

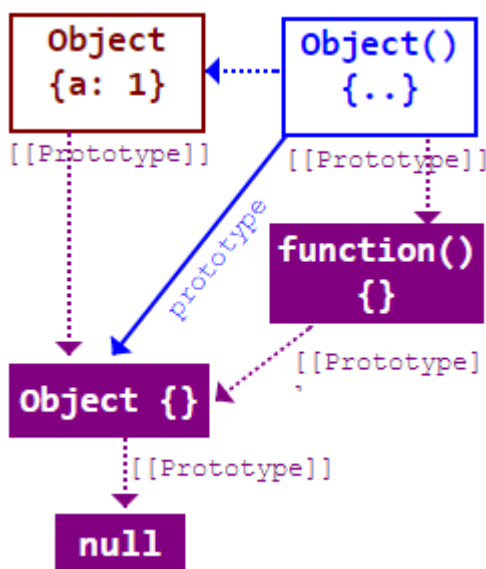
Clase 3 - Objetos

¿Qué es un objeto en JS? (*)

Un objeto es una colección de propiedades.

Toda propiedad es una asociación entre un nombre (o **clave**) y un **valor**. El valor de una propiedad puede ser una variable (o expresión constante), o una función. En el caso que la propiedad tiene por valor una función, hablamos de **método**.

JavaScript es un lenguaje de objetos basado en **prototipos**, donde los objetos heredan de otros objetos y en última instancia del prototipo de **Object** (`Object {}`).



Además de los objetos predefinidos de JS, es posible definir otros objetos. Éstos se pueden crear utilizando **notación literal**, o bien mediante **new Object()** o **funciones constructoras**.

Propiedades (*)

Un objeto de JavaScript tiene propiedades asociadas a él. Una propiedad de un objeto se puede pensar como una variable adjunta a dicho objeto. Las propiedades de un objeto definen sus características y se pueden acceder mediante:

- Notación de punto:

```
objeto.propiedad
```

- Notación de corchetes:

```
objeto['propiedad']
```

Una vez definido un objeto es posible crear (agregar) nuevas propiedades. Por ejemplo:

```
persona.pais = 'Argentina'
```

Las propiedades no asignadas de un objeto son `undefined`

```
persona.fechaNacimiento; // undefined
```

El nombre de la propiedad de un objeto puede ser cualquier cadena válida de JavaScript, o cualquier cosa que se pueda convertir en una cadena, incluyendo una cadena vacía. Sin embargo, cualquier nombre de propiedad que no sea un identificador válido de JavaScript (por ejemplo, el nombre de alguna propiedad que tenga un espacio o un guión, o comience con un número) solo se puede acceder utilizando la notación de corchetes. Esta notación es muy útil también cuando los nombres de propiedades son determinados dinámicamente (cuando el nombre de la propiedad no se determina hasta el tiempo de ejecución).

Creación de objetos

Creación de objetos con notación literal (*)

Cuando se utiliza la notación de llaves nos referimos a **notación literal**. Éstos se crean a partir de un juego de llaves `{}` Vamos con un ejemplo:

```
const juan = {  
  nombre: 'Juan',  
  correo: 'juan@gmail.com',  
  profesion: 'Comerciante',  
  fechaNacimiento: new Date(1994, 9, 1),  
  saludar: function () {  
    console.log('Hola soy ' + this.nombre);  
  }  
};
```

Nota:

La palabra reservada **this** en el contexto de los objetos, refiere al **objeto actual**. En los contextos de funciones permite referenciar al objeto **Window** (cuando estamos ejecutando código JS en un navegador).

Hay más ejemplos en el archivo [objetos-literales.js](#), ejecutando con

```
nodejs js/objetos-literales.js #ruta al archivo a ejecutar
```

Ademas tenemos otras dos formas de instanciar un objeto, utilizando el operador `new Object()` o a través de un `constructor de funciones`.

Creación de objetos con new Object

Utilizando el ejemplo anterior y el operador `new Object()`, también se podría reescribir la creación del objeto persona:

```
const persona = new Object();
persona.nombre = 'Juan';
persona.correo = 'juan@gmail.com';
persona.profesion = 'Comerciante';
persona.saludar = function(){
  console.log('Hola soy ' + this.nombre)
};
```

Creación de objetos con función constructora

Estas nos permiten crear nuevos tipos de objetos definidos por nosotros. Como alternativa, es posible crear un objeto con estos dos pasos:

1. Definir el tipo de objeto escribiendo una función constructora. Existe una fuerte convención para utilizar en mayúscula la letra inicial.
2. Crear una instancia del objeto con el operador `new`.

Para definir un tipo de objeto, crea una función para el objeto que especifique su nombre, propiedades y métodos. Por ejemplo, supongamos que se desea crear un tipo de objeto para el ejemplo anterior. Es decir un objeto de tipo `Persona` con las siguientes propiedades: `nombre`, `correo`, `profesion` y `saludar`. Para ello, escribir un ejemplo en el archivo `objetos-prototype.js`

```
function Persona(nombre, correo, profesion, fechaNacimiento) {
  this.nombre = nombre;
  this.correo = correo;
  this.profesion = profesion;
  this.fechaNacimiento = fechaNacimiento;
  this.saludar = function () {
    console.log('Hola soy ' + this.nombre);
  };
  this.edad = function () {
    const hoy = new Date();
    return hoy.getFullYear() - this.fechaNacimiento.getFullYear();
  };
}
```

y lo vamos a ejecutar con nodejs:

```
nodejs ./js/objetos-prototype.js #ruta al archivo a ejecutar
```

Observar que el operador **this** se utiliza para asignar valores a las propiedades del objeto en función de los valores pasados a la función.

Para crear un objeto llamado **persona** mediante la función constructora escribir:

```
const persona = new Persona('Juan', 'juan@gmail.com', 'Comerciante');
```

Esta declaración crea **persona** y le asigna los valores especificados a sus propiedades. Entonces el valor de **persona.nombre** es la cadena "Juan", para **persona.correo** es "juan@gmail.com", y así sucesivamente.

¿Y prototype?

