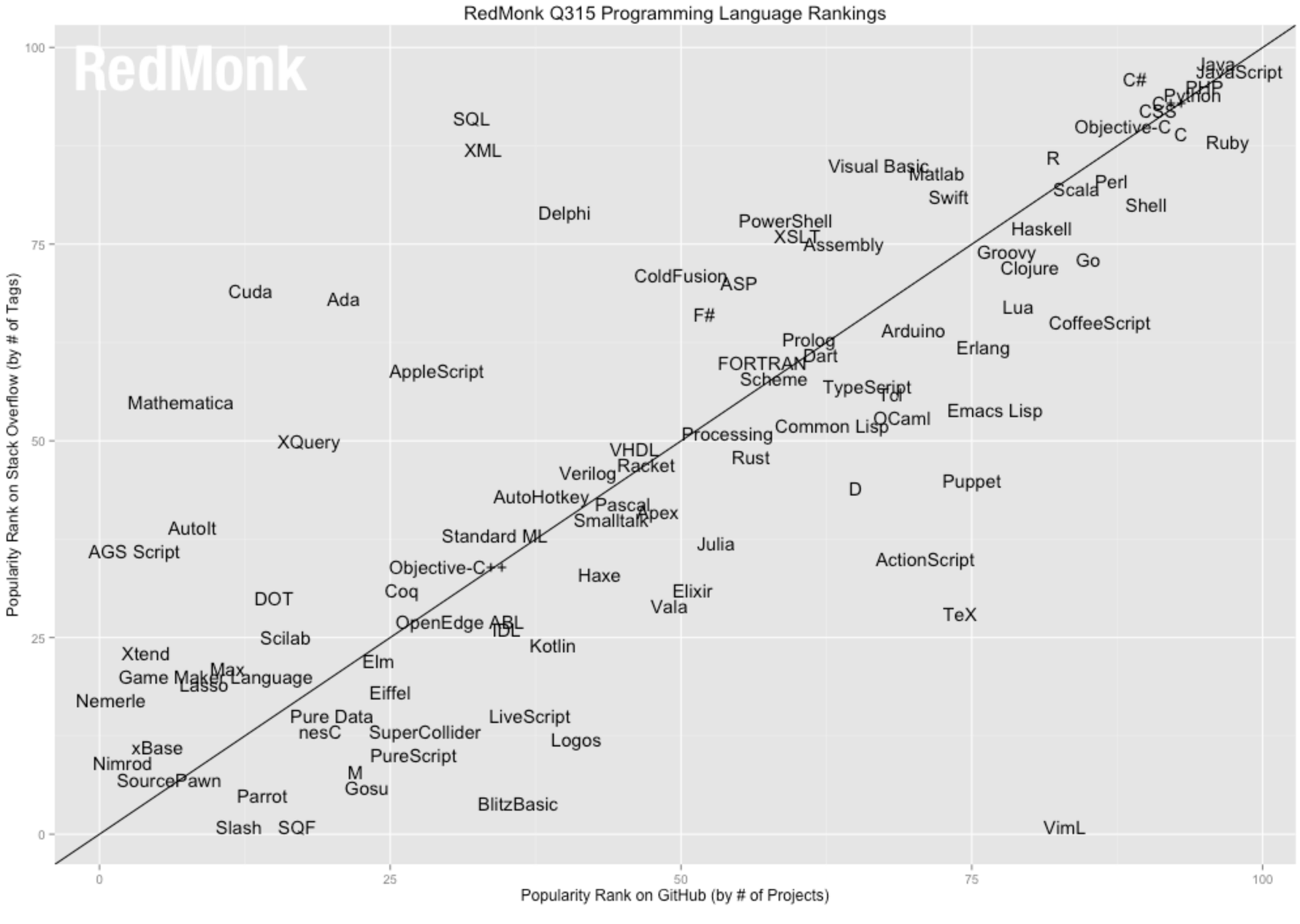
JavaScript

Sesión 1 - **JavaScript. El Lenguaje**

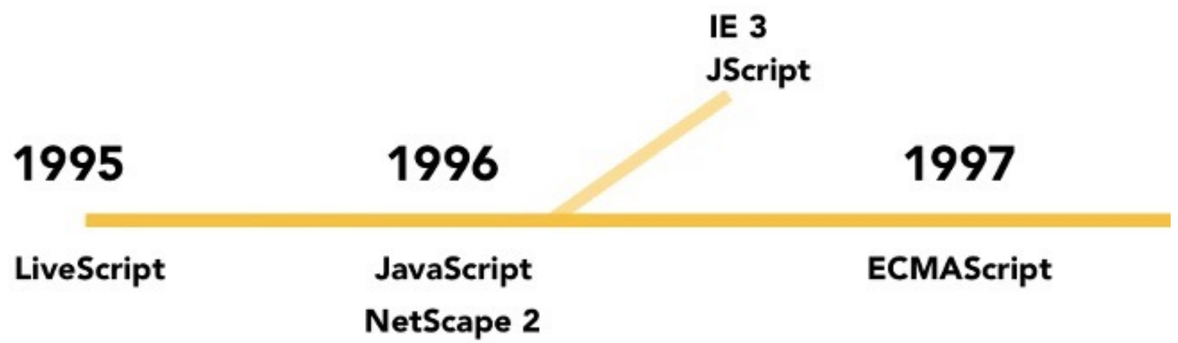
Índice

Situación Actual



Introducción

- *JavaScript* es un lenguaje de *script* de amplio uso en la web
  - También se utiliza en otros contextos, como en servidor→ *Node.js*
- Lenguaje con mala fama
  - Débilmente tipado
- *JavaScript* != *Java*
- Nombre similar por decisiones de *marketing*



ECMAScript

- Define el estándar de JavaScript, independiente de la implementación de cada navegador.

