JavaScript I

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos

gsyc-profes (arroba) gsyc.urjc.es

Enero de 2018

Contenidos

Introducción a JavaScript

JavaScript es un lenguaje de programación. Junto con HTML y CSS, es una de las principales tecnologías para presentar contenidos en la World Wide Web

- Creado por Brendan Eich, de Netscape, en 1995 como lenguaje de scripting para el navegador. Tardó 10 días en contruir el primer prototipo
- Está presente en el 100 % de los navegadores web modernos, donde no tiene rival

- El nombre *JavaScript* es poco afortunado. En su día se eligió por motivos de marqueting, para destacar que su sintaxis es similar a la de Java. Pero ahí acaba el parecido, es un lenguaje completamente distinto
- En 1996, Netscape encarga a Ecma International la normalización del lenguaje. La marca *java* pertenecía a Sun (luego a Oracle), así que el nombre formal del lenguaje se cambió a ECMAScript, aunque en la práctica lo normal es seguir llamándolo JavaScript

JavaScript Everywhere (1)

El éxito de internet lleva este lenguaje a ser masivamente utilizado, no solo en el navegador, se habla de *JavaScript everywhere*.
Aunque no fue inicialmente diseñado para esto, hoy puede usarse también en

- node.js
  Entorno de ejecución de JavaScript para el servidor.
- nw.js (antiguo node webkit)
  Electron (antiguo Atom Shell)
  Son entornos que permiten desarrollar aplicaciones nativas de escritorio mediante tecnologías web (JavaScript, html, css...)

JavaScript Everywhere (2)

- Mozilla Rhino. Implementación de JavaScript en java. Permite ejecutar código JavaScript fuera del navegador, en cualquier entorno donde esté disponible java
- Express.js
  Es un *Web Application Framework*, permite desarrollar aplicaciones web en el servidor. Basado en Node.js.
  Alternativa a Django o Ruby on Rails

Críticas a JavaScript

Es frecuente hacer críticas negativas a JavaScript, por diferentes motivos, algunos justificados, otros no tanto

- No es un lenguaje especialmente elegante, sobre todo las primeras versiones. Fue diseñado apresuradamente y eso se nota. Pero ha ido mejorando mucho con el tiempo
  - En JavaScript moderno, si el programador usa la técnicas adecuadas, se puede generar código de gran calidad
- Los primeros intérpretes eran lentos. Esto también ha mejorado mucho. Incluso hay subconjuntos estáticos de JavaScript como asm.js cuyos programas pueden ejecutarse al 70 % de la velocidad del código compilado en C++
  - Esto es muy adecuado para algoritmos que verdaderamente lo necesiten

- Todos los números son del mismo tipo: float
- La distinción entre los tipos *undefined* y *null* es bastante arbitraria
- Hasta ECMAScript 3 no tenía excepciones. Los programas fallaban silenciosamente
- Hasta ECMAScript 6, no tenía variables limitadas a un bloque, solo globales o limitadas a la función
- Hasta ECMAScript 6, no tenía soporte (nativo) para módulos
- Los números se representan como Binary Floating Point Arithmetic (IEEE 754). Esto tiene sus ventajas para trabajar con binarios, pero representa muy mal las fracciones decimales

```
> 0.3===0.3
true
> 0.1+0.2===0.3
false
> 0.3-(0.1+0.2)
-5.551115123125783e-17
```

- La barrera de entrada para empezar a programar en JavaScript es baja. Como *cualquiera puede programar en JavaScript*, el resultado es que *en JavaScript acaba programando cualquiera*. Esto es, hay mucho código de mala calidad
- Es orientado a objetos. Pero en las versiones anteriores a ECMAScript 6, solo admitía orientación a objetos basada en *prototipos*. Este modelo es distinto al de lenguajes como C++ o Java, que están basados en clases y herencia. Si el programador fuerza al lenguaje a un uso como el de C++ o Java, el resultado es antinatural, incómodo y problemático. *It's not a bug, it's a feature*
- ECMASCript 6 admite programación orientada a objetos basada en prototipos y programación orientada a objetos basada en clases

Versiones de JavaScript (1)

- Brendan Eich crea JavaScript. 1995
- ECMAScript 1. 1997. Primera versión normalizada
- ECMAScript 2. 1998. Pequeños cambios
- ECMAScript 3. 1999
  do-while, regexp, excepciones, mejor tratamiento de cadenas (entre otros)
- ECMAScript 4.
  Abandonado en 2008, por falta de acuerdo sobre si las mejoras deberían ser más o menos drásticas

