UT03.4: Sintaxis y operadores

Contenido

1. Introducción

Vamos a profundizar en la sintaxis de JavaScript, así como en el uso de operadores mediante una serie de ejemplos partiendo de nuestra página de prueba **MiTienda**. Los contenidos que vamos a trabajar son:

1. Sentencia `switch`.
2. Bucles `for/in` y `for/of`.
3. Objeto `Math`.
4. Cortocircuitos lógicos
5. Objeto `Date`.
6. Manejo de exceptiones.
7. Tipos de datos.
8. Operador de propagación.

1.1. Ficheros iniciales

Disponemos del fichero **util.js** utilizando anteriormente con las dos opciones para mostrar mensajes en un elemento en pantalla. Además, hemos añadido la versión de limpiar el contenido del elemento, y la que permite añadir al contenido un nuevo texto, con lo que podremos mostrar varios mensajes en la página.

```
function showMessage (message){
  document.getElementById("message").innerHTML = message;
}
function showMessageInElement (id, message){
  document.getElementById(id).innerHTML = message;
}
function cleanMessage (){
  document.getElementById("message").innerHTML = "";
}
function cleanMessageInElement (id){
  document.getElementById(id).innerHTML = "";
}
function addMessage (message){
  document.getElementById("message").innerHTML += message + "<br>";
}
function addMessageInElement (id, message){
  document.getElementById(id).innerHTML += message + "<br>";
}
```

En cuanto al HTML, hemos creado una sección que contendrá los botones de prueba de cada uno de los ejemplos que implementemos. A modo de prueba, dispones de un botón ya creado.

```
<!-- Examples Buttons -->
```

```
<section id="examples" class="container cat-list">
  <div class="row">
   <h3>01 Switch</h3>
    <button class="btn" onclick="alert();">Ejemplo 1</button>
  </div>
</section>
```

Si nos fijamos en el botón, dispone de un atributo `onclick` utilizado para asignar un manejador de eventos al elemento. Como vemos invocamos la ejecución de una función `alert`. Lo ideal es separar la página HTML de la funcionalidad JavaScript, y que asignemos los manejadores desde el código JavaScript, pero en ocasiones este método es más sencillo.

La forma de asignar eventos en el código HTML es con el nombre del evento precedido de una preposición en forma de atributo.

Por cada función deberás crear un botón que permita su invocación para probarla.

2. Switch

Comenzamos revisando el uso de la sentencia `switch` que permite para simplificar la combinación de estructuras `if/else` de forma más eficiente.

Creamos un fichero **01switch.js** en la carpeta **js** y lo importamos en la página HTML.

2.1. Clausula `default` en cualquier punto

No es necesario indicar la cláusula `default` al final de la sentencia `switch`.

```
function defaultAnywhere() {
  let computerId = 200;
  switch (computerId) {
    default:
      showMessage("Producto desconocido");
      break;
    case 1:
      showMessage("Computer 1");
      break;
    case 2:
      showMessage("Computer 2");
      break;
    case 3:
      showMessage("Computer 3");
      break;
  }
}
```

2.2. Clausula `case` múltiple

Podemos crear más de una clausula `case` para que se empareje con la expresión y se ejecute la misma sentencia.

```
function multipleCase() {
  let color = "Rojo";
```

```
  switch (color) {
    case "Rojo":
    case "Verde":
      showMessage("El color es rojo o verde");
      break;
    case "Azul":
    case "Azul claro":
    case "Azul oscuro":
      showMessage("El color es azul");
      break;
    default:
      showMessage("Sin color");
      break;
  }
}
```

2.3. Perdida de break

Cuando no indicamos un `break` el código seguirá ejecutándose hasta que encuentre uno o termine el `switch`.

```
function forgetABreak() {
  cleanMessage ();
  let computerId = 2;
  switch (computerId) {
    case 1:
      addMessage("Computer 1");
      break;
    case 2:
      addMessage("Computer 2");
    case 3:
      addMessage("Computer 3");
      break;
    default:
      addMessage("Producto desconocido");
      break;
  }
}
```

2.4. Comparación estricta

La expresión de comparación utilizada es estricta, es decir, se compara valor y tipo.

```
function strictComparison() {
  let computerId = "2";
  switch (computerId) {
    case 1:
      showMessage("Computer 1");
      break;
    case 2:
```

```
      showMessage("Computer 2");
    case 3:
      showMessage("Computer 3");
      break;
    default:
      showMessage("Producto desconocido");
      break;
  }
}
```

3. Bucles

Tenemos dos tipos de bucles más que podemos utilizar en la versión ES6, `for/in` y `for/of`. Vamos a estudiarlos.