- ECMAScript 3 → 1999, implementada por todos los navegadores
- ECMAScript 4 → se desechó

- ECMAScript 5 → 2009 → añade al lenguaje algunos objetos, métodos y propiedades y el modo estricto (*strict mode*)
  - Es la especificación que cumplen todos los navegadores moderno

- ECMAScript 6 → 2015 → añade promesas

strict mode

- Elimina características del lenguaje
  - Simplifica los programas y reduce los errores
- Obliga a declarar las variables
- Evita parámetros repetidos, propiedades repetidas en un objeto, …

- `"use strict"`

```
function modoEstricto() {
  "use strict";
  // resto de la función
}
```

1.2 Uso en el Navegador

- Etiqueta `<script>`
  - Dentro del documento HTML, incluyendo el código entre la apertura y el cierre de la etiqueta

```
<script>
   // Instrucciones JavaScript
</script>
```

  - Referenciando a un fichero externo → atributos `async`, `defer`

```
<script src="ficheroJavascript.js"></script>
```

- Manejador de un evento → `onclick`, `onmouseover`

```
<button onclick="nombreFuncionJavaScript()" />
```

- Protocolo URL → `javascript:`

```
<a href="javascript:nombreFuncionJavaScript()">Validar</a>
```

Hola ExpertoJavaUA

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>Hola ExpertoJavaUA</title>
  <meta charset="utf-8" />
  <script>
    console.log("Hola ExpertoJavaUA desde la consola");
    alert("Hola ExpertoJavaUA desde alert");
  </script>
</head>
<body></body>
</html>
```

1.3 Herramientas

- Intellij IDEA

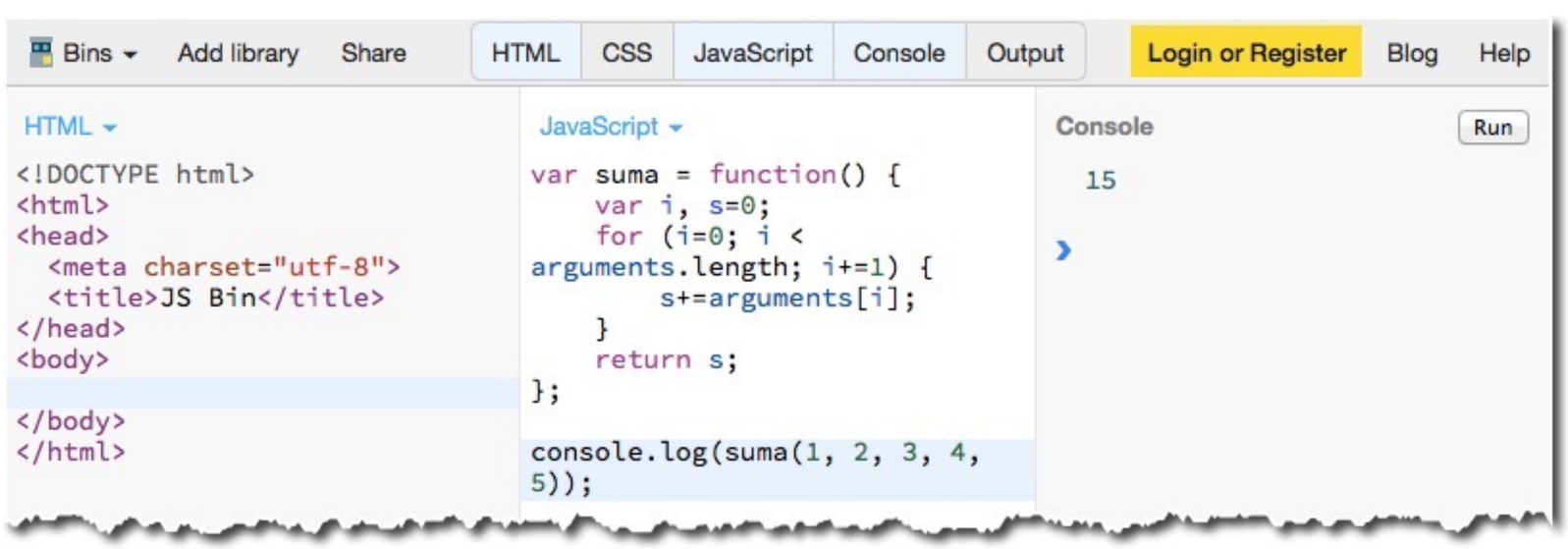
- Mozilla Developer Network (MDN): https://developer.mozilla.org/en/docs/Web/JavaScript/Reference
  - https://developer.mozilla.org/es/docs/JavaScript/Referencia

- JSBin: jsbin.com
- JSHint: jshint.com