Versiones de JavaScript (2)

- ECMAScript 5. Año 2009. Modo strict, nuevos arrays, soporte JSON (entre otros)
- ECMAScript 6. Año 2015
  Cambios muy relevantes: módulos, orientación a objetos basada en clases, parámetros opcionales en funciones, variables locales a un bloque
  - En el año 2015 los navegadores en general no soportaban ECMAScript 6, era necesario *transpilar* el código a ECMAScript 5.
  - En la actualidad (año 2019) cualquier navegador medianamente actualizado lo soporta. Con alguna excepción, por ejemplo el uso de módulos. La necesidad del transpilador es cada vez menor

Características de JavaScript

- Muy integrado con internet y el web
- La práctica totalidad de las herramientas necesarias para su uso son software libre
- El lenguaje no especifica si es interpretado o compilado, eso depende de la implementación
  - Técnicas modernas como la compilación JIT (*Just In Time*) y el uso de bytecodes hacen que la división entre compiladores e intérpretes resulta difusa
  - Podemos considerarlo un híbrido. Los *script engines* (motores) de JavaScript modernos tienden a ser más compilados que las primeras versiones
  - Se acerca más a un lenguaje interpretado: el motor necesita estar siempre presente, la compilación se hace en cada ejecución y siempre se distribuye el fuente y solo el fuente

- Es dinámico. Los objetos se crean sobre la marcha, sin definir una clase. A los objetos se les puede añadir propiedades en tiempo de ejecución
- Es dinámicamente tipado. El tipo de las variables y objetos puede cambiar en tiempo de ejecución
- Multiparadigma, admite los paradigmas de programación:
  - Imperativa
  - Funcional
  - Basada en eventos (*event-driven*)
  - Orientada a objetos basada en prototipos
  - Desde ECMAScript 6, orientada a objetos basada en clases (orientación a objetos *tradicional*)

Holamundo

JavaScript no tiene una forma nativa de mostrar texto, emplea distintos objetos, dependiendo de en qué entorno se ejecute

- En el navegador puede escribir HTML mediante `document.write()`
- Puede usar `console.log()`
  - En el navegador el texto saldrá por una consola (del propio navegador)
  - En node.js, por la salida estándar
- Puede abrir una ventana con `window.alert()`

Holamundo en HTML, incrustrado

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Hola, mundo</title>
   </head>
   <body>
      <script>
      document.write("Hola, mundo");
      </script>
   </body>
</html>
```

El elemento `<script>` puede aparecer 1 o más veces, tanto en la cabecera como en el cuerpo del documento HTML

`http://ortuno.es/holamundo01.html`

Holamundo en HTML, fichero externo

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Hola, mundo</title>
      <script src="js/holamundo.js"></script>
   </head>
   <body>
   </body>
</html>
```

holamundo.js:

```
console.log("Hola, mundo");
```

http://ortuno.es/holamundo02.html

Si la codificación del script es diferente de la codificación del fichero HTML, se indica con un atributo `charset` en el elemento `<script>`

Holamundo mínimo aceptado

Apurando la norma de HTML, pueden incluso omitirse los elementos <html>, <body> y <head>. Se consideran entonces sobreentendidos, el siguiente ejemplo también sería válido, aunque no recomendable en absoluto

```
<!DOCTYPE html>
  <meta charset="utf-8">
  <title>Hola, mundo</title>
  <script  src="js/holamundo.js"></script>
```

http://ortuno.es/holamundo03.html

node.js

- El entorno Node.js permite usar JavaScript como un lenguaje de programación en el servidor o en la consola
- También es útil para desarrollar código que luego vaya a ejecutarse en el navegador

¿Donde colocar el código?

- Nunca es recomendable incrustar el JavaScript dentro del HTML, excepto tal vez para páginas muy sencillas
- Un defecto muy habitual es organizar el código de la lógica de negocio en función de las *pantallas*, (aunque sea en un fichero externo)
- O peor aún: repartirlo por los botones y formularios

Sugerencia: desarrolla la lógica de negocio en Node.js, luego intégralo en el HTML

- Excepto tal vez cosas muy sencillas

¿nodejs o node?