Es importante recordar que javascript es un lenguaje basado en prototipos, en donde todos los objetos terminan heredando de **Object** y **prototype** nos permite agregar nuevos metodos a los constructores de los nuevos tipos de objetos que hayamos definido.

Por ejemplo:

```
nodejs ./js/objetos-prototype.js #ver ejemplo con protopye
```

Al utilizar **prototype** estamos afectando a todas las instancias del tipo de objeto en cuestión. En el ejemplo de **Persona** vemos que al manipular el resultado de la función **esMayorDeEdad** este se modifica para todas las instancias.

```
Persona.prototype.esMayorDeEdad = function () {  
  // if (this.edad() > 18)  
  //   return true;  
  // return false  
  return this.edad() > 18;  
};
```

Si en lugar de modificar **Persona**, modificamos una instancia, por ejemplo **carla**

```
carla.__proto__.esMayorDeEdad = function () {  
  return false;  
};
```

vemos que también afecta a **mario**. Considerar que para acceder a **prototype** por medio de la instancia, es necesario accederlo a través de la propiedad **__proto__**.

ECMAScript 6 y la palabra reservada *class*

Ahora con la sintaxis de **ECMAScript (ES6)**, se puede usar la palabra **class** para definir un tipo de objeto. Esta sintaxis implícitamente define objetos por medio de funciones constructoras, ya que el concepto de clases no existe en JS y la creación de objetos se termina haciendo a través de prototype. Lo que realmente nos facilita ES6 es una sintaxis amigable conocida como **syntactic sugar** o **lenguaje más dulce** para los programadores.

Ejemplo en archivo [objetos-es6.js](#)

```
class Persona {  
  
  constructor(nombre, correo, profesion, fechaNacimiento) {  
    this.nombre = nombre;  
    this.correo = correo;  
    this.profesion = profesion;  
    this.fechaNacimiento = fechaNacimiento;  
  }  
  
  saludar() {  
    console.log('Hola soy ' + this.nombre);  
  }  
  
  edad() {  
    const hoy = new Date();  
    return hoy.getFullYear() - this.fechaNacimiento.getFullYear();  
  }  
}
```

```
nodejs js/objetos-es6.js #ver ejemplo con es6
```

Una de las cosas nuevas que aporta ES6 es que al declarar tipos de objetos si o si, es necesario instanciarlos con la palabra reservada **new** sino lanza un error:

```
const maria = Persona('María', 'maria@gmail.com', 'Ingeniero en Sistemas de  
Información', new Date(2012, 9, 1));  
// lanza el error: TypeError: Class constructor Persona cannot be invoked without  
'new'
```

```
nodejs js/objetos-es6.js #Ver Inicialización de objetos sin new
```

Ademas algunas características de sintaxis y semantica de ES6 no son compartidas con es5. Por ello es importante tener en cuenta tanto las versiones de los lenguajes como los estándares.

Bibliografía o Referencia

- [JavaScript Guide - Mozilla Developer Network => Introduction](#)
- [Operador new](#)
- [Objetos - Prototype](#)
- [Clases ES6](#)

Clase 3 - Funciones de flecha

Repaso de funciones (*)

Teniendo en cuenta lo que se explicó de [funciones en la semana 1](#) podemos crear unos ejemplos para nuestro dominio de Personas, por ejemplo la función `obtenerPersonaMasGrande` que busca la persona más grande entre tres objetos `Persona` pasados por argumentos [arrow-funcntions.js](#)

```
obtenerPersonaMasGrande = function (p1, p2, p3) {  
  let mayor = p1;  
  if (p2.edad() > p1.edad())  
    mayor = p2;  
  else {  
    if (p3.edad() > p1.edad())  
      mayor = p3;  
  }  
  console.log('Soy una function/metodo común. Atte esMayorDeEdad');  
  return mayor;  
};
```

Podemos reescribirla como `arrow function`:

```
obtenerPersonaMasGrandeArrow = (p1, p2, p3) => {  
  let mayor = p1;  
  if (p2.edad() > p1.edad())  
    mayor = p2;  
  else {  
    if (p3.edad() > p1.edad())  
      mayor = p3;  
  }  
  console.log('Soy una arrow function. Atte esMayorDeEdad');  
  return mayor;  
};
```

Se pude verificar la ejecución con:

```
node js/arrow-functions.js #path a archivo
```

Básicamente las **arrow functions** son otra forma de escribir funciones con una sintaxis más simplificada, lo que normalmente conocemos como **expresadas**.

Existen algunas diferencias con las funciones declaradas y también algunas limitaciones, tal como se enuncia a continuación:

- No hay que usar **arrow function** cuando se está declarando metodos dentro de una clase o definición de objetos.
- No se puede usar **arrow functions** dentro de constructores.
- Si tienen solo una línea es posible omitir las {} y escribirla en una sola línea
- Si tienen un solo parámetro es posible omitir los ()
- Mediante **this** tienen acceso al contexto de su declaración
- Tienen un return implícito
- Son utilizadas dentro los iteradores de conjuntos

Nota

Si bien hay más limitaciones, solo se mencionan éstas ya que se relacionan con los puntos abordados hasta el momento. Para mas info ir a [Arrow function expressions](#).

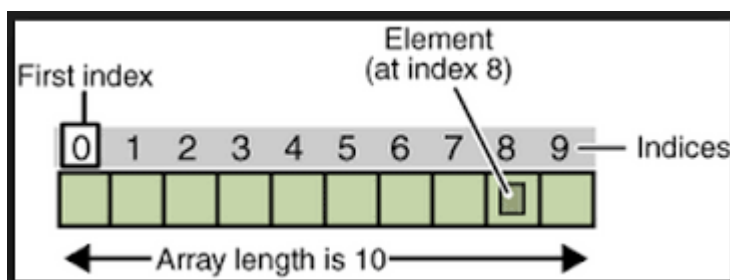
Bibliografía o Referencia

- [JavaScript Guide - Mozilla Developer Network => Introduction](#)
- [Funciones flecha](#)

Clase 3 - Arrays

Arrays o vectores (*)

Un arreglo es un conjunto de datos ordenados por posiciones, asociadas en una variable. Los valores se delimitan por []. Gráficamente:



