UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Apunte 6 - Javascript parte I

Tipos de datos

Tipos de datos de JavaScript

- **undefined**: es el valor que contiene una variable que no ha sido inicializada.
- **string**: representa una cadena de caracteres, las cadenas son inmutables en JavaScript.
- **boolean**: representa un valor lógico y puede tomar dos valores `true` y `false`
- **null**: representa un valor nulo o vacío, tiene solamente un valor: `null`
- **symbol**: es un valor primitivo único e inmutable y se puede utilizar como clave de una propiedad de objeto.
- **number**: representa números de punto flotante de doble precisión de 64 bits (tomando valores entre $-(2^{53} - 1)$ y $2^{53} - 1$). Además contiene los valores simbólicos: `+Infinity`, `-Infinity` y `NaN`
- **bigint**: permite almacenar enteros que exceden los límites impuestos para el tipo `number`. Para especificar un literal `bigint` se agrega n al número que representa.
- **object**: tipo compuesto formado por asociación de *claves* y *valores*

Determinar el tipo de datos de una variable

Podemos conocer el tipo primitivo de datos de una variable utilizando el operador `typeof`

```
let edad = 22;
console.log(typeof edad);
// number
```

Hay casos en donde este operador no nos da toda la información que necesitamos. Indicaremos algunos de ellos oportunamente.

Variables

Tipado dinámico

JavaScript es un lenguaje de *tipado dinámico*. Como programadores, esto implica que no tenemos que declarar el tipo de datos de la variable cuando la declaramos. El intérprete asigna a las variables un tipo durante el tiempo de ejecución basado en su valor en ese momento. En contraposición al *tipado estático* en que los tipos de las variables se definen en tiempo de compilación.

Declaración de variables `let`

Para declarar una variable se antepone la palabra reservada `let` al identificador de la variable.

```
let nombre;
```

Identificadores

Los identificadores de las variables son sensibles a mayúsculas. Esto quiere decir que JavaScript considera `nombre` y `Nombre` como dos variables completamente distintas

Buena práctica: usar *camelCase* para los identificadores. Esto es: si tiene una sola palabra, comienza con minúscula; si tiene más de una palabra, la primera comienza con minúscula y las subsiguientes con mayúscula. Además conviene limitarse al uso de los números, caracteres alfabéticos (0-9, a-z, A-Z) y el caracter de subrayado. Ejemplos:

```
let color;
let estadoCivil;
let fechaDeAlta;
```

Asignar un valor a una variable

Usando el operador de asignación `=` se asigna un valor a una variable.

```
nombre = "Juan";
```

que almacena la cadena `'Juan'` en la varible `nombre`

también se puede inicializar la varible cuando se declara.

```
let cantidadDisponible = 4;
```

Variables no inicializadas

Las variables declaradas y no inicializadas aún contienen el valor `undefined` Si se utiliza una variable cuyo contenido es `undefined` en una operación matemática se obtendrá un resultado `NaN` que significa "Not a Number" (no numérico). Si en cambio se realiza en una concatenación de una cadena con una variable conteniendo `undefined` el resultado será el de concatenar esta cadena con la cadena `'undefined'`

Variables de solo lectura `const`

Podemos declarar variables de solo lectura o *constantes* con `const`. Éstas no pueden cambiar su valor en el transcurso del programa.

```
const PI = 3.141592;
const EDAD = 23;
```

Buena práctica: complementando lo dicho anteriormente sobre los identificadores, aquellos que se utilizan para constantes se suelen especificar en mayúscula sostenida. En el ejemplo de arriba lo escribiríamos.

Declaración de variables usando `var`

La declaración de variables mediante `let` y `const` se introdujo en ES6 por lo cual es altamente probable que en todos los navegadores y otros entornos de ejecucuón en que trabajes en la actualidad estén disponibles. Sin embargo, es también posible que encuentres código que utiliza `var` para declarar variables. Esta forma está aún disponible por razones históricas y las diferencias con `let` y `const`, que no discutiremos aquí, incluyen los conceptos de *ámbito* (scope) y *elevación* (hoisting).