Creamos un fichero **02for.js** en la carpeta **js** y lo importamos en la página HTML.

3.1. for/in

Permite iterar sobre las propiedades y los métodos de un objeto, devolviendo su nombre. Utilizando la notación `objeto[name_propiedad]` podemos obtener su valor.

```
function forinSample() {
  cleanMessageInElement ("message2");
  let computer = {
    brand: 'HP',
    model: 'EliteBook',
    memory: 16,
    showInfo: function(){
      return this.brand + " " + this.model;
    }
  }
  for (const key in computer) {
    addMessageInElement("message2","'" + key + "'=" + computer[key]);
  }
}
```

3.2. for/of

Se utiliza para recorrer objetos iterables como un array. El bucle `for` tiene la ventaja de que se puede romper, es decir, podemos detener la iteración cuando lo necesitemos. El método `forEach` tiene la ventaja de ser conciso, sabemos que se aplica para cada elemento. Un bucle `for/of` toma las dos ventajas.

Disponemos un array de objetos literales.

```
let computers = [
  {
    computerID: 134,
    brand: 'HP',
    model: 'EliteBook',
    memory: 16,
```

```
  },
  {
    computerID: 14,
    brand: 'HP',
    model: 'EliteBook',
    memory: 32,
  },
  {
    computerID: 456,
    brand: 'HP',
    model: 'Pavilion',
    memory: 16,
  },
];
```

El siguiente código muestra como recorrer el array con el bucle. La constate *cp* toma el valor de cada elemento del array en cada iteración. El método `JSON.stringify` permite aplanar un objeto en forma de *string*.

```
function forofSample() {
  cleanMessageInElement ("message2");
  for (const cp of computers) {
    addMessageInElement("message2",JSON.stringify(cp));
  }
}
```

Aunque poco conocido, un *string* es un array de caracteres, por lo que lo podemos recorrer con el bucle.

```
function loopStringSample() {
  let model = "EliteBook";
  let letters = "";

  for (const char of model) {
    letters += char;
  }
  showMessage(letters);
}
```

3.3. Sentencias `break` y `continue`

Usar break en un bucle permite abortar la iteración con antelación. El siguiente código rompe el bucle cuando detecta un ordenador con más memoria de 16GB.

```
function breakSample() {
  cleanMessage();
  for (const cp of computers) {
    if (cp.memory > 16) {
      break;
    }
    addMessage(JSON.stringify(cp));
  }
```

```
}
```

Sin embargo, con una sentencia `continue`, solamente saltamos a la siguiente iteración en el bucle, por lo que solamente se mostrarán dos objetos.

```
function continueSample() {
  cleanMessage();
  for (const cp of computers) {
    if (cp.memory > 16) {
      continue;
    }
    addMessage(JSON.stringify(cp));
  }
}
```

4. Math

La clase `Math` en *JavaScript* tiene la misma base que en *Java*. Básicamente se trata de una clase que no necesita instanciar objetos ya que sus métodos y propiedades son estáticas. La clase engloba funciones matemáticas, y se utiliza en combinación con `Number`.

Creamos un fichero **03math.js** en la carpeta **js** y lo importamos en la página HTML.

4.1. Propiedades de Number

`Number` tiene las siguientes propiedades como podemos consultar en la guía MDN:

Propiedad	Descripción
Number.MAX_VALUE	El número representable más grande (±1.7976931348623157e+308).
Number.MIN_VALUE	El número representable más pequeño (±5e-324).
Number.NaN	Valor especial *not a number* ("no es un número"). Se utiliza en operaciones que generan un resultado imposible de transforma a un `Number`.
Number.NEGATIVE_INFINITY	Valor infinito negativo. Representa el desbordamiento negativo con `Number`.
Number.POSITIVE_INFINITY	Valor infinito positivo. Representa el desbordamiento positivo con `Number`.
Number.EPSILON	Diferencia entre 1 y el valor más pequeño mayor que 1 que se puede representar como un `Number` (2.220446049250313e-16).
Number.MIN_SAFE_INTEGER	Número entero seguro mínimo en JavaScript ($-2^{53} + 1$ o −9007199254740991)
Number.MAX_SAFE_INTEGER	Máximo número entero seguro en JavaScript ($+2^{53} - 1$ o +9007199254740991)