El intérprete de Node.js en principio se llama `node`

- En Linux
  - Este nombre ya estaba ocupado por otro programa. Así que las distribuciones Linux lo renombran a `nodejs`
  - Si el *otro* node no está instalado, normalmente `/usr/bin/node` es un enlace a `/usr/bin/nodejs`
    Por tanto, podemos usar indistintamente cualquiera de las dos formas
  - En resumen: según esté configurado nuestro Linux, el intérprete será `node`, `nodejs` o ambos indistintamente
- En macOS
  Generalmente se mantiene el nombre `node`

Entorno Linux

- Instalación

  ```
  apt-get install nodejs
  ```

- Ejecución

  ```
  nodejs holamundo.js
  ```

  O bien

  ```
  node holamundo.js
  ```

- Si el script es solamente para el servidor o el terminal y no para el navegador, también podemos ejecutarlo directamente

  ```
  jperez@alpha:~$ ./holamundo.js
  ```

  Para ello escribimos en la primera línea el siguiente comentario

  ```
  #!/usr/bin/nodejs
  ```

  O bien

  ```
  #!/usr/bin/env nodejs
  ```

Entorno MacOS

- Podemos ejecutar node y pasarle como primer argumento el nombre del script

  ```
  node holamundo.js
  ```

- O podemos añadir la siguiente primera línea al script

  ```
  #!/usr/bin/env nodejs
  ```

Entorno del Navegador

- El código que vaya a ejecutarse en el navegador no puede empezar por `#!/usr/....`
  (la almohadilla normalmente no significa comentario en JavaScript)

Comentarios

Los comentarios se pueden indicar de dos formas

- `//Comentarios de una sola línea, con dos barras`

- ```
  /* Comentarios con barra y asterisco.
     Pueden ocupar varias líneas, pero no anidarse */
  ```
  ```

Sentencias y expresiones

En JavaScript hay

- Sentencias. *Hacen cosas*

  `x = x+1;`

  Un programa en JavaScript podemos considerarlo como una secuencia de sentencias (*statements*)

- Expresiones. *Devuelven valores*

  `x + 1`

  En cualquier lugar donde JavaScript espera una sentencia, también puede haber una expresión. Se denomina entonces *sentencia expresión*, (*expression statement*)

  - Con tal de que la expresión no empiece ni por llave ni por la palabra reservada *function*, porque esto provocaría ambiguedad con objetos y funciones
  - Donde se espera una expresión, no puede ir una sentencia

Uso del punto y coma

Un bloque es una secuencia de sentencias, entre llaves ({})

- Las sentencias acaban en punto y coma
- Excepto las sentencias que acaban en un bloque
  - En este caso también se puede añadir un punto y coma, que se considera una sentencia vacia
- Si el programador no incluye los puntos y coma, el parser los añade con la *automatic semicolon insertion*. De hecho el JavaScript moderno tiende a omitir los puntos y coma, lo que en ciertos casos puede producir errores y confusiones.

Aquí recomendamos incluir siempre punto y coma al final de cada sentencia

use strict

En ECMAScript 5 aparece el *modo estricto*
Consiste en una serie de restricciones que producen un código de más calidad, menos propenso a errores. En general debemos usarlo siempre, para ello basta poner como primera sentencia del script

```
'use strict'
```

Es una cadena, no una sentencia. Aparece entre comillas.
Si tenemos que mezclar nuestro código con código antiguo, incompatible con el modo estricto, entonces podemos aplicar este modo función a función

```
function f(){
    'use strict'
    ...
}
```

Requisitos del modo estricto

Las principales normas de este modo son:

- Es necesario declarar explícitamente todas las variables
- Las funciones se deben declarar en *top level* o como mucho con un nivel de anidamiento (una función dentro de otra función). Pero no se admiten niveles más profundos de anidamiento.
- No se puede repetir el nombre un parámetro en la misma función
- El intento de modificar propiedades inmutables genera una excepción
- No se permite el uso de la sentencia *with*
- Un número que comienza por 0 no se considera que es un número octal

Tipos de datos

En JavaScript hay dos tipos de valores

- *primitive values*:
  boolean, number, string, null, undefined
- Objetos
  Los principales son: *plain objects*, arrays, regexp

Booleanos

- `true`
  `false`

Números

- A diferencia de la mayoría de los lenguajes de programación, solo hay un tipo para todos los números, incluyendo enteros y reales

Strings (cadenas)

- Se puede usar la comila doble o la simple indistintamente (obviamente la comilla de apertura debe coincidir con la de cierre)