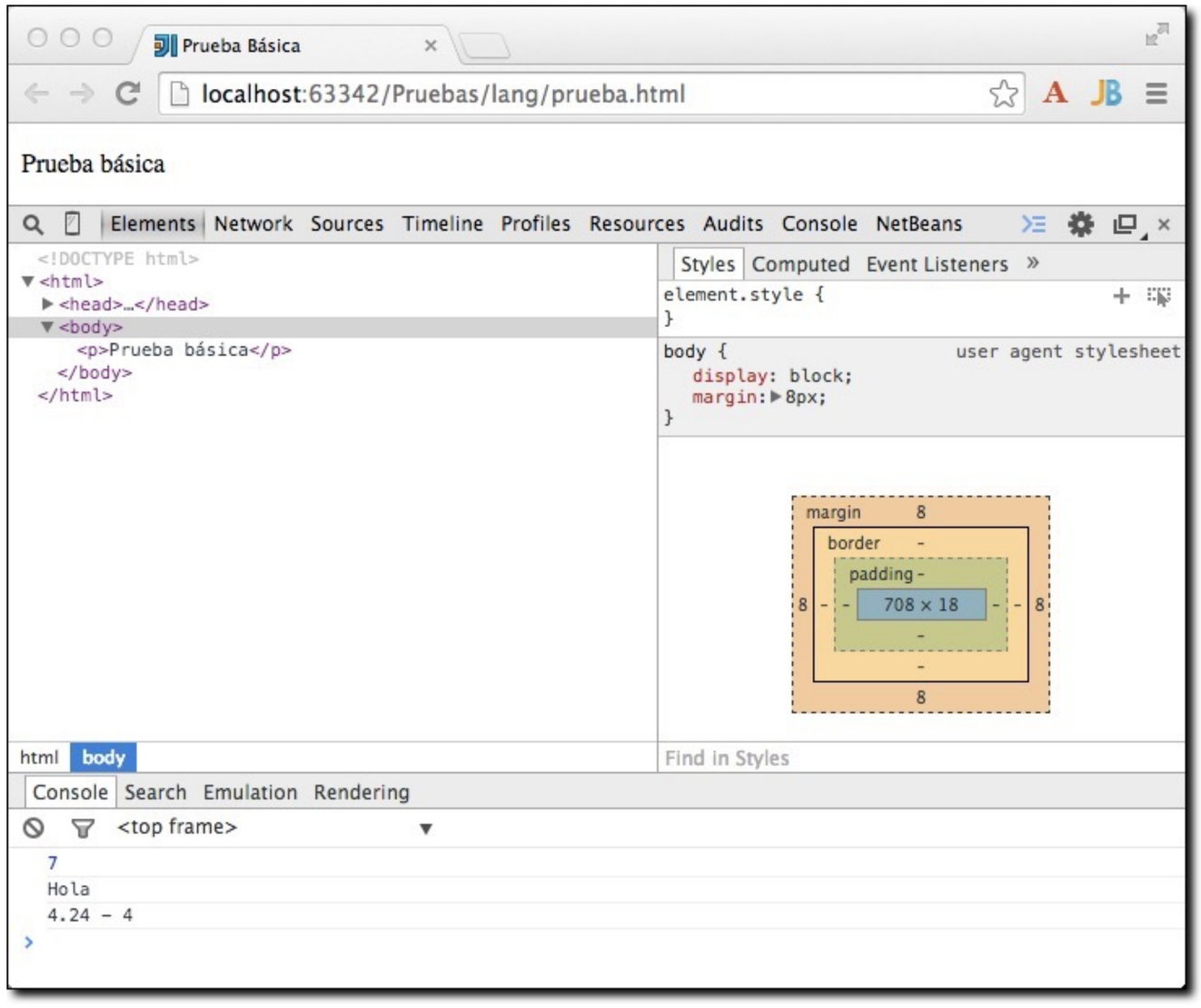
- Navegador Web.
  - Google Chrome (Chrome V8) → ECMAScript 5

JSBin

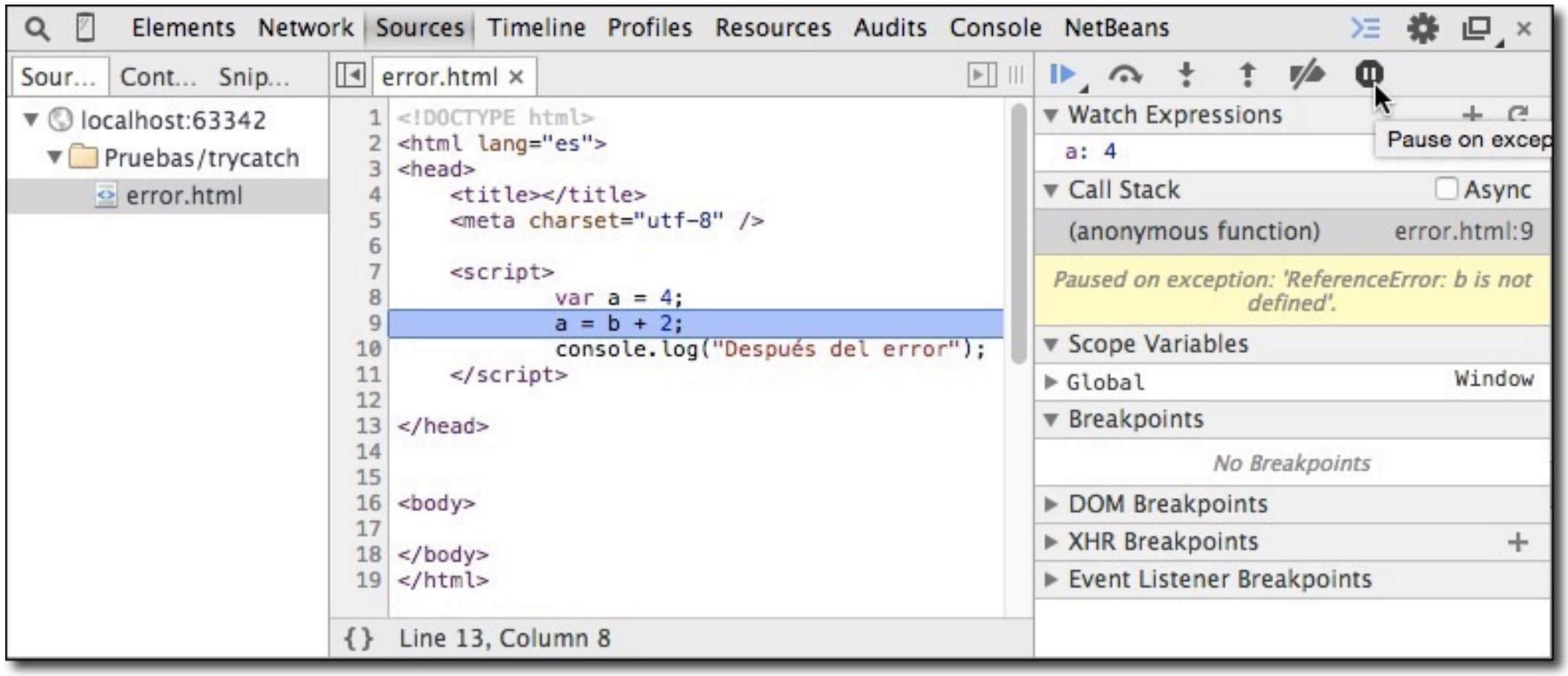
- Código en abierto
- Pensado para compartir código
  - http://jsbin.com/wikoha/edit?html,js,console
- HTML, CSS, JS, Console, Output
- Strict Mode
- *JSHint*
- Ideas:
  - Entrar con *GitHub*
  - Añadir descripción
  - Clonar bins



Chrome Developer Tools (*Dev-Tools*)



Debug con *Dev-Tools*



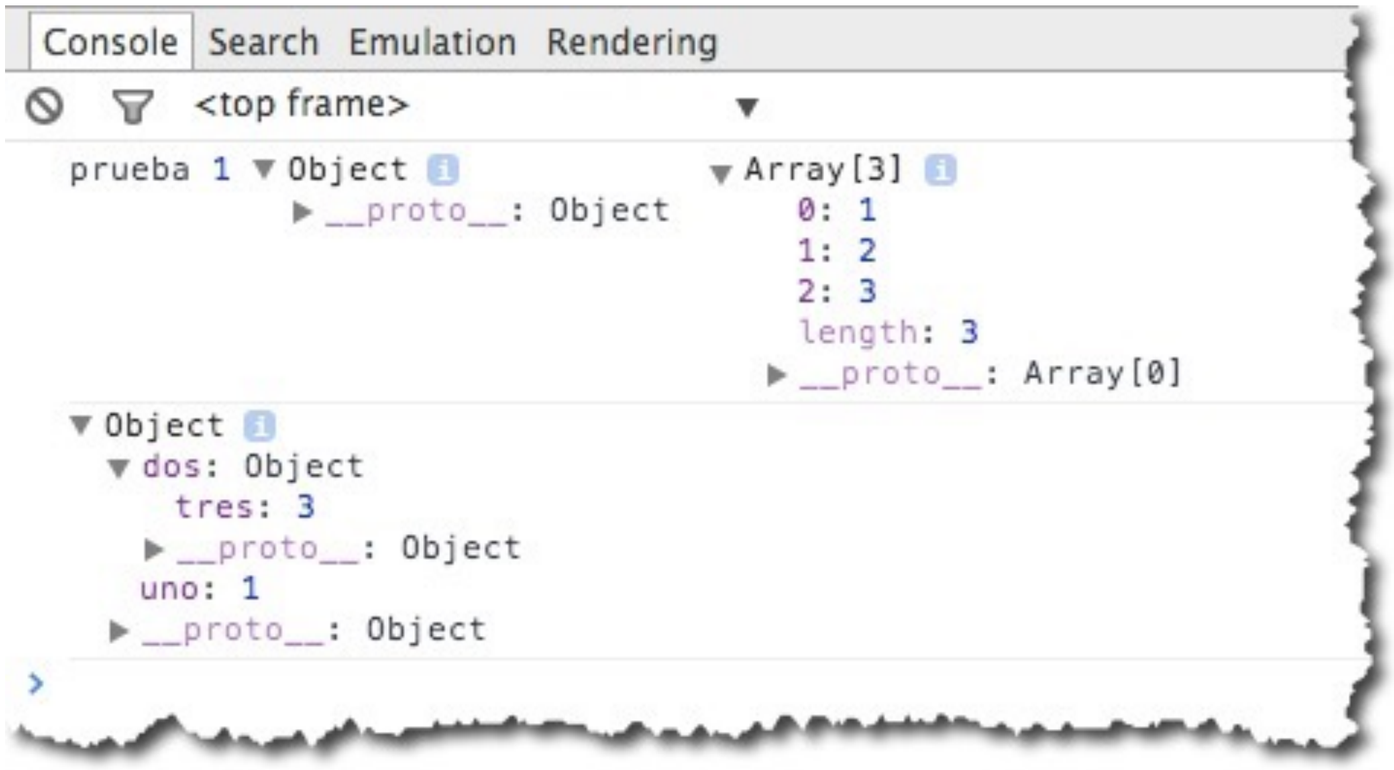
Console API

- Para escribir en la consola, usaremos el objeto `console` mediante diferentes métodos que reciben como parámetros el elemento que queremos mostrar:
  - `log()`

    `info()`, `warn()`, `error()`: diferentes niveles de log
  - `dir()`: enumera las propiedades
  - `assert()`: aserciones
  - `time()` y `timeEnd()`: rangos de tiempo

Ejemplo Console API