Buena práctica: siempre que sea posible utilizaremos `let` y `const` en vez de `var`

Comentarios

Los comentarios son segmentos de código que el intérprete de JavaScript ingnorará de forma intencional. Los comentarios en JavaScript admiten dos variantes:

Comentario de una línea `//`

Para indicar que una línea o parte de una línea es un comentario utilizamos `//`

```
// Esta línea es un comentario
var a; // Esta parte de la línea es un comentario
```

Comentario multilínea `/* */`

Podemos crear un comentario multilínea indicando el principio y fin del comentario con `/*` y `*/` respectivamente

```
/* Este es
    un comentario
    multilínea
*/
```

Cadenas

Declarar una cadena

Para especificar un literal de cadena se puede hacerlo entre comillas simples `'` o dobles `"`

```
let nombre = "Juan García";
let colorPreferido = 'Azul';
```

Incluir caracteres especiales en una cadena

Para incluir caracteres especiales, por ejemplo las mismas comillas, como parte de una cadena. Debemos anteponerle el caracter de escape `\`

```
const cita = 'Sonó una canción llamada:\n"Dirty old town"';
console.log(cita);
```

Caracter	Descripción
\'	comilla simple
\"	comilla doble
\\	barra invertida
\n	nueva línea
\r	retorno de carro
\t	tab
\f	nueva página

Concatenar cadenas

Para concatenar dos o más cadenas se usa el operador `+`

```
let saludo = "Hola";
let nombre = "Silvina";
let mensaje = saludo + " " + nombre + "!";
```

Extraer caracteres de una cadena

Podemos acceder a un caracter específico dentro de una cadena especificando su posición entre corchetes. Hay que tener en cuenta que las posiciones empiezan a contar desde 0,

```
console.log("La tercera letra del saludo es'" + saludo[2] + "'");
```

Longitud de una cadena

La longitud de una cadena está disponible a través de la propiedad `length`

```
console.log("la longitud del mensaje es de", mensaje.length, "caracteres");
```

Plantillas literales

Las plantillas literales o plantillas de cadena son literales de cadena que nos permiten hacer dos cosas muy útiles:

- Especificar literales de cadena de más de una línea
- Interpolar variables y expresiones en una cadena.

Las plantillas literales se especifican entre comillas invertidas (backtick) `

Por ejemplo para definir una cadena de más de una línea:

```
`Esta es una
cadena
      de
        varias
              líneas`;
```

Tambien podemos interpolar variables o expresiones incluyéndolas entre llaves precedidas del signo pesos: `${expresión}`

```
let valorA = 10;
let valorB = 5;
console.log(
    `El resultado de la suma de ${valorA} y ${valorB} es ${valorA + valorB}`
);
```

Estructuras condicionales

Sentencia `if ... else`

La instrucción de selección más típica es la sentencia `if` cuya forma general es:

```
if (<condición>) {
    <sentencias que se ejecutan si la condición es verdadera>
}
```

Ejemplo:

```
if (monto > 5000) {
    console.log("La operación requiere autorización especial");
}
```

Si se quiere ejecutar sentencias cuando la condición es negativa se agrega la clausula `else`: La forma general de la sentencia `if` es:

```
if (<condición>) {
    <sentencias que se ejecutan si la condición es verdadera>
}
else {
    <sentencias que se ejecutan si la condición es falsa>
}
```

Ejemplo:

```
if (monto > 5000) {
    console.log("La operación requiere autorización especial");
} else {
    console.log("La operación ha sido autorizada");
}
```

Sentencia `switch`

En ocasiones nos encontramos frente a una situación en la cual la selección se realiza entre varias opciones. Ésta se puede implementar utilizando `if .. else if` pero la sentencia `switch` nos brinda una alternativa sintáctica más simple y clara. La forma general de la sentencia switch es:

```
switch (<expresion>) {
  case <caso1>:
    <sentencias que se ejecutan para el caso1>
    break;

  case <caso2>:
    <sentencias que se ejecutan para el caso2>
    break;

  ...

  // Otros casos

  default:
    <sentencias que se ejecutan si ninguno de los casos anteriores se ejecutó>
}
```

Ejemplo:

```
switch (color) {
    case "rojo":
```

```
            console.log("#FF0000");
            break;
        case "azul":
            console.log("#0000FF");
            break;
        case "verde":
            console.log("#00FF00");
            break;

        default:
            console.log("#000000");
    }
```

Operador ternario

El operador ternario sirve como un atajo para la instrucción `if .. else`. Debe su nombre a que es el único operador que implica tres operandos. La forma general es:

```
<condición> ? <expresión 1> : <expresión 2>
```

Si <condición> evalúa como verdadera el operador retorna la <expresión1>, si no retorna la <expresión2>

Ejemplo:

```
console.log(
    "La operación",
    monto > 5000 ? "requiere autorización especial" : "está autorizada"
);
```