Tabla 1 Propiedades de `Number`

La siguiente función muestra el contenido de estas propiedades.

```
function numberProperties(){
```

```
  cleanMessageInElement ("message2");
  addMessageInElement("message2",Number.MAX_VALUE);
  addMessageInElement("message2",Number.MIN_VALUE);
  addMessageInElement("message2",Number.NaN);
  addMessageInElement("message2",Number.NEGATIVE_INFINITY);
  addMessageInElement("message2",Number.POSITIVE_INFINITY);
  addMessageInElement("message2",Number.EPSILON);
  addMessageInElement("message2",Number.MIN_SAFE_INTEGER);
  addMessageInElement("message2",Number.MAX_SAFE_INTEGER);
}
```

4.2. Métodos de Number

Ya hemos estudiado los métodos estáticos `Number.parseFloat()` y `Number.parseInt()`. También tenemos disponibles funciones de comprobación de valores, los vemos en la siguiente tabla.

Propiedad	Descripción
Number.isFinite()	Determina si el valor pasado es un número finito.
Number.isInteger()	Determina si el valor pasado es un número entero.
Number.isNaN()	Determina si el valor pasado es `NaN`. Versión más robusta del isNaN() global original.
Number.isSafeInteger()	Determina si el valor proporcionado es un número que es un entero seguro.

Tabla 2 Métodos de `Number`

En la siguiente función tenemos ejemplos de utilización de algunos de los métodos anteriores.

```
function numberMethods(){
  cleanMessageInElement ("message2");
  addMessageInElement("message2",
    Number.isFinite(Number.NEGATIVE_INFINITY)); //false
  addMessageInElement("message2",
    Number.isInteger(123.23)); //false
  addMessageInElement("message2",
    Number.isNaN(Number.parseInt("a"))); //true
}
```

Desde el ámbito global, tenemos disponibles funciones como `isNaN()`, `parseInt()` o `parseFloat()`, pero en la versión ES6 están siendo sustituidas por las incluidas en `Number`.

4.3. Método de Number.isNaN()

No podemos comparar un valor `NaN` mediante el comparador de igualdad. Para realizar este chequeo debemos utilizar el método `Number.isNaN()`.

Este método sustituye a la función `isNaN()` en la versión ES6. Mientras que la función no chequea tipos, por esa razón *"123"* devuelve false y *"abc"* devuelve *true*, el nuevo método devolverá en ambos casos *false* porque solamente se aplica a tipos *number*.

```
function checkIsNaN(){
```

```
  const x = Number.NaN;
  cleanMessageInElement ("message2");
  addMessageInElement("message2" ,x === Number.NaN); //false
  addMessageInElement("message2", Number.isNaN(x)); //true

  addMessageInElement("message2", isNaN("123")); //false
  addMessageInElement("message2", isNaN("abc")); //true

  addMessageInElement("message2", Number.isNaN("123")); //false
  addMessageInElement("message2", Number.isNaN("abc")); //false
}
```

4.4. Uso de Number.EPSILON

Podemos utilizar la constante `Number.EPSILON` como margen de error en comparaciones de números en coma flotante, y así poner un parche a los problemas de precesión de interprete en este tipo de operaciones.

Si la diferencia entre el dato calculado y el valor esperado es menor que `Number.EPSILON` devolvemos *true*. La función `epsEqu(x, y)` permite hacer esta comparación.

```
function epsEqu(x, y) {
  return Math.abs(x - y) < Number.EPSILON;
}

function useEpsilon(x, y){
  cleanMessage ();
  addMessage("0.1 + 0.2 === 0.3 -> " +
    (0.1 + 0.2 === 0.3)); //false
  addMessage("0.1 + 0.2 === 0.3 -> " +
    epsEqu(0.1 + 0.2, 0.3)); //true
}
```

4.5. Enteros seguros

El rango de enteros que podemos representa en JavaScript es de −253 < i < 253. Todo número entero que sobrepase estas cantidades genera error de precisión, por lo que, si trabajamos con números enteros muy grandes, es conveniente chequear si son seguros o no.