Los valores pueden ser del mismo tipo o de distinto tipo. Algunos ejemplos en [array.js](#):

```
const array = ['hola', 3.14, true];  
console.log('array:', array); // array: [ 'hola', 3.14, true ]
```

```
const array = new Array(3); // Crea tres posiciones, donde cada elemento tiene
                             como valor por defecto `undefined`
console.log('array con elementos undefined', array); // array con elementos
undefined [ <3 empty items> ]
```

```
const array = new Array('hola', 3.14, true);
console.log('array:', array); // array: [ 'hola', 3.14, true ]
```

Ejecutar ejemplos desde la consola

```
nodejs js/array.js #ruta al archivo a ejecutar
```

Manipulando arrays (*)

- Para acceder por índice: `array[0]` = primer elemento. En JS los arreglos son **0-index**. Si se utiliza un número de índice no válido, se obtendrá `undefined`.
- Para añadir un elemento: `array.push(nuevoElemento)`. Siempre se agregan dinámicamente al final sin límite.
- Para añadir al comienzo: `array.unshift(elemento)`. Esto devuelve el número de elementos.
- Para quitar el último elemento `array.pop()`. Esto devuelve el elemento eliminado.
- Para conocer la cantidad de elementos: `array.length`.

Importante: `length` es una propiedad de los array por lo tanto es sin paréntesis. Por ejemplo:

```
const numeros = [1, 35, 6, 54, 13, 3, 24]; // dar nombres representativos
console.log('array numeros', numeros)
console.log('primero elemento de numeros', numeros[0])
console.log('índice no válido de numeros', numeros[129])
// agrego nuevo elemento al final
numeros.push(65)
console.log('nuevo elemento al final de numeros', numeros[7])
// agrego nuevo elemento al principio de numeros
numeros.unshift(34)
console.log('nuevo elemento al final de numeros', numeros[0])
console.log(`El array números tiene ${numeros.length} elementos`);
```

Ejecutar sección `Manipulando arrays` en `array.js`

```
nodejs js/array.js #ruta al archivo a ejecutar
```

Iterando arrays (*)

Podemos usar tanto las estructuras iterativas tradicionales o Javascript nos provee de varios [metodos o funciones especiales](#) para manipular arrays. Entre ellos podemos encontrar:

Todas las funciones especiales ejecutan una función sobre los elementos del array. La sintaxis corresponde a funciones flecha.

Notar que la variable auxiliar **e** puede nombrarse como el desarrollador prefiera.

Dado el siguiente array:

```
const numeros = [1, 35, 6, 54, 13, 3, 24]; // dar nombres representativos
```

Podemos ejecutar algunos ejemplos en la sección [Iterando arrays](#)

- **forEach**: Recorrer los cada elemento. Como argumento del forEach va una funcion, por ejemplo

```
numeros.forEach(elemento => console.log(elemento));
```

Nota: En este caso la arrow function `e => console.log(e)` solo imprime por consola lo que le llega por argumento.

- **filter**: Filtrar los elementos, devuelve un array nuevo. No modifica el actual.

```
const mayoresA10 = numeros.filter((e) => e > 10);  
console.log(`Mayores a 10`, mayoresA10);  
console.log(`Numeros`, numeros);  
console.log(`La cantidad de elementos mayores a 10 son ${mayoresA10.length}`);
```

Nota: Notese que la arrow function pasada por parametro a `filter` tiene que retornar `true` para que el elemento e sea insertado en el array que se va a retornar.

- **map**: Mapea cada elemento del array y genera un array nuevo a partir de cada elemento mapeado. Por ejemplo

```
const numerosX10 = numeros.map((e) => e * 10);  
console.log(`elementos mapeados`, numerosX10);  
console.log(`Numeros`, numeros);  
console.log(`La cantidad de elementos mayores a 10 son ${numerosX10.length}`);
```

Ejercitación 1

Práctica 3.1: Se ha solicitado un script que permita cargar las precipitaciones promedio en cada mes del país (en nuestro caso serán generadas de manea aleatorias con valores comprendidos entre [15;35]°), en base a esos datos determinar:

- el promedio anual de lluvias
- el promedio de lluvias para el segundo semestre del año
- el mes más seco del año [Práctica 3.1.js](#)

Ejercitación 2: Personas

Crear un array con al menos 8 personas. Usar la clase persona previamente definida. Definir funciones que reciban como argumento el array vuelvan:

1- Personas mayores de edad: mayoresDeEdad(personas)

2- Persona cuya profesión sea una pasada por argumento, ej: personasXProfesion(personas, 'superheroe')

3- obtenerPersonaMasGrande(personas), tener en cuenta que ahora recibe como argumento un array con n personas.

4- obtenerProfesiones(personas). // sin duplicados

Bibliografía o Referencia

- [JavaScript Guide - Mozilla Developer Network => Introduction](#)
- [metodos o funciones de arrays](#)
- [sort](#)

Clase 3 - Módulos

¿Cuándo empezamos a usar módulos?

Como pasa en todos los sistemas de información y en distintos lenguajes el código se va incrementando. Si desde el principio no se ha definido una estructura a seguir y se ha escrito todo el código en el mismo archivo, el código escrito deja de inteligible, pasando a ser poco matentible e inflexible a cambios, entre otros.

Para dividir el código código segun cierta "funcionalidad" o enfoque arquitécnico es que javascript nos proporciona la capacidad de separar en **modulos** nuestro código. Por ejemplo la carpeta modulos tiene la siguiente estructura:

Estructura de [modulos](#)

```
modulos
├── acceso-a-datos
│   └── personas.datos.js
├── entidades
│   └── persona.js
├── logica-de-negocio
│   └── personas.servicio.js
├── index.js
└── package.json
```

- En `acceso-a-datos` se encuentran los archivos con los datos.
- En `entidades` tiene la definición de los tipos de datos usados.
- En `logica-de-negocio` tiene las funciones necesarias para cumplir con los requerimientos pedidos.