  ```
  'lorem'  "ipsum"
  ```

  Ya que HTML usa la comilla doble, es más habitual usar en JavaScript la comilla simple
  Excepto en JSON donde es obligatorio usar la cadena doble

En JavaScript hay dos tipos de datos para indicar que falta información

- *undefined*
  - Una variable no ha sido inicializada
  - Se ha llamado a una función sin especificar algún parametro
  - Se intenta leer una propiedad no existente de un objeto
- *null*

  Es un objeto que no tiene valor. Más o menos podríamos decir que es un objeto vacío (auque el verdadero objeto vacío es {})

La distinción entre undefined y null es algo arbitraria, en ocasiones puede aparecer cualquiera de los dos, así que es normal escribir cosas como

```
if (x===undefined || x===null){
}
```

Esto equivale a

```
if (!x) {
}
```

Aunque es menos claro, porque hay otros valores que también son considerados *false* (false, 0, NaN y la cadena vacía)

- El objeto vacío {} y el array vacío [] se consideran *cierto*

```
> var x
undefined
> typeof(x)
'undefined'
> x=null
null
> typeof(x)
'object'
```

Sería más razonable que el tipo de null fuera undefined, pero la primera implementación de JavaScript hacía esto (por error) y luego ya se tomó como norma

Conversión de tipos

- La función global `Number()` convierte una cadena en número. Devuelve `NaN` en caso de error
- La función global `String()` convierte un número en cadena

```
'use strict'

let x,y;
x=Number("  3 ");
console.log(x,typeof(x))  // 3  'number'
y=String(x)
console.log(y,typeof(y))  // 3  string

console.log(Number("23j"));  // NaN
```

Identificadores

Símbolos que nombran entidades del lenguaje: nombres de variables, de funciones, etc

- Deben empezar por letra unicode, barra baja o dólar. El segundo caracter y posteriores pueden ser cualquier carácter unicode
- Aunque los carecteres internacionales como eñes y tildes son fuentes potenciales de problemas: falta de soporte en el teclado del desarrollador, configuración del idioma en el sistema operativo, etc
- ¿Sensible a mayúsculas?
  - JavaScript: Sí
  - HTML: No
  - CSS: Sí

Para facilitar la difusión del software en proyecto reales, es recomendable el uso del inglés en el código fuente.
Obviamente el interfaz de usuario estará en español o en cualquier otro idioma conveniente para el usuario.

- Aunque aquí no lo haremos, la ventaja de un identificador en español, cuando estamos aprendiendo, es que queda claro que no es parte del lenguaje ni de ninguna librería estándar

Identificadores válidos:

- α `//Correcto, pero no recomendable`
  `alpha`
  `contraseña //Discutible`
  `$f //Discutible`
  `_valor`
  `x5`

Identificadores incorrectos:

- `5x`
  `#x`

Palabras reservadas

Las siguientes palabras tienen un significado especial y no son válidas como identificador:

```
abstract    arguments   await   boolean
break   byte    case    catch
char    class   const   continue
debugger    default delete  do
double  else    enum    eval
export  extends false   final
finally float   for function
goto    if  implements  import*
in  instanceof  int interface
let long    native  new
null    package private protected
public  return  short   static
super   switch  synchronized    this
throw   throws  transient   true
try typeof  var void
volatile    while   with    yield
```

Tampoco son identificadores válidos

```
Infinity NaN undefined
```

Números especiales

JavaScript define algunos valores numéricos especiales: `NaN` (Not a number), `Infinity`, `-Infinity`

```
'use strict'
let x,y;
x=1/0;
y= -1/0;
console.log(x);  // Infinity
console.log(y);   // -Infinity
console.log(typeof(x)); // number
console.log(typeof(y));  // number
console.log(typeof(NaN)) // number
```

Paradójicamente, `NaN` es un *number*

Operadores

Los principales operadores son

- Operadores aritméticos
  `+ - * / % ++ --`
- Operadores de asignación
  `= += -=`
- Operadores de cadenas
  `+ +=`

```
'use strict'
let x;
x=0;
++x;
console.log(x) // 1
x+=2;
console.log(x) // 3
--x;
console.log(x) // 2
x-=2;
console.log(x) // 0

x='hola'+'mundo'
console.log(x); // 'holamundo'
x+="!"
console.log(x); // 'holamundo!'
```

- JavaScript 1.0 solo incluía el *lenient equality operator*, comparador de igualdad tolerante
  `==` `!=`
  Hace conversión automática de tipos.