En el siguiente ejemplo podemos ver el límite de enteros seguros, y como al rebasar ese límite, las operaciones nos son precisas.

```
function safeIntegers(){
  cleanMessage ();
  addMessage(Number.isSafeInteger(9007199254740990)); //true
  addMessage(Number.isSafeInteger(9007199254740992)); //false
  addMessage(9007199254740992 + 3); //9007199254740996
}
```

4.6. Números aleatorios

El método `Math.random()` genera un número aleatorio decimal en el rango [0,1]. Mediante fórmulas matemáticas y redondeos podemos generar números aleatorios enteros en un rango determinado.

```
function randomIntegers(){
  cleanMessage ();
  addMessage(Math.trunc(Math.random() * 10));       // [0,9]
  addMessage(Math.trunc(Math.random() * 100));      // [0,99]
  addMessage(Math.trunc(Math.random() * 10) + 1);   // [1,10]
  addMessage(Math.trunc(Math.random() * 100) + 1); // [1,100]
}
```

5. Cortocircuitos lógicos

Los cortocircuitos lógicos simplifican muchas operaciones que podemos realizar en el lenguaje. Se basan en el uso de los operadores `&&` y `||`.

- Operador `&&`: Solo se evalúa la segunda expresión, y por tanto es ejecutada, cuando la primera expresión es verdadera.
- Operador `||`: Solo se evalúa la segunda expresión, y por tanto es ejecutada, cuando la primera expresión es falsa.

Creamos un fichero **04cortocircuito.js** en la carpeta **js** y lo importamos en la página HTML.

5.1. Cortocircuito para evitar errores

Si intentamos acceder a la propiedad de un objeto cuando es *undefined* el intérprete generaría un error. Podemos evitar este error utilizando `&&` siendo la primera expresión la verificación de que no tenemos un *undefined* y la segunda expresión el acceso a la propiedad.

```
function avoidErrors(){
  cleanMessage ();
  let obj = undefined;
  if (obj !== undefined && obj.property){
    addMessage(obj.property);
  }
  addMessage("Función ejecutada sin error");
}
```

5.2. Asignar valores predeterminados

En ES5 no tenemos la opción de asignar valores predeterminados a los parámetros de una función. Mediante `||` podemos asignar valores predeterminados a los parámetros no invocados. El problema con este tipo de estructuras es que, si el argumento que estamos pasando al parámetro es falso, como por ejemplo un 0, el parámetro será asignado con el valor predeterminado, produciendo un error.

```
function defaultValueParameter(param1, param2){
  param1 = param1 || 99;
  param2 = param2 || "Valor por defecto";
  cleanMessage ();
  addMessage(param1); //99
```

```
  addMessage(param2); //"Valor por defecto"
}
```

La invocación es:

```
defaultValueParameter(0);
```

5.3. Invocación opcional de función

Con `&&` podemos evaluar una expresión para invocar la ejecución de una función solo sí la expresión es verdadera.

```
function invokeFunction(){
  cleanMessage ();
  true && addMessage("Invocada"); //Invocada
  false && addMessage("No invocada"); //
}
```

6. Date

Objeto para manipular fechas mediante métodos, pero sin propiedades. Se utiliza igual que en Java, internamente almacena un valor numérico que representa la cantidad de milisegundos transcurridos desde 1 de enero de 1970, pudiendo representar un rango entre -100$_1$000.000 de días a 100$_1$000.000 a partir de esa fecha.

Creamos un fichero **05date.js** en la carpeta **js** y lo importamos en la página HTML.

Los siguientes ejemplos son obtenidos de la web de MDN (https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Numbers_and_dates)

6.1. Días hasta el final de año

El siguiente ejemplo muestra cómo podemos crear un objeto que representa la fecha actual, como podemos crear un objeto con una fecha determinada y modificar la fecha de un objeto mediante métodos *setter*.

```
function remainingDaysEndYear(){
  let today = new Date(); //Día actual
  let endYear = new Date(1995, 11, 31, 23, 59, 59, 999); //Establece día
y mes
  endYear.setFullYear(today.getFullYear()); //Establece año actual
  let msPerDay = 24 * 60 * 60 * 1000; //Nº de milisegundos por día
  let daysLeft = (endYear - today) / msPerDay;
  daysLeft = Math.round(daysLeft); //Días que quedan en el año
  showMessage(daysLeft);
}
```

Como podemos ver en la función, podemos operar con un objeto de tipo `Date`, en este caso el operador resta obtiene la diferencia en milisegundos entre las dos fechas.