```
console.log("prueba", 1, {}, [1,2,3]);
console.dir({uno: 1, dos: {tres: 3}});
```



1.4 Variables

- *keyword* `var` delante del nombre de la variable
- nombrado similar a *Java* → recomendado *camelCase*
- Sensible a las mayúsculas → `contador != Contador`

```
var contador; // undefined
contador = 5;
```

- El tipo del dato se asigna de manera automática al asignarle un valor. Si no tiene valor, es `undefined`.
- Sin modo estricto, podemos asignar valores a variables no declaradas, las cuales tendrán un alcance **global**.
  - Mucho cuidado al no usar el modo estricto y las variables globales.

Hoisting (elevación)

- Permite tener múltiples declaraciones con `var` a lo largo de un bloque.
- Todas actuarán como si estuviesen declaradas al inicio del mismo.

```
var a = 3;
console.log(b); // undefined
var b = 5;
```

- ***Single Var Declaration***
  - Patrón de diseño
  - Declarar las variables con una única instrucción
  - En la primera línea de cada bloque

```
var a = 3,
  b = 5,
  suma = a + b,
  z;
```

Cadenas

- UTF-16
- Entre comillas dobles o simples

```
"" // cadena vacía
'probando'
"3.14"
'nombre="miFormulario"'
"comemos en el McDonald's"
```

- Para unir cadenas, operador `+`
- Las cadenas ofrecen la propiedad `length` para averiguar su longitud

```
var msj = "Hola " + "Mundo";
var tam = msj.length;
```

Operaciones con cadenas

- **`chartAt`**`(indice)` → devuelve el carácter que ocupa la posición *indice (0-index)*
  - *ECMAScript* 5 permite acceder mediante notación array: cadena[indice]

- **`indexOf`**`(carácter)`, **`lastIndexOf`**`(carácter)`: obtiene el indice de la primera o última ocurrencia del carácter
  - `-1` si no la encuentra

- **`substring`**`(inicio [,fin])` → Devuelve la subcadena comprendida entre *inicio* y el final de la misma o el indice indicado por *fin*
  - **`slice`**`(inicio [,fin])`, **`substr`**`(inicio, longitud)`

- **`trim`**`()` → Elimina los espacios en blanco de inicio y fin de la cadena

Ejemplo operaciones `string`

```
var nombre = "Bruce Wayne";

console.log(nombre);
console.log(typeof(nombre));  //  "string"
console.log(nombre.toUpperCase());
console.log(nombre.toLowerCase());

console.log(nombre.length); // 11 -> es una propiedad

console.log(nombre.charAt(0));  // "B"
console.log(nombre.charAt(-1));  // ""

console.log(nombre.indexOf("u")); // 2
console.log(nombre.lastIndexOf("ce"));  // 3
console.log(nombre.indexOf("Super")); // -1

console.log(nombre.substring(6)); // "Wayne"
console.log(nombre.substring(6,9)); // “Way"

console.log(nombre.replace("e","i")); // “Bruci Wayne"
```

Números

- Punto flotante de 64 bits

```
var diez = 10; // entero
var pi = 3.14; // real
```

- Para eliminar decimales, método **`toFixed`**`(numDigitos)`

```
var pi = 3.14159265;
console.log(pi.toFixed(0)); // 3
console.log(pi.toFixed(2)); // 3.14
console.log(pi.toFixed(4)); // 3.1416
```

- Operaciones: `+`, `-`, `*`, `/`, `%`

- Contantes: `Infinity`, `-Infinity`

Texto ↔ Número

- **parseInt**(cadena [, base]) → texto a entero
- **parseFloat**(cadena [, base]) → texto a real

```
var cadena = "3.14";
var pi = parseFloat(cadena, 10);
var tres = parseInt(pi, 10);
```

- **NaN** → *Not a Number*

```
var numero1 = 0;
var numero2 = 0;
console.log(numero1/numero2); // NaN
console.log(parseInt("tres")); // NaN
```

- **isNaN**(valor) → *booleano* que indica si no es un número

```
var miNumero = "tres";
if (isNaN(miNumero)) {
  console.log("¡No es un número!");
}
```

`Math`

- Objeto con métodos estáticos para realizar operaciones matemáticas.
  - Potencia: `Math.pow(base,exp)`
  - Raíz cuadrada: `Math.sqrt(num)`
  - Redondear: `Math.round(num)`, `Math.ceil(num)`, `Math.floor(num)`
  - Mayor: `Math.max(num1, num2,…)`
  - Menor: `Math.min(num1, num2, …)`
  - Número real aleatorio entre 0 y 1: `Math.random()`
  - Número pi: `Math.PI`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Booleanos

- `true` o `false`
- Operadores: `&&`, `||`, `!`
- Comparadores: `<`, `<=`, `>`, `>=`, `==`, `!=`, `===`, `!==`

- Valores falsos: `0`, `-0`, `null`, `false`, `NaN`, `undefined`, o una cadena vacía (`""`)
- El resto de valores son verdaderos, es decir, objetos, arrays o valores distintos de 0.

```
var esBooleano = true;
```

===

- Operador **identidad**
- No realiza conversión de tipos
  - `==` → **sí** que realiza conversión de tipos

```
var verdadero = ("1" == true);
var falso = ("1" === true);
```

- Mejor usar el operador identidad para evitar la conversión de tipos
- Recuerda: **3 mejor que 2**

Coerción de tipos

- Cuando a un operador se le aplica un valor con un tipo de datos incorrecto, *JavaScript* convertirá el valor al tipo que necesita.