```
> '4'==4
true
```

  Esto produce muchos resultados problemáticos

```
> 0==false
true
> 1==true
true
> 2==false
false
> 2==true
false
> ''==0
true
> '\t123\n'==123
true
> 'true'==true
false
```

- Comparador de igualdad estricto.
  Aparece en JavaScript 1.3, es el que deberíamos usar siempre
  `===` `!==`
- Mayor y menor
  `>` `<` `>=` `<=`
- Operador condicional
  `condición? valor_si_cierto : valor_si_falso`

```
> edad=18
18
> (edad>17)? "mayor_de_edad":"menor"
'mayor_de_edad'
```

- Operadores lógicos
  `&&` `||` `!`

```
> !(true && false)
true
```

Funciones

Una función es una secuencia de instrucciones empaquetada como una unidad. Acepta 0 o más valores y devuelve 1 valor.
Las funciones en JavaScript pueden cumplir tres papeles distintos

- *Nonmethod functions*. Funciones *normales*, no son una propiedad de un objeto. Por convenio, su nombre empieza por letra minúscula
- *Constructor*. Sirven para crear objetos. Se invocan con el constructor `new`.
  Por convenio, su nombre empiezan por letra mayúscula
  `new Cliente()`
- Métodos. Funciones almacenadas como propiedad de un objeto

Declaración de funciones

Hay tres formas de declarar una función

- Mediante una declaración. Es la forma más habitual

```
function suma(x,y){
    return x+y;
}
```

- Mediante una expresión. Función anónima. Habitual por ejemplo en JQuery

```
function(x,y){
    return x+y;
}
```

- Mediante el constructor `Function()`. Crea la función a partir de una cadena. No recomendable.

```
new Function('x','y','return x+y');
```

Hoisting

JavaScript hace *hoisting* (elevación) con las funciones.
El motor de JavaScript mueve las declaración al principio del bloque,

```
'use strict'
console.log(f(1));   //2

function f(x){
   return x+1;
}
```

Paso por valor

En JavaScript, el paso de parámetros a una función es por valor (por copia). La función recibe una copia del valor del argumento. Si la función modifica este valor, el argumento original no queda modificado

```
'use strict'
function f(x){
    x = x + 1;
}
var a=0;
console.log(a);  // 0
f(a);
console.log(a);  // 0
```

Se puede simular el paso por referencia envolviendo el valor en un array

Valor devuelto

Una función siempre devuelve exactamente 1 valor. En caso de que la función no incluya la sentencia `return`, el valor es `undefined`

```
'use strict'

function f(){
}

console.log(f());   // undefined
```

Número de parámetros

Muchos lenguajes de programación obligan a que el número de parámetros en la declaración de una función sea igual al número de argumentos cuando se invoca.
JavaScript, no. Si faltan argumentos, se consideran `undefined` y si *sobran* se ignoran

```
'use strict'

function f(x,y){
    return x+y;
};
console.log(f(1));   // NaN
console.log(f(2,2));  // 4
console.log(f(1,1,1)); // 2
```

Valores por omisión

Para dar un valor por omisión a un parámetro omitido en la invocación de una función, podríamos hacer lo siguiente

```
'use strict'

function f(x){
    if (x===undefined) {
        x=0};
     return x + 1 ;
};
console.log(f()); //1
```

Aunque la forma habitual en JavaScript 5 y anteriores era esta otra:

```
'use strict'

function f(x){
    x = x || 0;   //  línea 4
    return x + 1;
};

console.log(f());   //1
```

Línea 4: Si x se omite, valdrá `undefined` y el operador || (or) devolverá el valor a su derecha

- En general deberíamos evitar construcciones rebuscadas y poco claras. Pero este caso concreto podemos considerarlo idiomático en JavaScript, resulta aceptable

En ECMAScript 6 es mucho más sencillo

```
function f(x=0){
    return x + 1;
}

console.log(f(2));  // 3
console.log(f());  // 1
```

Aunque en 2018 es habitual encontrarnos motores de JavaScript donde esto aún no está implementado

Ámbito de las variables

Ámbito (*scope*)
Zona del código donde una variable es accesible
Hay tres tipos de variables

- Globales
  Declaradas fuera de una función
- Locales
  Declaradas con `var`. O declaradas implícitamente (si no usamos el modo estricto)
- Locales, declaradas con `let`
  Aparecen en ECMAScript 6

Variables Globales

Son variables accesibles desde todo el script (dentro del mismo fichero .js o entre un par de etiquetas `<script>`)
En el caso de JavaScript incrustado en HTML, todo el código de la misma página HTML comparte el objeto Window y por tanto, las variables globales

- Algunas metodologías recomiendan que las variables globales se usen lo menos posible
- Otras, que no se usen nunca

