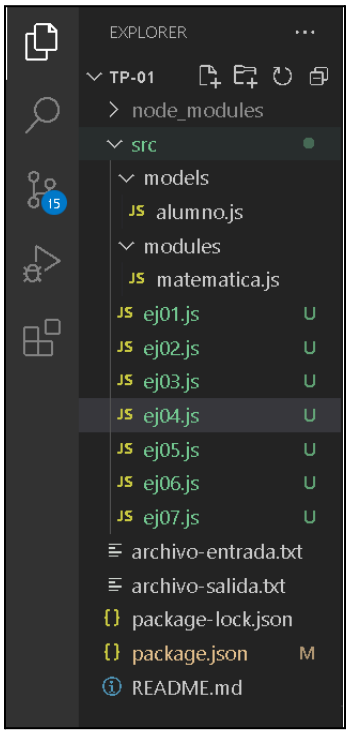
OBJETIVO

Vamos a crear una aplicación con NodeJs, con el objetivo de probar métodos de las clases `console`, utilizaremos módulos nativos y crearemos módulos propios. Son **8** Ejercicios en total.
Modificando el en la parte de scripts en el `package.json` para que se vaya ejecutando cada uno de los programas por separado.




```
...
    "ej01": "node src/ej01.js",
    "ej02": "node src/ej02.js",
...
```

Lo probamos con:

```
C:\>npm run ej01
```

Utilizar el siguiente comando para crear el proyecto:

```
C:\>npm init
```

ESTRUCTURA DEL PROYECTO.

La estructura del proyecto debe ser similar a la imagen que se muestra acá:

- \
- \src
- \src\models
- \src\modules

EJ01: Desarrollar una aplicación en la que podamos utilizar el `console.log()` haciendo pruebas con dos strings, concatenarlos e invertirlos ("Escuela" y "ORT" retorna -> "TROaleucsE")
Ejemplo:

```
let textoEntrada01 = "Escuela", textoEntrada02 = "ORT";
let textoSalida    = "";

textoSalida = concatInvert(textoEntrada01, textoEntrada02);
console.clear();  // Borra la pantalla en la consola.
console.log(`Textos de Entrada: "${textoEntrada01}" y "${textoEntrada02}"`);
console.log(`Texto de Salida: "${textoSalida}"`);

function concatInvert (texto1, texto2){
    let returnValue = '';
    // No seas vago, acá hay que hacer el cuerpo de la función!!!
    return returnValue;
}
```

Salida por consola:

```
Textos de Entrada  : "Escuela" y "ORT"
Texto de Salida    : "TROaleucsE"
```

Ayudita:

Para acceder a una posición específica de un string utiliza `texto[i]`. Podés saber la cantidad de caracteres utilizando `texto.length`. El primer elemento de un array es siempre 0. También existen otras funciones de la clase string como `split`, `concat`, etc.

Existen otras funciones de la clase Array, por ejemplo `concat`, `reverse`, `join`, etc.

EJ02: Crear un módulo matemática **`/src/modules/matematica.js`** que tenga 4 funciones **`sumar`**, **`restar`**, **`multiplicar`** y **`dividir`** (dos de ellas arrow functions), una constante (**`PI`**) y un Array con 4 string que tengan este contenido **`["dos", "cuatro", "ocho", "diez"]`**.
Exportarlas y utilizarlas en el programa principal. en el caso del array, mostrar todos sus elementos.
Ejemplo:

```
/* Importo la constante PI y la función sumar del módulo matematica. */

import {PI, sumar} from './src/modules/matematica.js';

let total, numero1=10, numero2=20;
console.clear();
console.log(`La constante PI vale '${PI}'`);      // Uso la constante PI importada.

total = sumar(numero1, numero2);                  // Uso la función sumar importada.
console.log(`sumar(${numero1}, ${numero2}) =  ${total}`);
...
```

```
/* Este es el módulo "matematicas" */
const PI = 3.14;

function sumar(x, y) {
  // No seas vago, acá hay que hacer el cuerpo de la función!!!
}

const multiplicar = (a, b) => {
  // No seas vago, acá hay que hacer el cuerpo de la función!!!
};

// Exporto todo lo que yo quiero exponer del módulo hacia el exterior.
export {PI, sumar, multiplicar};
```

Ayudita:

Mira al final como crear y utilizar un módulo.