Ahora bien, hay un archivo `package.json` (generado automáticamente mediante el comando `npm init`) que contiene una estructura definida muy concreta, y a través de sus campos se puede guardar y recuperar información muy útil. Por el momento la sintaxis y extensión del archivo lo vamos a dejar para más adelante.

```
{
  "type": "module"
}
```

Lo que si es importante, es que por medio de este archivo se le puede indicar a nodejs si se desea que se interprete los archivos usando las convenciones `CommonJS` o usando al sintaxis del `módulo ES`.

Actualmente con el valor `type: module` estamos usando sintaxis de ES6, para mostrar las sentencias `export` e `import`. Si este archivo esta ausente, por defecto nodejs interpreta con `CommonJS`, que son las convenciones comunes para Javascript que corren tanto en el navegador como con NodeJS.

```
{
  "type": "commonjs"
}
```

Para poder "exportar" variables, funciones, tipos de datos es importante usar la sentencia `export` en el archivo donde se declara lo que se desea exportar. Por ejemplo se exporta la variable `personasServicio` de `persona.servicio.js`:

```
const personasServicio = {
  mayoresDeEdad: mayoresDeEdad,
  obtenerPersonaMasGrande: obtenerPersonaMasGrande,
  personasXProfesion: personasXProfesion,
  obtenerProfesiones: _obtenerProfesiones
};

export { personasServicio };
```

Y se lo importa en `index.js`:

```
import { personas } from './acceso-a-datos/personas.datos.js';
import { personasServicio } from './logica-de-negocio/personas.servicio.js'
```

Importante: incluir la extensión `.js` en las rutas relativas para **importar** el contenido de otro módulo.

Para usarla en la línea 9 de ese mismo archivo [index.js](#):

```
const mayores = personasServicio.mayoresDeEdad(personas);
```

De esta forma evitamos tener todo el código en un solo archivo y separado las funciones de los tipos de datos y de los datos en si mismos. Para ejecutar nuestro archivo [index.js](#)

```
node js/modulos/index.js #ruta al archivo a ejecutar
```

Ejercitación: Personas (Continuación)

Crear un array con al menos 8 personas. Usar la clase persona previamente definida. Definir funciones que reciban como argumento el array vuelvan:

1- Personas mayores de edad: mayoresDeEdad(personas)

2- Persona cuya profesión sea una pasada por argumento, ej: personasXProfesion(personas, 'superheroe')

3- obtenerPersonaMasGrande(personas), tener en cuenta que ahora recibe como argumento un array con n personas.

4- obtenerProfesiones(personas). // sin duplicados

Bibliografía o Referencia

- [CommonJS](#)
- [NodeJS - package.json](#)
- [js-examples](#)

Clase 3 - JSON

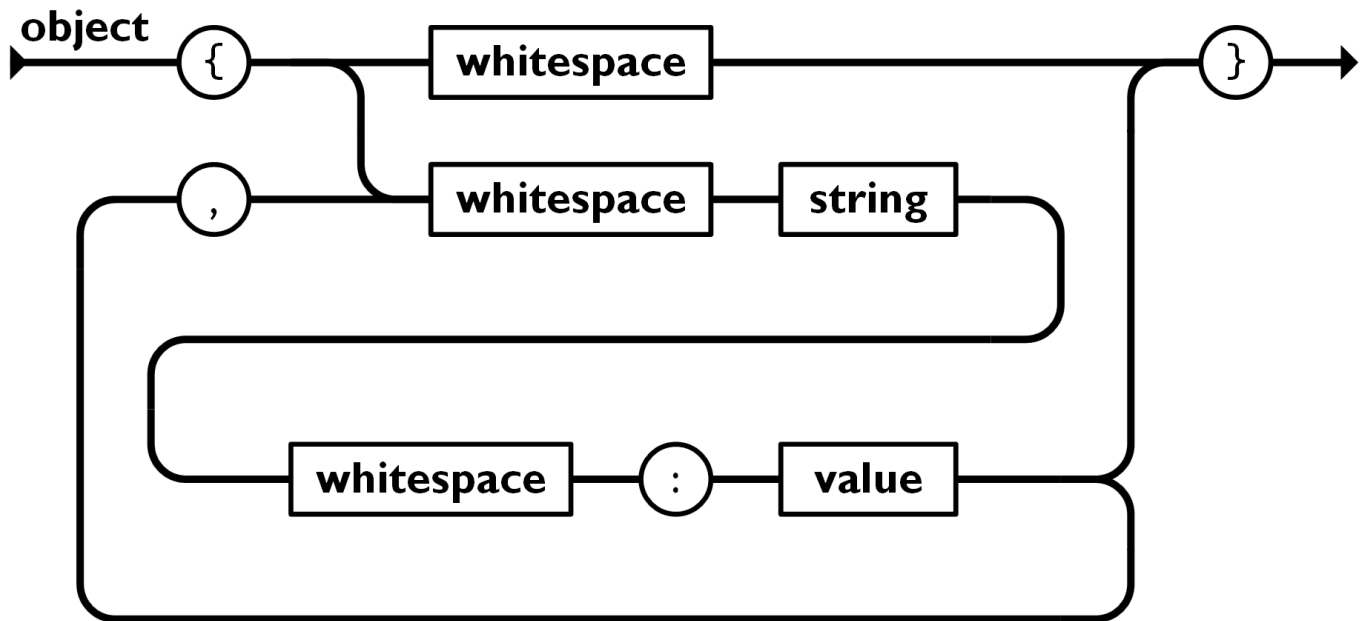
Repasando módulos

Estructura de [modulos-con-json](#)

```
modulos-con-json
├── acceso-a-datos
│   └── personas.datos.js
├── entidades
│   └── persona.js
├── logica-de-negocio
│   └── personas.servicio.js
├── index.js
└── package.json
```

En esta estructura hay un archivo `package.json` cuya extensión es `.json`. **JSON** (acrónimo de **JavaScript Object Notation**) es un formato de texto sencillo para el intercambio de datos. Entre las estructuras mas comunes en un JSON podemos encontrar: objetos o array.

Objetos JSON (*)



Por ejemplo un objeto sencillo es el del ejemplo `package.json` que tiene un unico par `clave-valor`, su clave es `type` y su valor es `module`.