Estructuras iterativas

El bucle `for`

El ciclo `for` repite un conjunto de instrucciones hasta que la condición especificada en su declaración evalúe como `false`. Tiene la siguiente forma:

```
for (<inicialización>; <condición>; <incremento>){
    instrucciones
    ...
}
```

Cuando se ejecuta el ciclo `for` sucede lo siguiente:

1. Inicialmente se ejecuta la expresión de `<inicialización>` habitualmente se incluyen en ésta la declaración e inicialización de uno o más contadores.
2. Luego se evalúa la `<condición>`, si ésta es verdadera se ejecutan las instrucciones en el cuerpo del ciclo `for`. Si esta expresión no se especifica, se considera siempre verdadera y el ciclo se ejecutará indefinidamente.
3. A continuación se ejecuta la expresión de `<incremento>`, habitualmente se incrementa el (o los) contadores en esta expresión.
4. Se regresa al paso 2.

Veamos un ejemplo simple que muestra en la consola los números del 1 al 10:

```
for (let i = 0; i < 10; i++) {
    console.log(i + 1);
}
```

En este ejemplo usamos un ciclo `for` para mostrar, uno en cada línea, los caracteres que contiene una cadena:

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

El bucle `while`

El bucle `while` tienen la siguiente forma:

```
while (<condición>){
    instrucciones
    ...
}
```

Las instrucciones del bloque se repiten mientras la <*condición*> evalúe a verdadero. La <*condición*> se evalúa al principio del bucle.

Veamos el ejemplo de mostrar los números del 1 al 10 por la consola. En esta oportunidad usando un bucle `while`

```
let i = 0;
while (i < 10){
    console.log(i+1);
    i++;
}
```

El bucle `do...while`

En el bucle do while, la *<condición>* se evalúa al final del bucle. Es decir que las instrucciones se ejecutarán al menos una vez hasta que se evalúe la *<condición>*

```
do {
    instrucciones
    ...
} while (<condición>)
```

El mismo ejemplo anterior, puede ser reescrito usando un ciclo `do...while` como:

```
let i = 0;
do {
    i++;
    console.log(i);
} while (i < 10);
```

La instrucción `break`

La instrucción `break` nos permite interrumpir la ejecución de un bucle `for`, `while`, `do...while` o una instrucción `switch` y continuar con la siguiente instrucción en el programa.

En el siguiente ejemplo mostramos en la consola todas las letras una cadena **hasta** que aparece la letra *l*

```
let color = "violeta";
for (let i = 0; i < color.length; i++) {
    console.log(color[i]);
    if (color[i] === "l") {
        break;
    }
}
```

La instrucción `continue`

La instrucción `continue` nos permite reiniciar un bucle `for`, `while`, `do...while`. Cuando se ejecuta la instrucción `continue` finaliza la iteración actual del bucle y sigue con la proxima iteración.

En este ejemplo usamos `continue` para mostrar todos los caracteres que componen una cadena **exceptuando** la letra *l*

```
let color = "violeta";
for (let i = 0; i < color.length; i++) {
    if (color[i] === "l") {
```

```
            continue;
        }
        console.log(color[i]);
    }
```

Operadores

Operadores aritméticos

Operador	Descripción
+	suma
-	resta
*	producto
/	división
%	módulo: retorna el resto de una division entera

Operadores de incremento y decremento

Operador	Descripción
++	incremento: incrementa en 1 la variable a la que se lo aplique
--	decremento: decrementa en 1 la variable a la que se lo aplique

Ambos operadores se pueden aplicar a la izquierda o a la derecha del valor que se incrementará/decrementará. En el primer caso incrementará/decrementará el valor antes de retornarlo, en el segundo caso lo incrementará despues de retornarlo.

```
let cantidad = 34;
console.log(cantidad++); //muestra 34
console.log(cantidad); // muestra 35
console.log(++cantidad); // muestra 36
```

Operadores de asignación

Operador	Descripción
=	asigna a la variable de la izquierda el valor que está a la derecha del operador
+=	suma el valor de la derecha al valor de la variable de la izquierda y retorna el nuevo valor
-=	resta el valor de la derecha, del valor de la variable de la izquierda y retorna el nuevo valor
*=	multiplica el valor de la variable en la izquierda por el valor en la derecha y retorna el nuevo valor

Operador	Descripción
/=	divide el valor de la variable en la izquierda por el valor de la derecha y retorna el nuevo valor

Operadores de comparación

Operador	Descripción
===	igual estricto
!==	distinto estricto
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que

Buena práctica: aunque en JavaScript existen otros operadores de igualdad y desigualdad: `==` y `!=` éstos no son estrictos y realizan una conversión implícita de tipo de datos en caso de ser necesario antes de comparar. Esto puede conducir a situaciones imprevisibles o no deseadas en algunas ocasiones. Por lo tanto usaremos, en general, los operadores estrictos para comparar.

