TSR: Práctica 1. Sesión 1
Ejecución de programas JavaScript

Trabajo a realizar

Durante esta primera sesión debes ejecutar, entender y ser capaz de razonar sobre diferentes características del lenguaje JavaScript, sobre todo si comparamos este lenguaje con algún lenguaje que conozcamos más, como puede ser Java.

Vamos a trabajar sobre los siguientes aspectos:

1. Hola mundo en JavaScript.
2. Ámbito de las variables
3. Tipos de las variables
4. Argumentos de funciones.
5. Clases en JavaScript
6. Clausuras
7. Programación asíncrona mediante callbacks.

Nota: Dispones de material de apoyo con varios documentos y ejemplos que te pueden servir para familiarizarte con el lenguaje JavaScript. Todo ello está contenido en la **Práctica 0: práctica no guiada, no presencial**, que estimamos que todo alumno debe realizar como paso previo al desarrollo de las prácticas. La Práctica 0, te debe permitir el trabajo con los próximos ejemplos de forma mucho más sencilla. Allí se presentan múltiples ejemplos y explicaciones de varios conceptos que seguimos trabajando y ampliando en esta sesión de prácticas. **Si no has hecho la práctica 0,** deberías repasar los documentos de esa práctica ya que contienen material que te facilitará el trabajo en esta práctica y siguientes.

1.- Hola mundo en JavaScript

Con este ejemplo repasamos los elementos mínimos de un programa en JavaScript.

1.1 Lee y ejecuta el programa “ej1_HolaMundo.js”

- Observa el uso de la directiva “use strict” al comienzo del programa.
- Observa que no hay función “main”. Se ejecuta directamente línea a línea todo el código que proporcionemos.
- Observa la diferencia entre declarar una función y ejecutarla.
- Razona todos los pasos que se ejecutan y lo que observas que se imprime por la pantalla.

2.- Ámbito de las variables

Hemos de conocer las diferencias entre utilizar “let” y utilizar “var” para declarar variables. Debemos saber cuándo podemos usar cada una de ellas y qué implicaciones tiene. También debemos remarcar lo que ocurre si no empleamos “let” ni “var” sin emplear el modo estricto de JavaScript.

2.1 Lee y Ejecuta el programa “ej2_AmbitoVariables.js” y **contesta** a las siguientes cuestiones

Cuestión 1. Observa que el programa modifica el valor de una variable global dentro de la función “f1”. ¿Podemos modificar el valor de las otras variables globales? ¿Qué diferencias hay entre declarar variables globales con “let” y con “var”?

Cuestión 2. Observa que en la línea 24 se imprime el valor de la variable “var_i”. ¿podemos imprimir el valor de la variable “let_i” ? Prueba a hacerlo y razona lo que observas. Razona qué diferencias hay entre usar “let” y usar “var”.

Cuestión 3. Este programa no utiliza la directiva “use strict” al comienzo del programa. Añádela y prueba a ejecutar de nuevo el programa. ¿Qué observas? Explica para qué sirve la directiva “use strict”.

Cuestión 4. Modifica el programa para que funcione correctamente usando la directiva “use strict”. Corrige el error que ha cometido el programador por referenciar equivocadamente la variable “local1_let”. Razona qué ventajas tiene para el programador el uso de la directiva “use strict”.

3. Tipos de las variables

JavaScript es un lenguaje no tipado. Este hecho puede ser bastante confuso en muchas situaciones y es necesario programar siendo conscientes de los tipos de las variables y de sus conversiones.

3.1 Lee y ejecuta el programa “ej3_TiposDeVariables.js”

3.2 Razona por qué la resta funciona de forma diferente a la suma.

3.3 Consulta en Internet información sobre el lenguaje “TypeScript”. Razona los motivos que están llevando a su creciente implantación a día de hoy.

4. Argumentos de funciones

JavaScript es un lenguaje bastante flexible en cuanto al número de argumentos de las funciones y sus tipos. Esta característica es potente y al mismo tiempo puede llevar a errores de programación.

4.1 Lee y ejecuta el programa “ej4_Argumentos.js” y **contesta** a las siguientes cuestiones:

Cuestión 1. Explica qué hace el pseudo-vector “arguments”. ¿Se puede acceder a los diferentes argumentos mediante este pseudo-vector?

Cuestión 2. Al imprimir result3, vemos “NaN”. ¿Qué significa NaN y por qué vemos este resultado?

Cuestión 3. ¿Qué significan los 3 puntos suspensivos en la llamada a imprimir “resultv1” ?