```
console.log(8 * null) // 0
console.log("5" - 1) // 4
console.log("5" + 1) // 51
console.log("five" * 2) // NaN
console.log(false == 0) // true
```

`Date`

- Objeto que permite trabajar con fechas
- Constructor vacío para la fecha actual, entero para *ms* desde fecha *Epoch* (1/1/70) o parámetros (año, mes, día)

```
var fecha = new Date();
console.log(fecha); // Wed May 21 2014 21:03:59 GMT+0200 (Hora de verano romance)

var nochevieja = new Date(2014, 11, 31);
console.log(nochevieja); // Wed Dec 31 2014 00:00:00 GMT+0100 (Hora estándar romance)

var cenaNochevieja = new Date(2014, 11, 31, 22, 30, 0);
console.log(cenaNochevieja); // Wed Dec 31 2014 22:30:00 GMT+0100 (Hora estándar romance)
```

- Métodos:
  - `getFullYear()`: devuelve el año de la fecha con cuatro dígitos
  - `getMonth()`: número del mes del año (de 0 a 11)
  - `getDate()`: número de día del mes

Autoevaluación → Fechas

- ¿Qué saldrá por consola?

```
var cenaNochevieja = new Date(2014, 11, 31, 22, 30, 0);
var anyo = cenaNochevieja.getFullYear();
var mes = cenaNochevieja.getMonth();
var diaMes  = cenaNochevieja.getDate();
var incognita = new Date(cenaNochevieja.setDate(diaMes + 1));

console.log(incognita);
```

Comparando fechas

- Operadores `<` o `>` con el objeto `Date`
- Comparación con igualdad/identidad con el método `getTime()` → devuelve el *timestamp*

```
var cenaPreNochevieja = new Date(2014, 11, 30, 22, 30, 0);
var cenaNochevieja = new Date(2014, 11, 31, 22, 30, 0);
var cenaNochevieja2 = new Date(2014, 11, 31, 22, 30, 0);

console.log( cenaPreNochevieja < cenaNochevieja ); // true
console.log( cenaNochevieja == cenaNochevieja2 ); // false
console.log( cenaNochevieja === cenaNochevieja2 ); // false
console.log( cenaNochevieja.getTime() == cenaNochevieja2.getTime() ); // true
console.log( cenaNochevieja.getTime() === cenaNochevieja2.getTime() ); // true
```

- Para comparaciones más complejas → **Datejs** (www.datejs.com)

`typeof`

- Devuelve una cadena con el tipo del operando
- `'number'`,`'string'`,`'boolean'`,`'undefined'`,`'function'` y `'object'`

```
typeof 94.8 // 'number'
typeof "Batman" // 'string'
```

1.5 Instrucciones

- Sintaxis similar a Java
  - Condicionales: `if`, `if/else` (también podemos usar el operador ternario `?:`) y `switch`
  - Iterativas: `while`, `do/while`, `for`, `for… in`, `break` y `continue`
  - Tratamiento de excepciones: `try/catch` y `throw`
  - Depuración: `debugger` y `label`

- Punto y coma
  - *JavaScript* permite omitirlo en algunas circunstancias
  - No se recomienda omitir su uso

1.6 Funciones

- Una función es un objeto que puede invocarse
  - Pueden almacenarse en variables, objetos y *arrays*
  - Pueden pasarse como argumentos a funciones
  - Una función puede devolver una función (ella misma u otra)
  - Pueden contener métodos
  - *keyword* **`function`**
- No hay error si no coincide el número de parámetros y el de argumentos
  - `if (numArgumentos < numParametros) { parametrosSinAsignar = undefined; }`
  - Sin comprobación de tipos.
- Para devolver una valor → `return` (es opcional)
  - Si una función no hace `return`, el valor devuelto será `undefined`.

Función declaración

- Función con nombre
- Los parámetros se pasan por copia

```
function suma(alfa, beta) {
  return alfa + beta;
}

function operando(gamma, delta, fn) {
  return fn(gamma, delta);
}

var epsilon = operando(3, 4, suma);
```

Función expresión

- Una función se considera un valor
- Función anónima que se asigna a una variable

```
var miFuncionExpresion = function (param1, param2) {
  // instrucciones
  return variable;
}
```

- Si no hacemos `return`, se asigna `undefined` a la expresión

Función expresión como parámetro

```
var suma = function (alfa, beta) {
  return alfa + beta;
};

var operando = function (gamma, delta, fn) {
  return fn(gamma, delta);
};

var epsilon = operando(3, 4, suma);
```

```
var operando = function (gamma, delta, fn) {
  return fn(gamma, delta);
};

var epsilon = operando(3, 4, function(alfa, beta) {
  return alfa + beta;
});
```

Función declaración *vs* expresión

- Las funciones declaración se cargan antes de cualquier código
  - El motor *JavaScript* permite ejecutar una llamada a esta función incluso si está antes de su declaración (debido al *hoisting*).
- Las funciones expresión se cargan conforme lo hace el *script*
  - No se puede realizar una llamada a la función hasta que sea declarada
  - Hay que colocarlas antes del resto de código que quiera invocar dicha función.