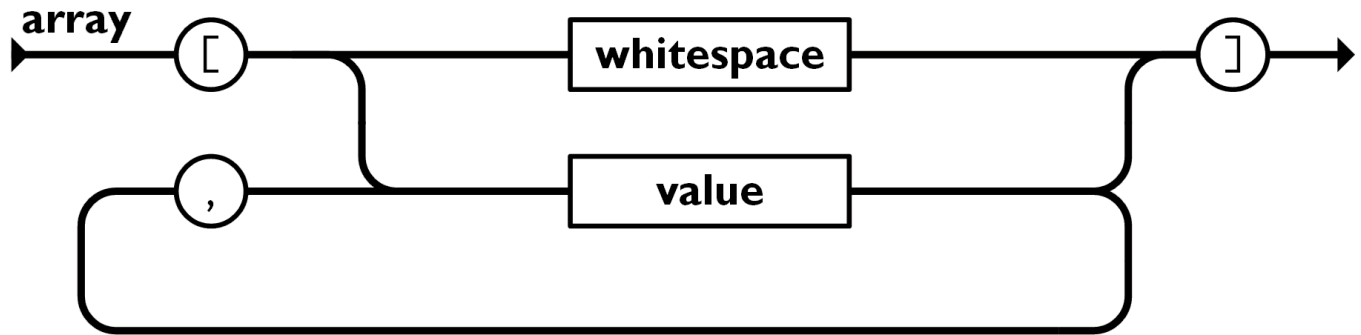
```
{  
  "type": "module"  
}
```

Otro ejemplo, con Persona:

```
{  
  "Id": 4,  
  "nombre": "Maria",  
  "correo": "maria@gmail.com",  
  "profesion": "Medico clínico",  
  "fechaNacimiento": "2016-07-01T03:00:00.000Z"  
}
```

Array JSON

En los ejemplos anteriores el `valor` en cada par `clave-valor` ha sido un string o un un número, pero también puede ser otro objeto.

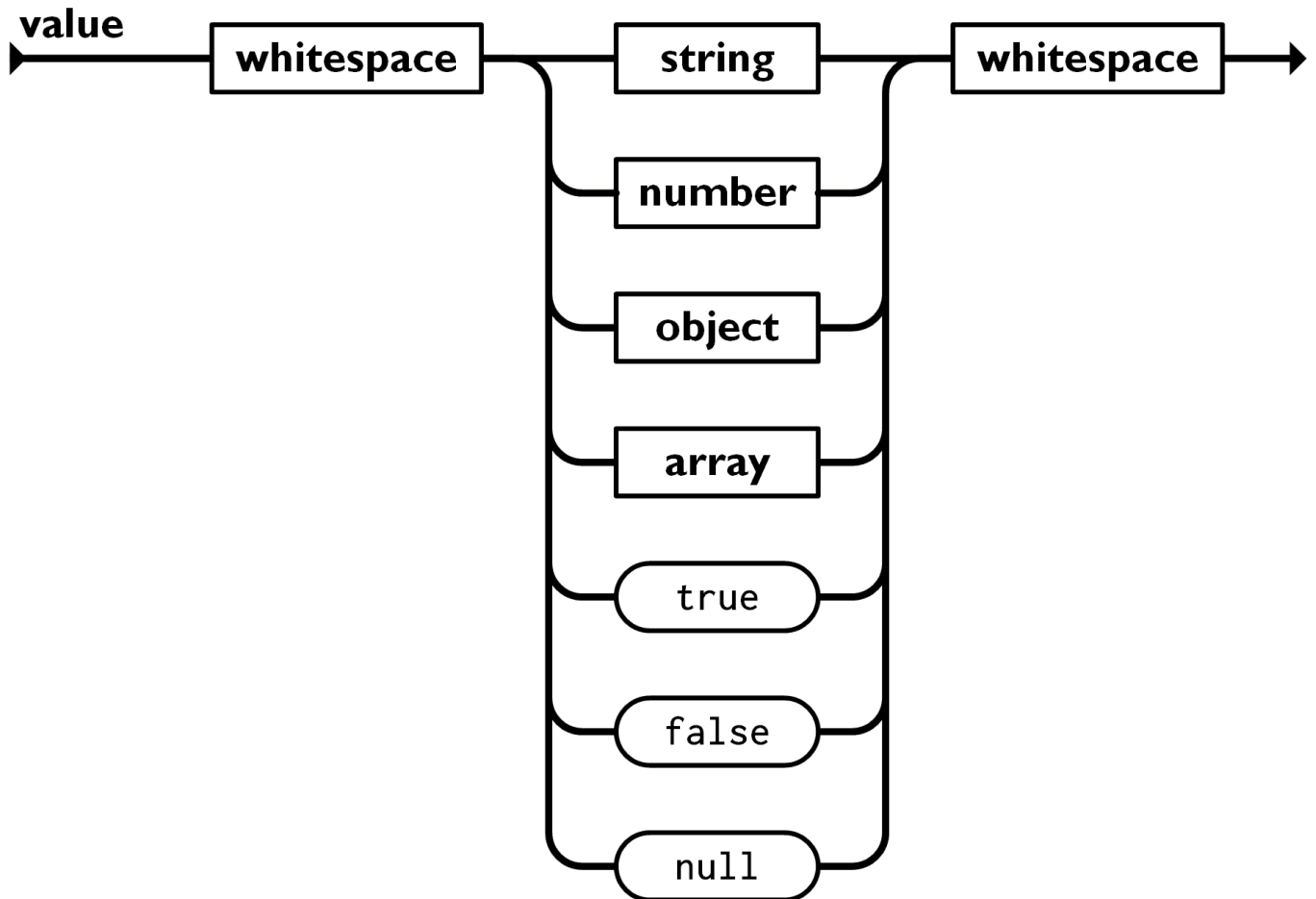


Por ejemplo, ver ejemplo completo en [personas.json](#):

```
[
  {
    "Id": 4,
    "nombre": "Maria",
    "correo": "maria@gmail.com",
    "profesion": "Medico clínico",
    "fechaNacimiento": "2016-07-01T03:00:00.000Z"
  },
  {
    "Id": 5,
    "nombre": "Milagros",
    "correo": "milagros@gmail.com",
    "profesion": "Vigilante",
    "fechaNacimiento": "2015-07-01T03:00:00.000Z"
  }
]
```