Operadores lógicos

Operador	Descripción
&&	y (and)
||	o (or)
!	negación (not)

Conversión y coerción de tipos

Ya dijimos que JavaScript es un lenguaje de *tipado dinámico*. Esta característica hace que el tipo de datos de una variable se define en tiempo de ejecución, por lo tanto es posible utilizar variables de un tipo en operaciones que esperan otro tipo o combinar distintos tipos de datos en operaciones que esperan operandos del mismo tipo.
Para que esto sea posible JavaScript implementa un mecanismo llamado *coerción* de tipos, que realiza una conversión de tipos de forma **implícita** en ese contexto para que estas operaciones sean posibles. Esta flexibilidad conlleva falta de control sobre cómo este proceso se lleva a cabo, es entonces en muchos casos necesario realizar una *conversión* de tipos de forma **explícita** para obtener el resultado esperado. Veamoslo con un ejemplo:

```
const a = "3";
const b = 9;
console.log(a + b);
```

Quizá esperamos que la salida de este programa sea el número `12`, sin embargo el resultado es un string `39` ¿Qué ha pasado? JavaScript convirtió de forma implícita la variable `b` a `string` para poder usar el operador `+` que ya hemos visto que en ese contexto realiza una concatenación entre las cadenas y de ahí el resultado que obtenemos. En este ejemplo sencillo es fácil ver que sucedió pero en muchas ocasiones esto puede resultar confuso o se puede percibir como impredecible (aunque no lo es en realidad) ¿Cómo hacemos para que esta operación retorne la suma entre los dos valores numéricos? Necesitamos realizar una *conversión* explícita para que ésto suceda.

```
const a = "3";
const b = 9;
console.log(Number(a) + b);
```

Ahora sí nos muestra `12` como esperábamos.

Veamos ahora las principales conversiones de tipos entre tipos primitivos:

Conversión a number

Para convertir un valor a número usamos

```
Number(valor);
```

Alternativamente podemos usar las funciones globales:

```
parseInt(valor);
```

```
parseFloat(valor);
```

La conversión se lleva a cabo según estas reglas

Valor	Se convierte en...
`undefined`	`NaN`
`null`	`0`
`true`	`1`
`false`	`0`
`string`	Se eliminan los espacios al principio y al final, si el texto es vacío el valor será `0` de lo contrario es el valor que contiene el string, si no se puede convertir el valor será `NaN`

Ejemplos: conversion-tipos-a-number.js

Conversión a string

Para convertir a string usamos:

```
String(valor);
```

también podemos usar el método `toString()`

```
valor.toString();
```

Ejemplos: conversion-tipos-a-string.js

Conversión a boolean

Para convertir un valor a boolean utilizamos la función:

```
Boolean(valor);
```

En este caso la regla de conversión estará dada por los valores que evalúan a falso (llamados *falsy* de dificil traducción), el resto de los valores son *truthy*: evalúan a `true`

Los valores *falsy* son los siguientes:

- `false`
- `0`
- `-0`
- `0n`
- Cadena vacía: `''`,`""`, ````
- `null`
- `undefined`
- `NaN`

Ejemplos: conversion-tipos-a-boolean.js

Funciones

En términos generales una función es un subprograma, un conjunto de instrucciones que pueden ser reutilizadas para realizar una tarea específica. A estas instrucciones les llamamos el *cuerpo* de la función. Le podemos pasar valores a la función y ésta puede retornar un valor. En JavaScript las funciones son objetos y como tales pueden ser entre otras cosas asignados a una variable. Se distinguen de otros objetos porque pueden ser invocadas.

Definir funciones

Definir una función es el mecanismo para crear la función y darle un nombre. La función no se ejecuta cuando se crea sino cuando es invocada. En principio tenemos dos alternativas para definir una función: *declaración de función* (function statement) y *expresión de función* (function expression)

Declaración de función (function statement)

De esta forma declaramos una función para poder usarla (llamarla) cuando sea necesario. La sintaxis consta de la palabra `function` seguida de:

- el nombre de la función
- una lista de parámetros entre paréntesis `()` y separados por coma `,`
- el cuerpo de la función entre llaves `{}`

En este caso estamos declarando una función que no toma parámetros y no retorna ningún valor

```
function holaMundo() {
    console.log("Hola mundo");
}
```

Una función puede tomar uno o más parámetros

```
function saludo(nombre) {
    console.log("Hola", nombre);
}
```

Una función puede también retornar un valor

```
function sumar(a, b) {
    return a + b;
}
```

Expresión de función (function expression)

En la expresión de función asignamos la función a una variable o constante y no necesitamos dar un nombre a la función, es decir, ésta puede ser anónima.