```
cantar();
estribillo(); // TypeError: undefined

function cantar() {
  console.log(“¿Qué puedo hacer?");
}

var estribillo = function() {
  console.log(“He pasado por tu casa 20 veces");
};
```

Callback

- Función que se le pasa a otra función para ofrecerle a esta segunda función un modo de volver a llamarnos más tarde.
- Al llamar a una función, le enviamos por parámetro otra función (un *callback*) esperando que la función llamada se encargue de ejecutar esa función *callback*.
- Se utilizan mucho para gestionar los eventos del DOM

```
function haceAlgo(miCallback){
  //hago algo y llamo al callback avisando que terminé
  miCallback();
}

haceAlgo(function(){
  console.log('he acabado de hacer algo');
});
```

Workflow mediante *callbacks*

- Distintos *callbacks* que se van llamando en determinados casos
  - puntos de control sobre una función para facilitar el seguimiento de un *workflow*

```
function haceAlgo(callbackPaso1, callbackPaso2, callbackTermino){
   // instrucciones proceso 1
   callbackPaso1('proceso 1');
   // instrucciones proceso 2
   callbackPaso2('proceso 2');
   // instrucciones proceso Final
   callbackTermino('fin');
}
function paso1(quePaso){
   console.log(quePaso);
}
function paso2(quePaso){
   console.log(quePaso);
}
function termino(queHizo){
   console.log(queHizo);
}
haceAlgo(paso1, paso2, termino);
```

`arguments`

- Cada función recibe dos parámetros adiciones: `this` y `arguments`.
- `arguments` da acceso a todos los argumentos recibidos mediante la invocación de la función, incluso los que sobraron y no se asignaron a parámetros.
  - Permite escribir funciones que tratan un número indeterminado de parámetros.
- Estructura similar a un array, aunque realmente no lo sea.
  - propiedad `length` para obtener el número de parámetros.
  - notación `arguments[x]` para acceder a cada elemento.
  - carece del resto de métodos que ofrecen los *arrays*.

http://jsbin.com/wikoha/edit?js,console

```
var suma = function() {
  var i, s=0;
  for (i=0; i < arguments.length; i+=1) {
    s += arguments[i];
  }
  return s;
};

console.log(suma(1, 2, 3, 4, 5));  // 15
```

1.7 Alcance

- Determina desde donde se puede acceder a una variable
  - donde nace y donde muere.
- Alcance **global** → cualquier variable o función global pueden ser invocada o accedida desde cualquier parte del código de la aplicación.
  - Por defecto, todas las variables y funciones que definimos tienen alcance global.
  - Si olvidamos declarar una variable, su alcance es global
- Alcance de **función** → al definir una variable dentro de una función, la variable vive mientras lo hace la función.
  - Aquella variable/función que definimos dentro de una función (padre) es local a la función pero global para las funciones anidadas (hijas) a la que hemos definido la función (padre) → alcance **anidado**.
  - Podemos definir funciones dentro de funciones con alcance anidado en el hijo

    serán accesibles por el nieto, pero no por el padre.

Autoevaluación → Alcance

```
var varGlobal = "Esta es una variable global.";

var funcionGlobal = function(alfa) {
  var varLocal = "Esta es una variable local";

  var funcionLocal = function() {
    var varLocal = "¡Hola Mundo!";
    console.log(varLocal);
    console.log(alfa);
  };

  funcionLocal();
  console.log(varLocal);
};

// console.log(varLocal)
funcionGlobal(2);
```

http://jsbin.com/didake/1/edit?js,console

IIFE - Función expresión de invocación inmediata

- Función que se ejecuta inmediatamente
  - También conocido como *closure* anónimo
- Evita colisiones con variables locales
- Las variables declaradas dentro de un IIFE son locales a la misma

```
(function(){
  // código
})();
```

```
(function(){
  var a = 1;
  console.log(a);  // 1
})();
console.log(a);  // Uncaught ReferenceError: a is not defined
```

```
(function() {
  var a = b = 5;
})();
console.log(b);
```

http://jsbin.com/giziqo/1/edit?js,console

Hoisting y alcance

- Dentro de una función, al referenciar a una variable local nombrada de manera similar a una variable global, al hacer *hoisting* y declararla más tarde, la referencia apunta a la variable local.

```
"use strict";
var a = "global";
console.log(b); // undefined
var b = 5;
console.log(b); // 5

function hoisting() {
  console.log(b); // undefined
  var b = 7;
  console.log(b); // 7
}
hoisting();
```

```
function prueba() {
  console.log(a);
  console.log(hola());