Cuestión 4. Observa qué se imprime como resultado “resultv2”. ¿Por qué obtenemos este resultado?

resultv2: 1,2,3,4undefinedundefined

5. Clases en JavaScript

El soporte a programación orientada a objetos de JavaScript es un tanto primitivo, aunque ha mejorado mucho desde las versiones de 2015. A día de hoy se puede usar la sintaxis más reciente y cómoda de clases o utilizar la forma "antigua" de crear clases e instancias. Ambas formas coexisten.

5.1. Lee y ejecuta el programa ej5-1_Clases.js

Observa cómo se emplea "this", de forma similar a cómo se utiliza en otros lenguajes.
Observa cómo se declaran los atributos de las clases. Borra el atributo "nombre" y vuelve a ejecutar el programa. Observa que la declaración de atributos es completamente opcional.
Observa cómo se declaran los constructores y cómo se emplea "super".

Este programa hace uso de la sintaxis más moderna de JavaScript. Aún es muy posible que el soporte a objetos de JavaScript sea mejorado en futuras versiones.

Aunque el uso de clases es recomendable en JavaScript, no se ejercitará en esta asignatura. Por ello, no se presentan otros ejemplos relacionados con clases en esta práctica.

6.- Clausuras

Seguramente uno de los conceptos más importantes de JavaScript (y de otros lenguajes) son las clausuras. Esta sección de prácticas no pretende dar un curso completo de clausuras, pero sí dar unas pinceladas básicas a su empleo.

Para simplificar la explicación podemos resumir diciendo que toda función que se ejecuta en JavaScript tiene asociada una clausura. La clausura de una función son las variables (y funciones) que se referencian desde dicha función. Estos símbolos que se referencian desde la función, permanecen en memoria, mientras la función esté siendo referenciada. El sentido práctico lo encontramos cuando la clausura contiene **símbolos que no sean globales.**

El caso más sencillo lo tenemos en una función que no referencia a nada. En este caso podemos decir que su clausura es vacía. Más habitualmente se dice simplemente que esta función no tiene clausura o que no hace uso de clausuras. Lo mismo ocurre si la clausura únicamente referencia símbolos globales, pues estos símbolos siempre son accesibles y no es necesario ningún "esfuerzo por parte del lenguaje" para permitir el acceso a estos símbolos.

6.1.- Lee y ejecuta el programa "ej6-1_Clausuras.js"

Observa que la función f1 no tiene clausura.
Observa que la clausura de f2 es la variable x. Como se trata de un símbolo global, realmente no es una clausura en sentido estricto, pues la función puede acceder a esta variable global sin ningún problema.
Observa la función f3. En este caso tenemos el **patrón habitual** de las clausuras. Una función que retorna otra función. La función f3() se llama **función generadora** y la función que retornamos se la suele llamar **función clausura**, o simplemente clausura, para abreviar. La función clausura retornada por f3 tiene como clausura el argumento "arg" y la variable "i". Ambos símbolos son locales a la

función “f3” y por tanto son visibles en la función clausura, pues están en su ámbito. El soporte a clausuras de JavaScript mantendrá estas variables en memoria mientras la función clausura esté referenciada. Observa cómo la función clausura está referenciada en la variable “f”. Observa cómo perdura el valor de la variable “i” entre invocaciones sucesivas.

6.2.- Lee y ejecuta el programa “ej6-2_Clausuras.js”

Este ejemplo ilustra un caso de uso habitual, sobre todo en software JavaScript para navegadores. Observa como hay 3 segmentos de código que hacen prácticamente lo mismo. El primer segmento usa una variable global. El segundo segmento de código usa una función generadora y una clausura y de esta forma evitamos el uso de variables globales. Sin embargo creamos un símbolo global para mantener la función. El tercer segmento de código es más complejo, pero más potente, pues mediante clausuras y funciones anónimas podemos evitar el uso de símbolos globales.

Responde a la siguiente cuestión.
Cuestión 1. ¿Por qué puede resultar interesante emplear un patrón como el descrito en este ejemplo cuando hacemos software que será empleado como biblioteca, desde un navegador o desde nodejs?

6.3.- Lee y ejecuta el programa “ej6-3_Clausuras.js”

Este ejemplo ilustra una clausura que contiene variables, argumentos y funciones. Después de ejecutar y estudiar el código, contesta la siguiente cuestión:

Cuestión 1. ¿Cuál es la clausura que retorna la función generadora? Detalla las variables, argumentos y funciones que forman parte de la clausura.

6.4.- Lee y ejecuta el programa “ej6-4_Clausuras.js”

Este ejemplo muestra una función generadora que retorna una función entre 2 posibles. Después de ejecutar y estudiar el código, contesta las siguientes cuestiones:

Cuestión 1. Cuál es la clausura de la función g0

Cuestión 2. Cuál es la clausura de la función g1

Cuestión 3. ¿Cuántas copias en memoria hay de la variable “traza” ? Recuerda que las clausuras mantienen en memoria las variables a las que referencian.