Valores en JSON (*)

Cuando hablamos de **valor**, un valor puede contener cualquiera de los siguientes tipos de datos:



Por ejemplo, suponiendo que una persona no tiene correo el JSON quedaría así:

```
{
  "Id": 4,
  "nombre": "Maria",
  "correo": null,
  "profesion": "Medico clínico",
  "fechaNacimiento": "2016-07-01T03:00:00.000Z"
}
```

Ahora, si en lugar de tener **profesion**, tiene profesiones quedaría:

```
{
  "Id": 4,
  "nombre": "Maria",
  "correo": null,
  "profesion": ["Medico clínico", "Superheroe"],
  "fechaNacimiento": "2016-07-01T03:00:00.000Z"
}
```

Suponiendo que las profesiones están tipificadas y cada profesión tiene su id, el JSON quedaría de la siguiente forma. También supongamos que este JSON tiene que representar la propiedad tienePasaporte,

seria de la siguiente forma

```
{
  "Id": 4,
  "nombre": "Maria",
  "correo": null,
  "profesiones": [
    {
      "id": 3,
      "nombre": "Medico clínico"
    },
    {
      "id": 4,
      "nombre": "Superheroe"
    }
  ],
  "fechaNacimiento": "2016-07-01T03:00:00.000Z",
  "tienePasaporte": true
}
```

Leyendo JSONs con ES6

```
import personas from './personas.json' assert { type: 'json' };
```

El assert permite incluir información adicional. Esto es porque tenemos definido en el `package.json` que use ECMAScript para interpretar los archivos. Si tuvieramos en type `CommonJS`, la importación de un JSON sería

```
const personas = require('./personas.json')
```

Para ver las diferencias de importación vamos importar con el tipo `commonjs`. Por ultimo, para ejecutar el ejemplo podemos correr el archivo `js/modulos-con-json/index.js` que lee los archivos del `personas.json`

```
node js/modulos-con-json/index.js #ruta al archivo a ejecutar
```

Ejercitación: Personas (Continuación)

Leer un archivo .json y crear un array a partir de sus objetos. Definir funciones que reciban como argumento el array vuelvan:

1- Personas mayores de edad: `mayoresDeEdad(personas)`

2- Persona cuya profesión sea una pasada por argumento, ej: `personasXProfesion(personas, 'superheroe')`

3- obtenerPersonaMasGrande(personas), tener en cuenta que ahora recibe como argumento un array con n personas.

4- obtenerProfesiones(personas). // sin duplicados

Bibliografía o Referencia

- [JSON](#)
- [ECMAScript - tc39/proposal-import-assertions](#)
- [Import JSON in ES modules](#)
- [Ver jsons formateados](#)

Clase 3 - JSON

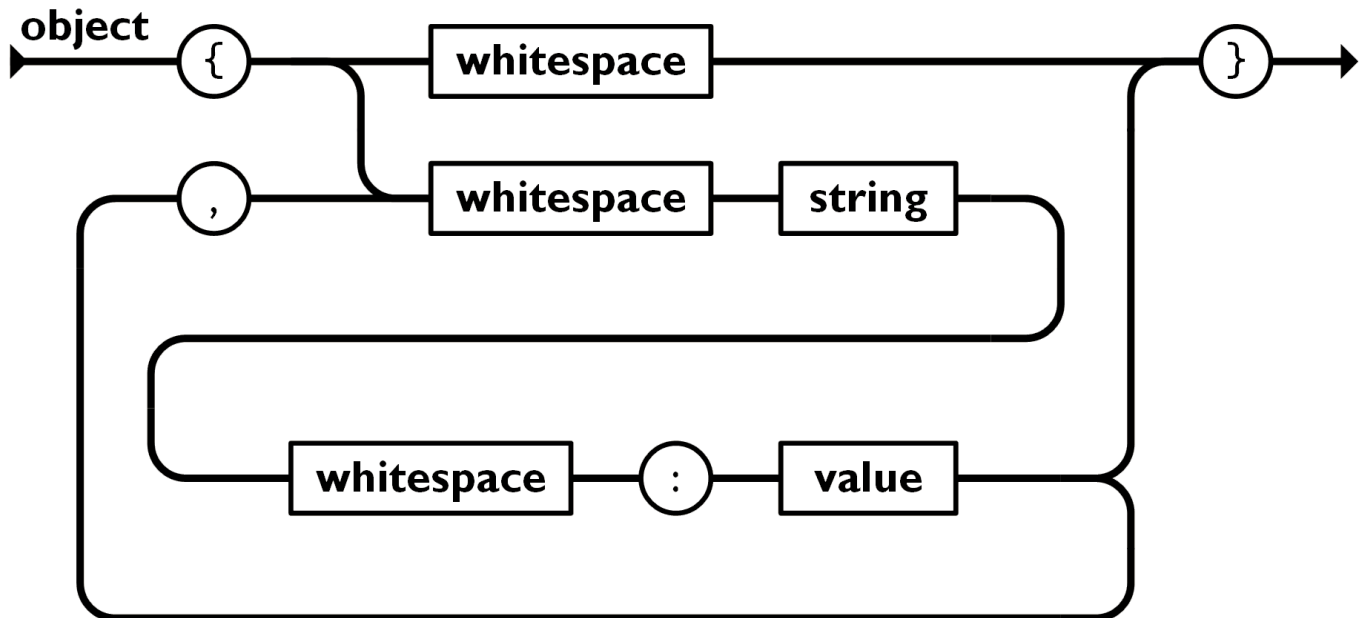
Repasando módulos

Estructura de [modulos-con-json](#)

```
modulos-con-json
├── acceso-a-datos
│   └── personas.datos.js
├── entidades
│   └── persona.js
├── logica-de-negocio
│   └── personas.servicio.js
├── index.js
└── package.json
```

En esta estructura hay un archivo `package.json` cuya extensión es `.json`. **JSON** (acrónimo de **JavaScript Object Notation**) es un formato de texto sencillo para el intercambio de datos. Entre las estructuras mas comunes en un JSON podemos encontrar: objetos o array.