EJ03: Crear la clase `Alumno` en **`/src/models/alumno.js`** que tenga tres propiedades (**`username, DNI y edad`**) y también un método **`toString()`** que retorne la información del alumno (string).
Instanciar en un programa principal tres objetos y mostrarlos en la consola.

```
import Alumno from './src/models/alumno.js';

let alumno1 = new Alumno("Esteban Dido"  , "22888444", 20);
let alumno2 = new Alumno("Matias Queroso", "28946255", 51);
let alumno3 = new Alumno();

alumno3.username    = "Elba Calao";
alumno3.dni         = "32623391";

console.clear();
console.log(alumno1);
console.log(alumno2);
console.log(alumno3.toString());
```

Ayudita:

Mira al final como crear y utilizar un módulo de clase.

EJ04: Desarrollar una aplicación que utilizando el módulo interno **`fs`** (FileSystem) haga un programa que lea un archivo y lo escriba con otro nombre. Desarrollarlo en forma de función.
Por ejemplo:

```
import fs from 'fs';

const ARCHIVO_ENTRADA           = "./archivo-entrada.txt";
const ARCHIVO_SALIDA            = "./archivo-salida.txt";
console.clear();
copiar(ARCHIVO_ENTRADA, ARCHIVO_SALIDA);
function copiar(origen, destino){
    // No seas vago, acá hay que hacer el cuerpo de la función!!!
}
```

Ayudita:

Mira de que se trata el módulo de FileSystem de node, y googlea como usarlo.
https://nodejs.org/api/fs.html

EJ05: Desarrollar una aplicación que utilizando el módulo interno **`url`**, invoque a una función nuestra que se llame **`parsearUrl(url)`**, que dada una url retorna un objeto con las siguientes propiedades.
Se debe utilizar la clase **URL** del módulo url, ver la documentación:
https://nodejs.org/api/url.html#the-whatwg-url-api
Objeto a retornar:

```
{
  "host"       : "",
  "pathname"   : "",
  "parametros" : {}
}
```

Ejemplo:

```
let miUrl    = null;
let miObjeto = null;
miUrl    = 'http://www.ort.edu.ar:8080/alumnos/index.htm?curso=2022&mes=mayo';
miObjeto = parsearUrl (miUrl);
console.log(miObjeto);

function parsearUrl(laURL){
    // No seas vago, acá hay que hacer el cuerpo de la función!!!
}
```

Salida por consola:

```
{
      host: 'http://www.ort.edu.ar:8080',
      pathname: '/alumnos/index.htm',
      parametros: URLSearchParams { 'curso' => '2022', 'mes' => 'mayo' }
}
```

Ayudita:

Mira cómo se usa la clase URL, especialmente cómo se construye (constructor).
https://nodejs.org/api/url.html

EJ06: Modificar el ejercicio anterior para que en caso de que dentro de la función **`parsearUrl(url)`** se produzca una excepción (por ejemplo un nombre de URL invalida), muestre la excepción en la consola y retorna un objeto con sus propiedades nulas o un objeto vacío (según corresponda)
Nota: Se acuerdan del try/catch?

EJ07: Realizar un programa que utilizando una biblioteca existente (buscarla en https://www.npmjs.com/) podamos hacer una función en la que le enviamos un nombre de un país (abreviado) y nos retorne el nombre de la moneda que utiliza el mismo.
En el caso de que el nombre del país no exista, debe retornar null.