6.2. Reloj digital

A través de los métodos *getter* podemos obtener los datos que nos permitan trabajar con una fecha. En el ejemplo, vemos como manipular la hora, los minutos y los segundos para obtener un reloj en formato AM/PM.

```
function digitalClock(){
```

```
  let time = new Date();
  let hour = time.getHours();
  let minute = time.getMinutes();
  let second = time.getSeconds();
  let temp = '' + ((hour > 12) ? hour - 12 : hour);
  if (hour === 0)
    temp = '12';
  temp += ((minute < 10) ? ':0' : ':') + minute;
  temp += ((second < 10) ? ':0' : ':') + second;
  temp += (hour >= 12) ? ' PM' : ' AM';
  showMessage(temp);
}
```

6.3. Formato de fecha

El objeto `Date` puede ser transformado a un tipo primitivo *string* mediante un conjunto de métodos. El problema que tenemos es que está transformación utilizar el formato anglosajón con el mes anterior al día.

Para resolver esta circunstancia, tenemos los métodos *locale* que transforman la fecha en función de la configuración regional del ordenador.

```
function dateFormat(){
  cleanMessage ();
  let d = new Date();
  addMessage(d.toDateString());//Formato anglosajón
  addMessage(d.toLocaleDateString());//Formato local
}
```

7. Manejo de excepciones

Los errores en la ejecución del código ocurren provocando estados de excepción que debemos gestionar para evitar que se propaguen. Tenemos cuatro tipos de sentencias para la gestión de excepciones:

- `try`: Testea un bloque de código para chequear si se produce un error.
- `catch`: Permite manejar el error.
- `throw`: Lo utilizamos para crear nuestros estados de excepción personalizados.
- `finally`: este bloque de código es ejecutado independientemente de que se haya producido un estado de excepción, o el código del bloque `try` se haya ejecutado correctamente.

Creamos un fichero **06excepcion.js** en la carpeta **js** y lo importamos en la página HTML.

7.1. Bloque `try-catch`

Cualquier error producido en el bloque `try` provoca que se ejecute el bloque `catch`. Este bloque recibe un parámetro denominado `error` que es un objeto de tipo `Error`. Un objeto `Error` tiene dos propiedades:

- `name`: Contiene el nombre del error.
- `message`: Describe el tipo de error que ha ocurrido.

En la siguiente función se produce un error porque la variable `x` no está declarada, por lo que se ejecuta el bloque `catch`.

```
function simpleTryCatch() {
  'use strict';
  let result;
  cleanMessage ();
  try {
    result = x / 10;
  } catch (error) {
    addMessage(error.name);//ReferenceError
    addMessage(error.message);//x is not defined
  }
}
```

7.2. Bloque finally

Este bloque siempre se he ejecuta independientemente de que ocurra un error o no.

```
function finallyCatchSample() {
  'use strict';
  let result;
  cleanMessageInElement ("message2");
  try {
    addMessageInElement("message2" ,"Va a ocurrir un error.");
    result = x / 10;
    addMessageInElement("message2" ,"Esta línea nunca ocurre");
  } catch (error) {
    addMessageInElement("message2" ,"Gestionamos la excepción: " + error.
name);
  }
  finally {
    addMessageInElement("message2" ,"El bloque finally siempre se ejecuta
.");
  }
}
```

Prueba a modificar la función anterior evitar que se provoque una excepción.

7.3. Errores personalizados

A través de la sentencia `throw` podemos lanzar nuestras excepciones, personalizando el objeto con el error para que sea gestionado en otro punto del código. En este ejemplo, `throwError()` invoca en un bloque `try` a `attemptDivision()`. Esta segunda función provoca un error, y en su bloque catch lanza un error personalizado a través de un objeto literal a `throwError()`, que será quien lo gestione.

```
function throwError() {
  try {
    attemptDivision();
  } catch (error) {
    showMessage(error.name + ": " + error.message);
```

```
  }
}

function attemptDivision() {
  let result;
  try {
    result = x / 10;
  } catch (error) {
    throw {
      "message": "Error en attemptDivision(): " + error.name,
      "name": "CustomError"
    };
  }
}
```

7.4. Manejo de tipos de error

Para gestionar tipos de error diferentes en un bloque `catch`, tenemos que diferenciar qué tipo de error es en base a la propiedad `name` utilizando una estructura `switch`.

```
function handleError(error) {
  switch (error.name) {
    case "ReferenceError":
      showMessage("Reference error: " + error.message);
      break;
    case "RangeError":
      showMessage("Range error: " + error.message);
      break;
    case "TypeError":
      showMessage("Type error: " + error.message);
      break;
    case "URIError":
      showMessage("URI error: " + error.message);
      break;
    case "SyntaxError":
      showMessage("Syntax error: " + error.message);
      break;
    case "EvalError":
      showMessage("Evaluation error: " + error.message);
      break;
    default:
      showMessage("Error: " + error.name + " - Mensaje: " + error.message
);
      break;
  }
}
```

La función anterior solamente chequea los tipos de errores predefinidos en el intérprete, dejando la cláusula default para errores personalizados.