```
'use strict'
var x=0;
function f(){
  x=3;
}

function g(){
  return(x)
}

f();
console.log(g());  //3
```

Variables locales con var

Las variables declaradas con `var` son locales a su función

- Esto incluye a las funciones anidadas dentro de la función

```
'use strict'
function f(){
  var x=0;
  g();
  console.log(x);  //0
}

function g(){
  var x=3;
}

f();
```

Variables locales con let

Las variables declaradas con `let`

- Son locales a su bloque
- Tienen el comportamiento habitual en la mayoría de los lenguajes de programación
- Aquí recomendaremos usar siempre `let` y no `var` a menos que
  - Queramos alguna variable global
  - Tengamos que programar en una versión antigua de JavaScript

```
'use strict'
function f() {
  var x = 1;
  if (true) {
    var x = 2;  // La misma variable
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function g() {
  let x = 1;
  if (true) {
    let x = 2;  // Variable diferente
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
f();
g();
```

En ECMAScript 5 y precedentes, para conseguir algo similar a esto se usaba un truco no muy elegante denominado IIFE (immediately invoked function expression)

- Consiste en declarar una función sin nombre, abrir y cerrar paréntesis a continuación para que se invoque inmediatamente y ponerlo todo entre paréntesis

```
(funtion() {
    }());
```

Condicional

La sentencia `if` funciona como en muchos otros lenguajes de programación

```
'use strict'
var x="ejemplo";
if (x.length < 4){
    console.log("Cadena muy corta");
};


if (2 > 0) {
    console.log("cierto");
}
else {
    console.log("falso");
};
```

Es recomendable usar siempre los bloques de sentencias (las llaves). Aunque si la sentencia es única, pueden omitirse

```
if (2 > 0) console.log("cierto");
else console.log("falso");
```

switch

Evalúa la expresión entre paréntesis después de `switch` y salta a la cláusula `case` cuyo valor coincida con la expresión. O a la cláusula `default` si ninguna coincide.

```
'use strict'

let y;
let x=":";
switch(x){
    case(';'):
        y="punto y coma";
        break;
    case(':'):
        y="dos puntos";
        break;
    default:
        y="caracter desconocido";
}
console.log(y);  // dos puntos
```

Después de cada `case` se indican una o más sentencias, lo habitual es que la última de ellas sea `break`

- También se puede concluir lanzando una excepción con `throw` o saliendo de la función `return`, aunque esto último no es recomendable

Si no se incluye ninguna sentencia de finalización, la ejecución continúa.

- Si esa es la intención del programador (y no un olvido), es recomendable indicarlo de forma explícita
- Tradicionalmente se usa la expresión `fall through` (*cae a través, pasa, se cuela*)

```
'use strict'

let x='ubuntu';
let so="";
switch(x){
    case('ubuntu'):
        //fall through
    case('debian'):
        //fall through
    case('fedora'):
        //fall through
    case('redhat'):
        so='linux';
        break;
    case('macos'):
        so="macos"
        break;
    default:
        so='no soportado';
}

console.log(so);
```

La expresión de cada case puede ser cualquiera:

```
'use strict'
function cuadrante(x,y){
    let r;
    switch(true){
        case( x>= 0 && y>=0):
            r=1;
            break;
        case( x< 0 && y>=0):
            r=2;
            break;
        case( x< 0 && y<0):
            r=3;
            break;
        case( x>= 0 && y<0):
            r=4;
            break;
        default:
            r=NaN;
    }
    return r;
}
console.log(cuadrante(1,-1));  // 4
```

while

```
'use strict'

var x=5;
var cadena="";

while(x>0){
    --x;
    cadena+="*"
}
console.log(cadena);  //*****


x=5;
cadena="";

while(true){
    if(x<1) break;
    --x;
    cadena+="*"
}
console.log(cadena);  //*****
```

for

La sentencia `for` también es como en C y muchos otros lenguajes

- Entre paréntesis y separado por punto y coma se indica la sentencia inicial, la condición de permanencia y la sentencia que se ejecuta después de cada ejecución del cuerpo
- A continuación, el bloque (o sentencia) a ejecutar

```
'use strict'
let cadena="";
for(let i=0; i<5; ++i){
    cadena+="*";
}
console.log(cadena);  //*****
```

Bucles sobre cadenas

- Podemos acceder a los caracteres individuales de una cadena mediante corchetes
- La primera posición es la 0
- La longitud de la cadena se puede consultar con la propiedad `length` de la cadena