Veamos los ejemplos anteriores usando expresión de función

```
const holaMundo = function () {
    console.log("Hola mundo");
};
```

```
let saludo = function (nombre) {
    console.log("Hola", nombre);
};
```

```
let sumar = function (a, b) {
    return a + b;
};
```

Algunos de los beneficios de usar expresiones de función son que de esta forma las funciones pueden ser usadas en:

- clausuras o cierres
- como argumentos de otras funciones
- como expresiones de función invocada inmediatamente (IIFE)

Diferencia entre declaración de función y expresión de función

- La declaración de función es elevada mientras que la expresión de función no lo es. Esto significa que se puede invocar una función definida mediante declaración de función antes de que sea definida, pero no podemos hacer eso con una expresión de función.
- Si usamos expresión de función podemos usar la función inmediatamente después de definirla. Si usamos declaración de función debemos esperar hasta que el script entero termine de analizarse para que pueda ser ejecutada.
- Las expresiones de función pueden ser utilizada como argumento para otra función, las declaraciones de función no.
- Las expresiones de función pueden ser anónimas, las declaraciones de función no.

Ejemplo de elevación de funciones

Este programa muestra 10 como salida:

```
console.log(foo());
function foo() {
    return 10;
}
```

El equivalente con expresión de función da un error porque la función `bar` no está definida en el momento de ser llamada:

```
console.log(bar());
let bar = function () {
    return 10;
};
```

Invocar funciones

Cuando se declara una función no se ejecuta. Necesitamos invocarla explícitamente. Invocamos una función como función o como método mediante una expresión de función (cualquier expresión que evalúa a un objeto función) seguido de un paréntesis que abre `(`, una lista de cero o más argumentos y un paréntesis que cierra `)`

```
const suma = function (a, b) {
    return a + b;
};

let resultado = suma(4, 6);
```

Hay distintos patrones para llamar una función. Las funciones pueden ser invocadas:

- como función
- como método
- como constructor
- indirectamente a través de sus métodos `call()` y `apply()`

Acabamos de ver el primero de ellos, iremos estudiando el resto de forma progresiva

Expresión de función invocada inmediatamente (IIFE)

Una expresión de función invocada inmediatamente es una expresión de función que se ejecuta tan pronto como se define.

Sintácticamente está formada por dos partes. La primera incluye la expresión de función entre paréntesis `()`. La segunda parte es la invocación de la función usando el operador `()`.

Veamos un ejemplo:

```
(function () {
    var number = Math.random() * 10;
    console.log(number >= 5);
})();
```

Uno de los usos más importantes de las IIFE es para mantener el scope global limpio. Dado que las variables o funciones definidas dentro de una función no pueden ser accedidas desde fuera de la función.

Ejercicios

Ejercicio 1:

Estructuras condicionales, operadores, funciones

Escribí un programa que dado un año como entrada diga si es bisiesto o no.

Tener en cuenta que un año es bisiesto si:

- Divisible por 4.
- No divisible por 100.
- Divisible por 400. (2000 y 2400 son bisiestos pues aún siendo divisibles por 100 lo son también por 400. Pero los años 1900, 2100, 2200 y 2300 no lo son porque solo son divisibles por 100).

Ejemplo:

Entrada:

```
1924
```

Salida:

```
1924 es un año bisiesto
```

Ejercicio 2:

Estructuras iterativas, estructuras condicionales, cadenas, funciones

Escribí un programa que dado un texto como una frase, escriba en la consola el texto, poniendo una palabra en cada línea y encerrado en una recuadro hecho con un caracter especial como el asterisco o numeral.

Ejemplo:

Entrada:

```
Errar es humano
```

Salida:

```
**********
* Errar  *
* es     *
* humano *
**********
```

Ejercicio Paso a Paso

Referencias

- JavaScript Guide - Mozilla Developer Network - Primeros pasos con JavaScript
- JavaScript Guide - Mozilla Developer Network - Elementos básicos de JavaScript

- El tutorial de JavaScript Moderno