- `ReferenceError`: Hacemos referencia a una variable que no existe.
- `RangeError`: Un valor está fuera de los límites del conjunto de valores permitidos.
- `TypeError`: Se produce cuando una operación no puede ser realizada, típicamente cuando un valor no es del tipo esperado. Ejemplos:
  - Un argumento pasado a una función es incompatible o tiene un tipo no esperado.
  - Intento de modificar un valor que no puede ser cambiado.
  - Intento de usar un valor en una forma inapropiada.
- `URIError`: Representa un error en el manejo de una URI.
- `SyntaxError`: Tenemos de un código sintácticamente erróneo.

Para más información consulta la referencia de errores de MDN https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error.

Las siguientes funciones utilizan el manejo de errores para determina el tipo de error producido.

```
function rangeErrorSample() {
  let result = 0;

  try {
    result.toPrecision(200);
  } catch (error) {
    handleError(error);
  }
}

function typeErrorSample() {
  let result = 0;

  try {
    result.toUpperCase();
  } catch (error) {
    handleError(error);
  }
}

function uriErrorSample() {
  let uri = "http://www.example.com/path%%%/file name";

  try {
    // URI error
    decodeURI(uri);
  } catch (error) {
    handleError(error);
  }
}
```

8. Operador de propagación (...)

Este operador permite a un objeto de tipo iterable como un array o un string, se expanda en lugares donde se esperaba 0 o más argumentos.

8.1. Transformar un string a un array

El siguiente ejemplo muestra como recorrer un string para generar un array con cada uno de sus caracteres.

```
function stringToArray() {
  let reference = "DKR2427";
  let values = [...reference];
  showMessage(values);//D,K,R,2,4,2,7
}
```

8.2. Copiar arrays

El operador permite crear una nueva instancia de array en base a otro. En la función siguiente, partimos de un array y generamos un segundo en base al primero utilizando el operador de propagación. Chequeamos el segundo array para ver como las modificaciones no afectan al primer array.

```
function copyArray() {
  let arr = [1, 2, 3];
  let arr2 = [...arr];

  cleanMessage ();
  arr2.push(4);
  arr2[0] = 99;

  addMessage(arr);
  addMessage(arr2);
}
```

Tenemos que tener la precaución si utilizamos el operador con un array de objetos, ya que por debajo, un objeto es una referencia, por lo que estaríamos copiando son las referencias. El ejemplo vemos que al hacerse la copia, todas las copias apuntan al mismo objeto, y por tanto todas las copias pueden modificarlo.

```
function copyObjectArrays() {
  cleanMessage ();
  let diff = [...computers1];
  diff[0].computerID = 999;
  addMessage(computers1[0].computerID);//999
  addMessage(diff[0].computerID);//999
}
```

8.3. Concatenar arrays

Además de utilizar el método `concat()` podemos concatenar arrays utilizando el operador de propagación.

```
function concatenateArray() {
```

```
  let join = [...computers1, ...computers2];
  showMessage(join.length);//6
}
```

8.4. Propagar constructor

Podemos crear un objeto pasando los parámetros a través de un array y propagando sus elementos para invocar al constructor.

```
function spreadInConstructors() {
  cleanMessage ();
  let dt = new Date(2020, 9, 15);  // 15 Oct 2020
  addMessage(dt.toLocaleDateString());
  let dateFields = [2020, 10, 15]; // 15 Nov 2020
  dt = new Date(...dateFields);
  addMessage(dt.toLocaleDateString());
}
```

8.5. Propagar argumentos

Basado en el caso anterior, podemos propagar los elementos de un array para invocar una función.

```
function spreadForFunctionArgs() {
  cleanMessageInElement ("message2");
  multipleParams(1, 2, 3);
  let args = [1, 2, 3];
  multipleParams(...args);
}

function multipleParams(arg1, arg2, arg3) {
  addMessageInElement("message2" , arg1);
  addMessageInElement("message2" , arg2);
  addMessageInElement("message2" , arg3);
}
```