  var a = 1;
  function hola() {
    return 2;
  }
}
prueba();
```

http://jsbin.com/xocamo/1/edit?js,console

1.8 *Timers*

- Permiten la invocación de funciones tras un lapso de tiempo y con repeticiones infinitas
- `setTimeout(función, tiempoMS)` → ejecuta la *función* después de un periodo de *tiempoMS* milisegundos.
  - No bloquea la ejecución del código
- `setInterval(función, intervaloMS)` → ejecuta la *función* inmediatamente, y repite la llamada cada *intervaloMS* milisegundos, de manera ininterrumpida

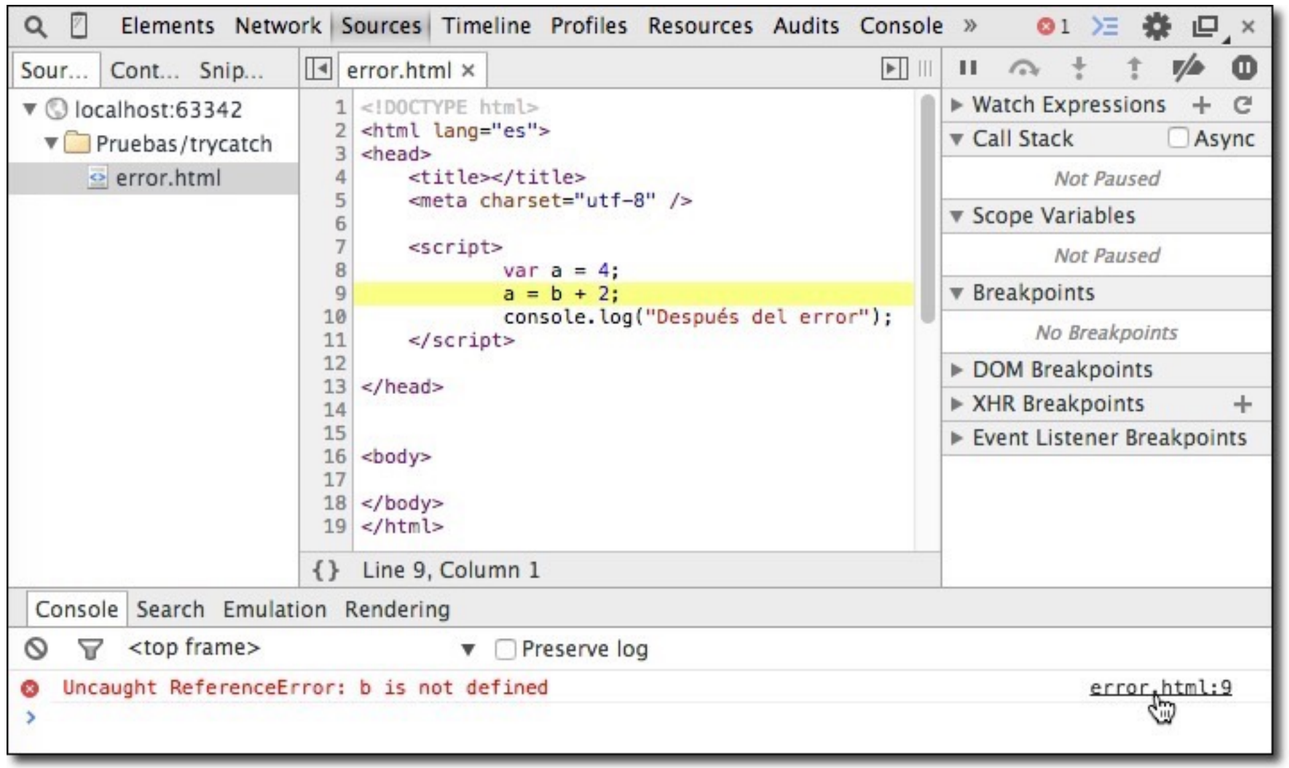
```
(function() {
  var miFuncion = function() {
    console.log("Batman vuelve");
  };
  setTimeout(miFuncion, 2000);
}());
```

```
(function() {
  miFuncion = function() {
    console.log("Batman vuelve");
  };
  setInterval(miFuncion, 2000);
}());
```

1.9 Gestión de errores

```
var a = 4;
a = b + 2;
console.log("Después del error");
```

- *DevTools* → Mensaje de error → a la derecha, pinchar sobre `archivo:linea` → panel *Sources*

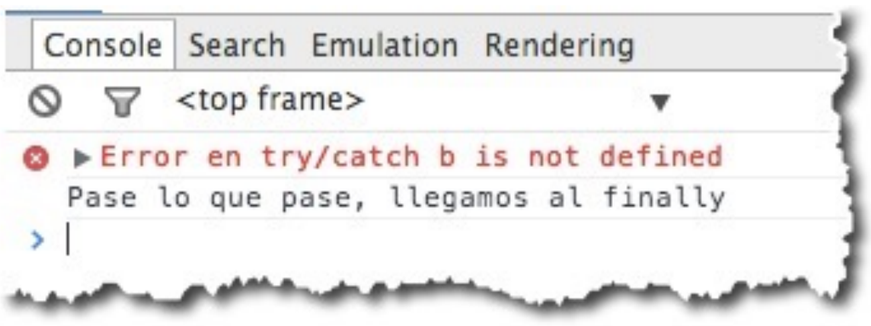


Capturar excepciones

- `try` / `catch` / `finally` → igual que *Java*
  - Al capturar una excepción podemos acceder a la propiedad `message`

```
try {
  var a = 4;
  a = b + 2;
  console.log("Después del error");
} catch(err) {
  console.error("Error en try/catch " +  err.message);
} finally {
  console.log("Pase lo que pase, llegamos al finally");
}
```

http://jsbin.com/kenufe/2/edit?js,console



Lanzar excepciones

- `throw` → permite lanzar una excepción, ya sea una cadena de texto o un objeto `Error`.

```
function ecuacion2grado(a,b,c) {
  var aux = b*b-4*a*c;
  if (aux < 0) {
    throw "Raíz Negativa";
    // throw new Error("Raíz Negativa");
  }
  // resto del código
}
```

Debug

- *Dev-Tools*
- Instrucción `debugger`
  - Funcionamiento similar a un *breakpoint*
  - Si está activo el depurador, la ejecución se detendrá.
  - Sino, no pasará nada.

```
function funcionQueDaProblemas() {
  debugger;
  // código que funciona de manera aleatoria
}
```

Errores Comunes

- No cerrar una cadena, olvidando las comillas de cierre.
- Olvidar el punto y coma tras asignar una función anónima a una variable/propiedad.
- Invocar a una función, método o variable que no existe.
- Errores de sintaxis, por ejemplo, `document.getElementByID("miId");`
- Referenciar un elemento del DOM que todavía no se ha cargado.
- En una condición, usar una asignación (=) en vez de una comparación (== o ===). Aunque la aplicación no se va a quejar, siempre será verdadera.

- Pasar a una función menos parámetros de los necesarios no provoca ningún error, pero podemos obtener resultados inesperados.

¿Preguntas?