Instalar una dependencia:

Desde la carpeta raíz del proyecto, donde se encuentra el `package.json` debemos tipear:

```
C:\DAI\TP02>npm i currency-map-country
```

El `package.json` queda modificado, se ha agregado la dependencia.

```
{
  ...
  "dependencies": {
    "currency-map-country": "^1.0.12"
  },
  ...
}
```

Ejemplo:

```
let monedaDelPais, codigoPais;
codigoPais = 'AR';
monedaDelPais = obtenerMoneda(codigoPais);
console.log(`La moneda del país ${codigoPais} es: ${monedaDelPais}`);

codigoPais = 'UZA';
monedaDelPais = obtenerMoneda(codigoPais);
console.log(`La moneda del país ${codigoPais} es: ${monedaDelPais}`);
```

Salida por consola:

```
La moneda del país AR es : ARS
La moneda del país UZA es: null
```

Nota: Utilizar el paquete **currency-map-country**. Ver su utilización en npm:
https://www.npmjs.com/package/currency-map-country

EJ08: Realizar un módulo “wrapper” (envolvedor, como el wrap de pollo!) de la api de OMDB (https://www.omdbapi.com/) que utilizamos con postman.
La idea es que nuestro módulo exponga tres funciones y que internamente invoque a la API de OMDB y nos retorne el resultado. Para realizar los request a la api desde node vamos a utilizar axios.

Instalar la dependencia:

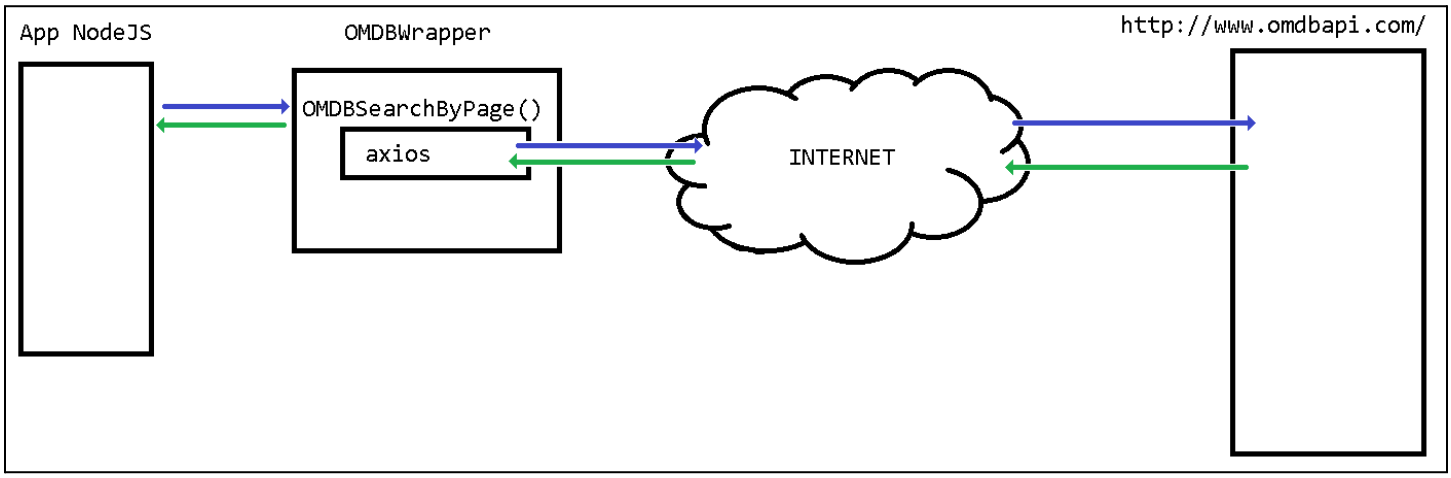
Desde la carpeta raíz del proyecto, donde se encuentra el `package.json` debemos tipear:

```
C:\DAI\TP02>npm i axios
```

El `package.json` queda modificado, se ha agregado la dependencia.

```
{
  ...
  "dependencies": {
    "axios": "^1.6.8"
  },
  ...
}
```

Esta es la idea de lo que estamos haciendo::