Objetos JSON (*)



Por ejemplo un objeto sencillo es el del ejemplo `package.json` que tiene un unico par **clave-valor**, su clave es **type** y su valor es **module**.

```

{
  "type": "module"
}

```

Otro ejemplo, con Persona:

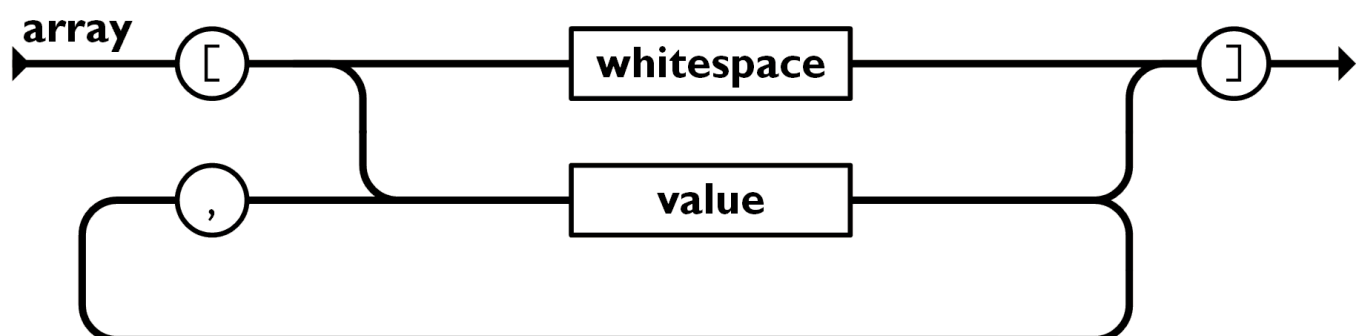
```

{
  "Id": 4,
  "nombre": "Maria",
  "correo": "maria@gmail.com",
  "profesion": "Medico clínico",
  "fechaNacimiento": "2016-07-01T03:00:00.000Z"
}

```

Array JSON

En los ejemplos anteriores el **valor** en cada par **clave-valor** ha sido un string o un un número, pero también puede ser otro objeto.

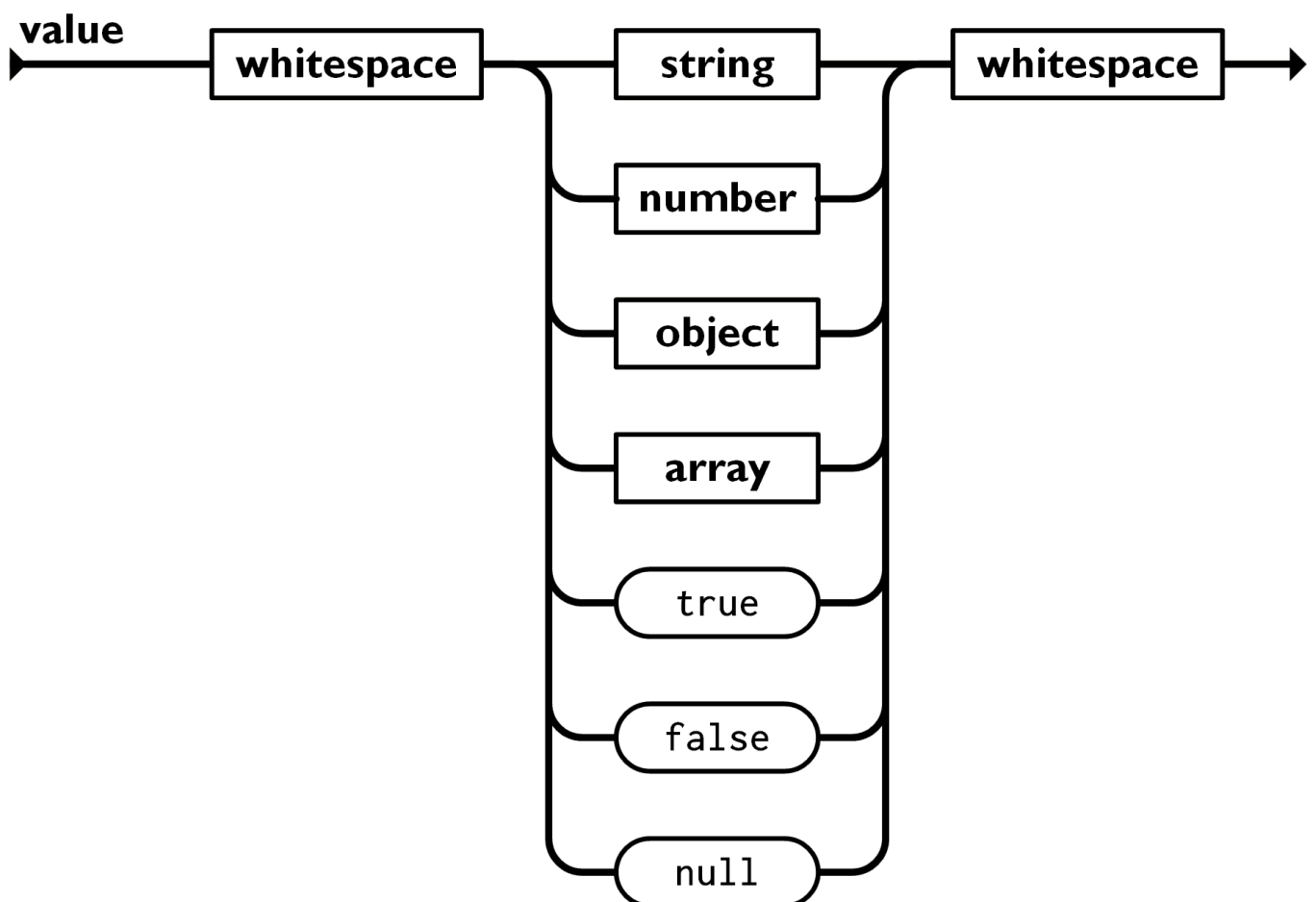


Por ejemplo, ver ejemplo completo en [personas.json](#):

```
[
  {
    "Id": 4,
    "nombre": "Maria",
    "correo": "maria@gmail.com",
    "profesion": "Medico clínico",
    "fechaNacimiento": "2016-07-01T03:00:00.000Z"
  },
  {
    "Id": 5,
    "nombre": "Milagros",
    "correo": "milagros@gmail.com",
    "profesion": "Vigilante",
    "fechaNacimiento": "2015-07-01T03:00:00.000Z"
  }
]
```