```
'use strict'
let x;

x="Lorem Ipsum"

for (let i=0; i<x.length; ++i){
    console.log(x[i]);
}
```

JavaScript tiene una característica que puede ser fuente de problemas: si intentamos acceder a una propiedad inexistente de un objeto, simplemente devuelve `undefined`
Supongamos que, por error, escribamos `x.lengh` en vez de `x.length`

```
for (let i=0; i<x.lengh; ++i){   //ERROR!
    console.log(x[i]);
}
```

- En la primera iteración, la condición del bucle será `0 < undefined`
- Esto se evalúa como `false`
- El bucle concluye silenciosamente, sin generar ningún error

Generalmente esto es un comportamiento no deseado, puede resultar un error difícil de trazar

En ECMAScript 6 podemos recorrer una cadena de forma muy conveniente con for-of

```
'use strict'
let x="Lorem Ipsum";

for (let c of x){
    console.log(c);
};
```

Manipulación de cadenas

Las cadenas tienen diversos métodos que permiten su manipulación
Todos estos métodos devuelven una nueva cadena, dejando la original intacta

- `toUpperCase()` y `toLowerCase()` devuelven la cadena en mayúsculas/minúsculas

```
> 'contraseña'.toUpperCase()
'CONTRASEÑA'
> 'LoReM IPsum'.toLowerCase()
'lorem ipsum'
```

- El método `trim()` devuelve la cadena eliminando los espacios a la izquierda y a la derecha
  - Espacios en sentido amplio, incluye tabuladores y el caracter fin de línea

```
> '    ABC  '.trim()
'ABC'
```

- El método `indexOf()` devuelve la posición de la primera aparición de una subcadena. O el valor -1 si no está incluida

```
> '__abc'.indexOf('abc')
2
> '__abc'.indexOf('xxx')
-1
```

- `lastIndexOf()` devuelve la última aparición de una subcadena. O el valor -1 si no está incluida

```
> 'a.tar.gz'.lastIndexOf('.')
5
```

- `slice(x,y)` devuelve la subcadena comprendida entre la posición `x` (incluida) y la `y` (excluida)

```
> '0123'.slice(0,3)
'012'
```

- Si `x` o `y` exceden las dimensiones de la cadena, no es problema

```
> 'abc'.slice(0,7)
'abc'
> 'abc'.slice(-5,7)
'abc'
```

- Si `x` es mayor o igual que `y` , devuelve la cadena vacía

```
> 'abc'.slice(3,2)
''
> 'abc'.slice(2,2)
''
```

- El método `split(c)` trocea una cadena, usando el caracter `c` como separador. Devuelve un array

```
> "a,b,c".split(',')
[ 'a', 'b', 'c' ]
```

- El método `replace(x,y)` devuelve una cadena donde la subcadena `x` ha sido reemplazada por `y`

```
> 'color beige'.replace('beige','crema')
'color crema'
```

Arrays

Un array es un objeto donde se hace corresponder un número natural con un valor

- Se declara entre corchetes, en el interior habrá elementos, separados por comas
- A diferencia de otros lenguajes de programación, es una estructura dinámica: no es necesario fijar su tamaño a priori

```
'use strict'

let a,b,c,d;

a=[ ] ; //  array vacío
b=[7, 8, 9] // array con números
c=['rojo', 3, 0] // array con elementos heterogéneos
d=[ [0, 1], [1, 1]]  // array con arrays anidados
```

Los arrays en JavaScript tienen muchos métodos disponibles.
Mostramos algunos de los principales

- Atención: algunos son *destructivos*, esto es, modifican el array
- Otros, devuelven un array con la modificación requerida

```
'use strict'
let a,x;
a=['sota', 'caballo']

// Longitud del array
console.log(a.length);  // 2

// Acceso a un elemento individual
console.log(a[1]);  // caballo

// Añadir un elemento al final
a.push('rey');
console.log(a);   // [ 'sota', 'caballo', 'rey' ]

// Extraer un elemento al final
x=a.pop() //
console.log(x); // rey
console.log(a);   // [ 'sota', 'caballo']
```

```
// Extraer un elemento al principio
x=a.shift();
console.log(x); // sota
console.log(a);   // [ 'caballo']

// Añadir un elemento al principio
a.unshift('alfil');
console.log(a); // ['alfil', 'caballo']

// Añadir un elemento, creando huecos
a[3]="torre";
console.log(a);   // ['alfil', 'caballo', , 'torre']