El módulo `omdb-wrapper.js` debe quedar así:

```
/* Módulo  OMDBWrapper*/
import axios from "axios";

const APIKEY          = "7b62fa5d"; // Poné tu APIKEY, esta no funciona.

const OMDBSearchByPage = async (searchText, page = 1) => {
  let returnObject = {
      respuesta     : false,
      cantidadTotal : 0,
      datos         : {}
    };

  // No seas vago, acá hay que hacer el cuerpo de la función!!!
  return returnObject;
};

const OMDBSearchComplete = async (searchText) => {
  let returnObject = {
      respuesta     : false,
      cantidadTotal : 0,
      datos         : {}
    };

  // No seas vago, acá hay que hacer el cuerpo de la función!!!
  return returnObject;
};

const OMDBGetByImdbID = async (imdbID) => {
  let returnObject = {
      respuesta     : false,
      cantidadTotal : 0,
      datos         : {}
    };

  // No seas vago, acá hay que hacer el cuerpo de la función!!!
  return returnObject;
};

// Exporto todo lo que yo quiero exponer del módulo:
export {OMDBSearchByPage, OMDBSearchComplete, OMDBGetByImdbID};
```

UTILIZACIÓN DE AXIOS.

Ejemplo de index.js (nótese el **async** / **await**):

```
import axios from "axios";

let respuesta = await Test();
console.log('respuesta', respuesta);

const Test = async (searchText) => {
  const requestString = `https://www.omdbapi.com/?apikey=7c62gb5e&s=cars`;
  const apiResponse = await axios.get(requestString);
  return apiResponse.data;
};
```

Salida por consola:

```
respuesta {
    Search: [
      {
        Title: 'Cars',
        Year: '2006',
        imdbID: 'tt0317219',
        Type: 'movie',
        Poster: 'https://m.media-amazon.com/image300.jpg'
      },
      ...
      {
        Title: 'Counting Cars',
        Year: '2012–',
        imdbID: 'tt2338096',
        Type: 'series',
        Poster: 'https://m.media-amazon.com/images3._V1_SX300.jpg'
      }
    ],
    totalResults: '358',
    Response: 'True'
}
```

Nota: Utilizar el paquete **axios** https://www.npmjs.com/package/axios

Require() vs Import:

En Node.js, require() es una función utilizada para importar módulos y dependencias en tu código. Por otro lado, import es una característica introducida en ECMAScript 6 (también conocido como ES6) para importar módulos en JavaScript moderno. La principal diferencia entre require() e import radica en su sintaxis y en cómo funcionan por debajo.

Ejemplo **require**:

```
const module = require('module');
```

Ejemplo **import**:

```
import module from 'module';
```

En el caso de **import**, también podemos utilizar la sintaxis de desestructuración para importar elementos específicos:

```
import unaFuncionDefault, { otraFuncion, unaConstante, ... } from 'module';
```

Para poder utilizar el **import** es necesario agregar en el package.json la siguiente línea:

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

Diferencias al momento de carga:

- **require**: Los módulos se cargan de forma sincrónica, lo que significa que se cargan cuando el código lo alcanza durante la ejecución.
- **import**: Los módulos se cargan de forma asíncrona y se evalúan en el tiempo de carga. Esto significa que los módulos se cargan antes de que se ejecute cualquier código en el módulo actual.

Diferencias de Compatibilidad:

- **require**: Es compatible con todas las versiones de Node.js.
- **import**: Es una característica de ES6 y requiere una configuración adecuada (como Babel) para ser utilizado en Node.js. A partir de Node.js v14, la importación de módulos de ES6 está habilitada por defecto, pero aún puede requerir configuración adicional dependiendo de tu entorno y requisitos de compatibilidad.

Creando una clase en Javascript:

En javascript podemos crear de la siguiente manera.

Ejemplo de la clase **Persona**:

```
export default class Persona {
    constructor(nombre, casado = false) { // casado tiene un valor default.
        this.nombre   = nombre;
        this.casado   = casado;
    }

    getNombre() {
        return this.nombre;
    }

    getCasado() {
        return this.casado;
    }

    getInformacion(){
        return `nombre:${this.nombre}, casado:${this.casado}`;
    }
}
```

Ejemplo de su utilización:

```
import Persona from './persona.js'; // Importo el módulo.

let persona1 = new Persona("Monica Galindo"  ,  true);
let persona2 = new Persona("Matias Queroso");
let persona3 = new Persona();

console.log(persona1);
console.log(persona2);
console.log(persona3.getInformacion());
```

Salida por consola:

```
Persona { nombre: 'Monica Galindo', casado: true }
Persona { nombre: 'Matias Queroso', casado: false }
nombre:undefined, casado:false
```