Valores en JSON (*)

Cuando hablamos de **valor**, un valor puede contener cualquiera de los siguientes tipos de datos:



Por ejemplo, suponiendo que una persona no tiene correo el JSON quedaría así:

```
{
  "Id": 4,
```

```
"nombre": "Maria",
"correo": null,
"profesion": "Medico clínico",
"fechaNacimiento": "2016-07-01T03:00:00.000Z"
}
```

Ahora, si en lugar de tener `profesion`, tiene profesiones quedaría:

```
{
  "Id": 4,
  "nombre": "Maria",
  "correo": null,
  "profesion": ["Medico clínico", "Superheroe"],
  "fechaNacimiento": "2016-07-01T03:00:00.000Z"
}
```

Suponiendo que las profesiones estan tipificadas y cada profesión tiene su id, el JSON quedaría de la siguiente forma. También supongamos que este JSON tiene que representar la propiedad `tienePasaporte`, seria de la siguiente forma

```
{
  "Id": 4,
  "nombre": "Maria",
  "correo": null,
  "profesiones": [
    {
      "id": 3,
      "nombre": "Medico clínico"
    },
    {
      "id": 4,
      "nombre": "Superheroe"
    }
  ],
  "fechaNacimiento": "2016-07-01T03:00:00.000Z",
  "tienePasaporte": true
}
```

Leyendo JSONs con ES6

```
import personas from './personas.json' assert { type: 'json' };
```

El `assert` permite incluir información adicional. Esto es porque tenemos definido en el `package.json` que use ECMAScript para interpretar los archivos. Si tuviéramos en `type CommonJS`, la importación de un JSON sería

```
const personas = require('./personas.json')
```

Para ver las diferencias de importación vamos a importar con el tipo `commonjs`. Por último, para ejecutar el ejemplo podemos correr el archivo `js/modulos-con-json/index.js` que lee los archivos del `personas.json`

```
node js/modulos-con-json/index.js #ruta al archivo a ejecutar
```

Ejercitación: Personas (Continuación)

Leer un archivo `.json` y crear un array a partir de sus objetos. Definir funciones que reciban como argumento el array y devuelvan:

- 1- Personas mayores de edad: `mayoresDeEdad(personas)`
- 2- Persona cuya profesión sea una pasada por argumento, ej: `personasXProfesion(personas, 'superheroe')`
- 3- `obtenerPersonaMasGrande(personas)`, tener en cuenta que ahora recibe como argumento un array con `n` personas.
- 4- `obtenerProfesiones(personas)`. // sin duplicados

Bibliografía o Referencia

- [JSON](#)
- [ECMAScript - tc39/proposal-import-assertions](#)
- [Import JSON in ES modules](#)
- [Ver jsons formateados](#)

Clase 3 - Recomendaciones

Código Limpio y programación defensiva (Desarrollo seguro)

Nombre de variables:

```
// [X] No representativo  
const array = [1, 35, 6, 54, 13, 3, 24];
```

```
// [X] Redundante  
const listaNumeros = [1, 35, 6, 54, 13, 3, 24];
```

```
// [✓] ok
const numeros = [1, 35, 6, 54, 13, 3, 24];
```

Empezando a pensar en programación defensiva

Cuando creamos funciones, considerar casos en los que pueda **explotar**

```
function _obtenerProfesiones(personas) {
  const profesiones = personas.map((p) => p.profesion);
  const profesionesSinDuplicados = [];
  for (const profesion of profesiones) {
    // console.log(profesion, profesionesSinDuplicados.some(p =>
    p.toLowerCase() === profesion.toLowerCase()))
    if (!profesionesSinDuplicados.some(p => p.toLowerCase() ===
    profesion.toLowerCase()))
      profesionesSinDuplicados.push(profesion);
  }
  return profesionesSinDuplicados;
}
```

Por ejemplo:

- ¿Qué pasa si personas es **null** o **undefined**?
- ¿Qué pasa si en profesión viene algo distinto a un string?
- ¿Qué pasa si en edad viene algo distinto a un numero?

Considerando algunos casos de prueba podemos describir nuestra función:

```
function _obtenerProfesiones(personas) {
  if (!personas) { // true si personas == null o == undefined
    console.warn('No se cargaron personas.')
    return [];
  }
  // console.log(typeof 4324);
  // console.log(typeof 'asadad');
  // console.log('typeof 4324 !== \'string\' ', typeof 4324 !== 'string');
  const profesiones = personas.map((p) => {
    if (typeof p.profesion === 'string')
      return p.profesion;
    console.warn(`La persona ${p.nombre} tiene una profesión no válida:
    ${p.profesion}`)
    return null;
  });
  // console.log('profesiones', profesiones);
  const profesionesSinDuplicados = [];
  for (const profesion of profesiones) {
    // console.log(profesion, profesionesSinDuplicados.some(p =>
    p.toLowerCase() === profesion.toLowerCase()))
```

```
        if (!!profesion && !profesionesSinDuplicados.some(p => p.toLowerCase() ===
profesion.toLowerCase()))
            profesionesSinDuplicados.push(profesion);
    }
    return profesionesSinDuplicados;
}
```

Casos de prueba, en ningun caso deberia explotar:

```
personas = undefined; // falla
```

```
personas = []; // ok
```

```
personas.push(new Persona('Luis miguel', 'luismi@gmail.com', 234324, new
Date(1965, 6, 1))); // falla
```

Bibliografía o Referencia