiene: " +libro[prop] );
}

// Devuelve:
// La propiedad titulo contiene: Aprendiendo JavaScript
// La propiedad autor contiene: Carlos Azaustre
// La propiedad numPaginas contiene: 64
// La propiedad editorial contiene: carlosazaustre.es
// La propiedad precio contiene: 2.95
```