// La propiedad length incluye los huecos
console.log(a.length);  // 4

// Truncar un array
a.length=0;
console.log(a);   //   []

a=['alfil', 'caballo', 'torre']
a.reverse();
console.log(a);   //  ['torre', 'caballo', 'alfil']
```

Hay diversas formas de recorrer un array

- Al estilo C

```
'use strict'
let l=["a",,"c"]
for(let i=0; i<l.length; ++i){
    console.log(l[i])
}
// a undefined c
```

También itera sobre los huecos

- Con el método `forEach`
  - Recibe una función, que se aplicará a cada elemento del array
  - Es habitual pasar una función anónima

```
'use strict'
let l=["a",,"c"]
l.forEach(function(x){
    console.log(x)
});
// a c
```

Se ignoran los huecos

- Especialmente conveniente es `for-of`, disponible en ECMAScript 6

```
'use strict'
let l=["a",,"c"]
for(let x of l){
    console.log(x)
}
// a undefined c
```

También itera sobre los huecos

No debemos usar for-in para recorrer un array, porque los arrays, además de índice, pueden tener otras propiedades que también se recorrerían

```
'use strict'
let a,x;
a=[7, 8];
a.color='azul'
for (x in a){
    console.log(x); // 0, 1, color
}
```

Los métodos `indexOf()` y `lastIndexOf()` se comportan de igual forma que sobre las cadenas

```
'use strict'
let a;
a=[7, 8, 9, 7];

console.log(a.indexOf(9)); // 2
console.log(a.indexOf(3)); // -1
console.log(a.lastIndexOf(7)) // 3
```

Objetos

Los objetos de JavaScript son similares a los diccionarios de otros lenguajes, son estructuras que tienen

- Valores, llamados propiedades.
  Pueden ser valores primitivos (booleano, número, cadena, null, undefined) o bien una función o bien otro objeto.
- Claves
  Cada valor está asociado a una clave. La clave es una cadena.

Los más sencillos son los objetos literales o *plain objects*.

- Se declaran entre llaves.
- Cada propiedad tiene la forma nombre, dos puntos, valor
- Las propiedades van separadas por comas. Desde JavaScript 5 se permite que la última propiedad también acabe en coma

```
'use strict'
let x={
    unidades:2,
    color:'verde',
    tamaño:'grande',
};
console.log(x); // { unidades: 2, color: 'verde', 'tamaño': 'grande' }
console.log(x.unidades);  // 2
console.log(x.precio); //undefined
```

Además de unar la notación
`objeto.clave`
se puede usar la notación
`objeto["clave"]`

- Tiene la ventaja de que permite usar claves calculadas

```
'use strict'
let p={ latitud:40.3355, longitud:-3.8773 };

console.log(p.latitud);  // 40.3355
console.log(p["latitud"]);  // 40.3355

let clave="latitud";
console.log(p[clave]);  // 40.3355
```

Podemos obtener la lista de claves de un objeto usando el método `keys()` del *built-in object* `Object`

`Object.keys(miObjeto)`

```
'use strict'
let p={ latitud:40.3355, longitud:-3.8773 };

let clave, claves;
claves=Object.keys(p) ;
console.log(claves); // [ 'latitud', 'longitud' ]

for (clave of claves){
    console.log(clave, ":", p[clave]);
}
/*
  latitud : 40.3355
  longitud : -3.8773
*/
```

Para saber si un objeto es un array, disponemos de la función `Array.isArray()`

```
'use strict'
let miObjeto={color:"verde"}
let miLista=[1,2]

console.log(typeof(miObjeto))  //object
console.log(typeof(miLista))  //object

console.log(Array.isArray(miObjeto))  //false
console.log(Array.isArray(miLista))   //true
```

Una función solo devuelve 1 argumento. Si necesitamos que devuelva más, podemos usar estos objetos

```
'use strict'
function f(x,y){
    let r={};
    r.suma=x+y;
    r.producto=x*y;
    return r;
};
console.log(f(2,3)); // { suma: 5, producto: 6 }
console.log(f(2,3).suma);   // 5
console.log(f(2,3).producto);  // 6
```

Referencias

- *Speaking JavaScript. An In-Depth Guide for Programmers*
  Axel Rauschmayer. O'Reilly Media, 2014

  `http://proquest.safaribooksonline.com/book/programming/javascript/9781449365028`

- *JavaScript: The Definitive Guide, 6th Edition*
  David Flanagan. O'Reilly Media, 2011

  `http://proquest.safaribooksonline.com/book/programming/javascript/9781449393854`