7.- Programación asíncrona mediante callbacks

La programación asíncrona proporciona una aproximación a la concurrencia diferente a la que proporciona la programación clásica multi-hilo. En una aproximación que puede verse como complementaria, más que como alternativa.

En las clases de teoría se va a profundizar en el **bucle de eventos**, como mecanismo central a la programación asíncrona que encontramos en JavaScript. Por tanto, esta sección de prácticas no sustituye al contenido de teoría, sino más bien, nos puede servir como una primera aproximación a ciertos aspectos prácticos.

Para simplificar, podemos decir que un programa en "nodejs", puede **registrar manejadores de eventos**. Cuando estos eventos sucedan, se ejecutarán los manejadores. Es algo similar a los manejadores de interrupción de los sistemas operativos, o los manejadores de eventos que encontramos en ciertas bibliotecas gráficas. (De hecho, varias bibliotecas gráficas que se estudian en esta Universidad se basan en un bucle de eventos: AWT, Swing, JavaFX, etc).

El evento más sencillo de generar y de manejar seguramente es el tiempo.

Con la llamada "setTimeout", hacemos simultáneamente 2 cosas: instalamos un manejador para el evento "timeout" y pedimos que se produzca el evento "timeout" dentro de un determinado número de milisegundos. Más adelante, cuando el tiempo que hayamos indicado venza, se ejecutará este manejador de forma asíncrona.

7.1.- Lee y ejecuta el programa ej7-1_Timeouts.js

Observa lo que hace el programa y contesta las siguientes cuestiones:

Cuestión 1. Por qué vemos el mensaje "cinco" antes del "cuatro"

Cuestión 2. Por qué vemos "dos" antes que "tres" o "tres" antes que "dos". Para razonar esta cuestión ejecuta varias veces el programa modificando el número de iteraciones del bucle para que haga 100, 1000, 10000, 100000 iteraciones.

7.2.- Lee y ejecuta el programa ej7-2_Timeouts.js

El programa registra 10 manejadores de evento y alcanza el final del programa.

Observa que el programa no termina al imprimir el mensaje final por la consola.

Observa el valor de variable que se imprime.

Contesta las siguientes cuestiones:

Cuestión 1. ¿Por qué siempre imprime 10?

Cuestión 2. ¿Por qué imprime un mensaje cada segundo?

Cuestión 3. ¿Por qué termina el programa al imprimir 10 mensajes por la consola? ¿Por qué no terminó al ejecutar la última línea de código del programa?

7.3.- Lee y ejecuta el programa ej7-3_Timeouts.js

Este programa combina la gestión de eventos con clausuras. Con este pequeño ejemplo podemos apreciar que mejora el ejemplo anterior gracias a las clausuras.

El código es breve, pero es necesario estudiarlo con cuidado. Dedícale varios minutos. Si comprendes este programa habrás dado un paso importante para comprender la forma de programar mediante JavaScript y nodejs.

Contesta a las siguientes cuestiones:

Cuestión 1. Identifica la función generadora y la función de clausura. ¿Cuándo y cuántas veces se llama a la función generadora? ¿Cuántas funciones de clausura se crean y quién las invoca?

Cuestión 2. ¿Cuál es el contenido de las clausuras?

Cuestión 3. ¿Qué ventaja ofrece en este ejemplo el uso de las clausuras?

7.4.- Lee y ejecuta el programa ej7-4_Timeouts.js

Este ejemplo crea clausuras sin utilizar funciones generadoras. Podemos más bien hablar de un bloque generador de las clausuras. Sirva este ejemplo para recordar que las funciones generadoras constituyen un patrón habitual para la creación de clausuras, pero como este ejemplo ilustra, no son estrictamente necesarias.

Observa que el bloque generador de clausuras declara una variable mediante “let”.

Contesta a la siguiente cuestión.

Cuestión 1. Modifica el programa para que declare la variable dentro del bloque “do” mediante “var”. Razona por qué ahora el funcionamiento es diferente.

7.5.- Lee y ejecuta el programa ej7-5_Timeouts.js

Este ejemplo contiene un programa un poco más completo que el resto. Podemos observar clausuras, un bucle de generación de clausuras y proporciona una función que será notificada cuando todas las clausuras terminen. La forma de notificar este fin, es mediante un callback.

Estudia y ejecuta el código para entender qué hace y contesta a la siguiente cuestión.

Cuestión 1. ¿Cuándo se ejecutará la función de callback proporcionada a la función forkJoinAsync y desde dónde se ejecutará?