

## 1 - Instalación de Node.js

Node.js es una plataforma para el desarrollo de aplicaciones en JavaScript que se ejecutan fundamentalmente en un equipo servidor (pero eventualmente podemos desarrollar aplicaciones cliente).

Presenta grandes ventajas en la implementación de aplicaciones que deben responder en tiempo real como por ejemplo los juegos multijugador.

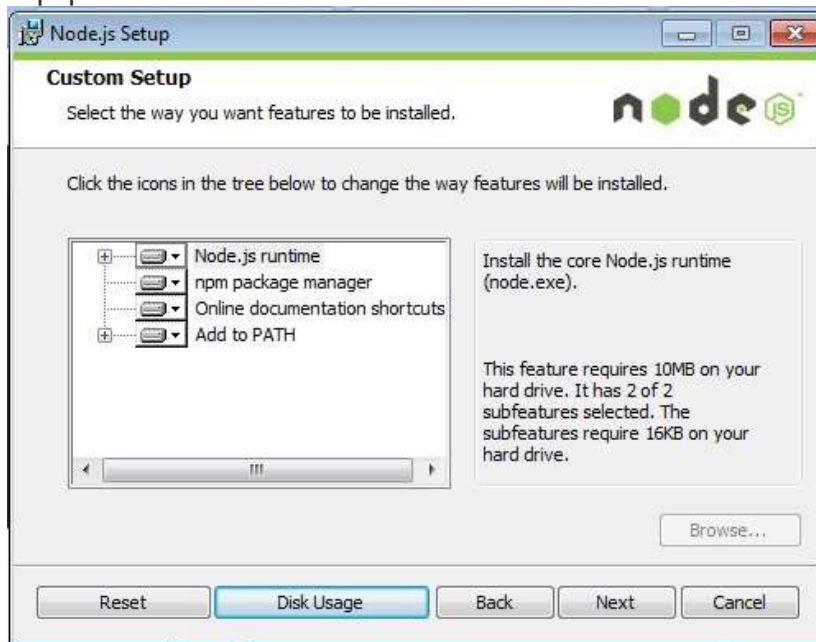
Poco sentido tiene utilizar Node.js en sitios web mayormente estáticos. Hay muchas aplicaciones actuales donde el contenido de la página varía constantemente: Twitter, Facebook, chats, juegos multijugador etc. donde Node.js puede hacer más eficiente la aplicación que se ejecuta en el servidor.

Tiene su origen por el año 2009 y toma como base el motor de JavaScript V8 desarrollado por Google.

Cuando desarrollamos aplicaciones en la plataforma Node.js utilizamos el lenguaje JavaScript para la implementación. Disponemos de numerosos módulos que nos facilitan la implementación de servidores web, administración de archivos, protocolos de comunicaciones etc.

Lo primero que haremos es instalar el Node.js que lo podemos descargar de: [aquí](#) (Hay una versión recomendada y es la que instalaremos).

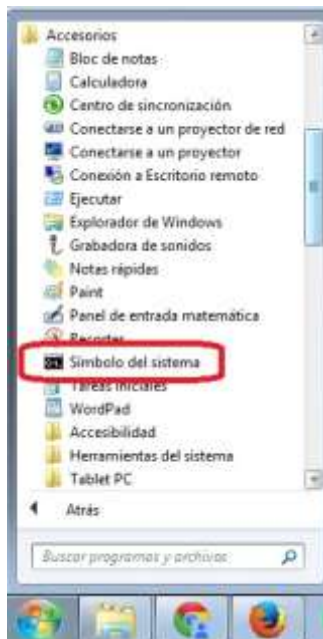
Procedemos a ejecutar el programa que descargamos e instalamos en nuestro equipo:



Dejamos por defecto que instale el Node.js runtime (es el corazón de Node.js)

El **npm** que es otro programa que nos permitirá agregar otros paquetes a nuestro entorno de desarrollo de Node.js

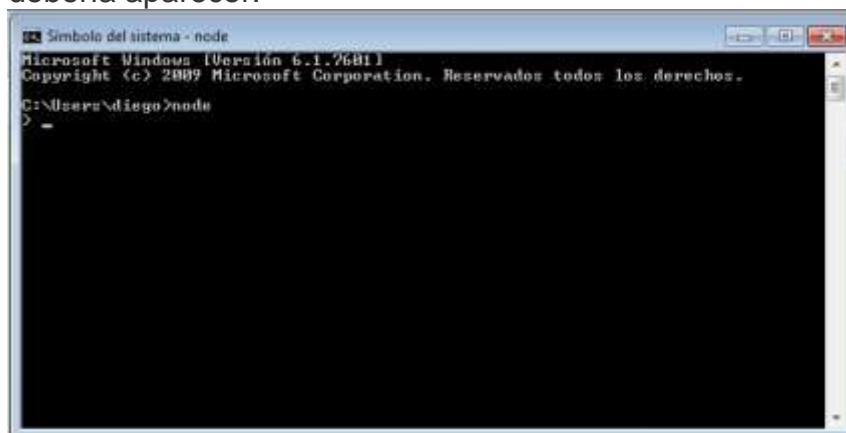
Una vez instalado el Node.js debemos abrir una consola ("Símbolo del sistema") en nuestro sistema operativo para controlar que funciona correctamente, esto lo podemos hacer desde el menú:



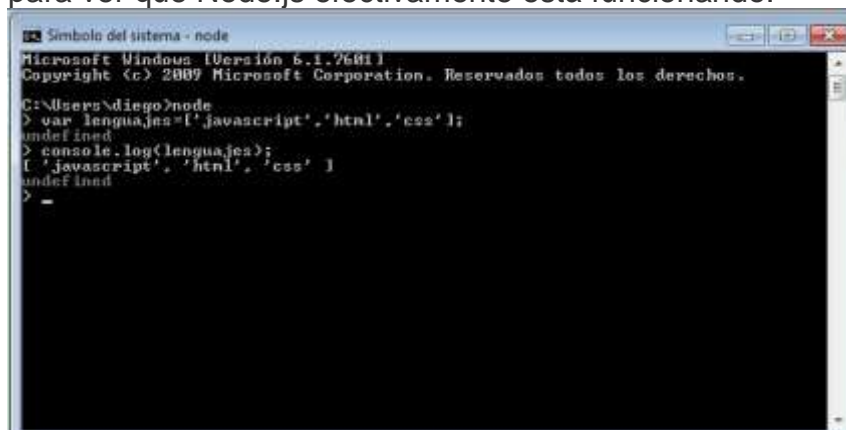
O escribiendo el comando "cmd" donde dice "Buscar programas y archivos":



Si el programa Node.js se instaló correctamente al teclear node en la consola debería aparecer:



Desde esta línea de comandos podemos ejecutar instrucciones en JavaScript para ver que Node.js efectivamente está funcionando:



Como vemos definimos un vector llamado lenguajes y lo inicializamos con 3 string. Luego mediante el objeto console llamamos al método log para que muestre el contenido de la variable de tipo vector llamada lenguajes.

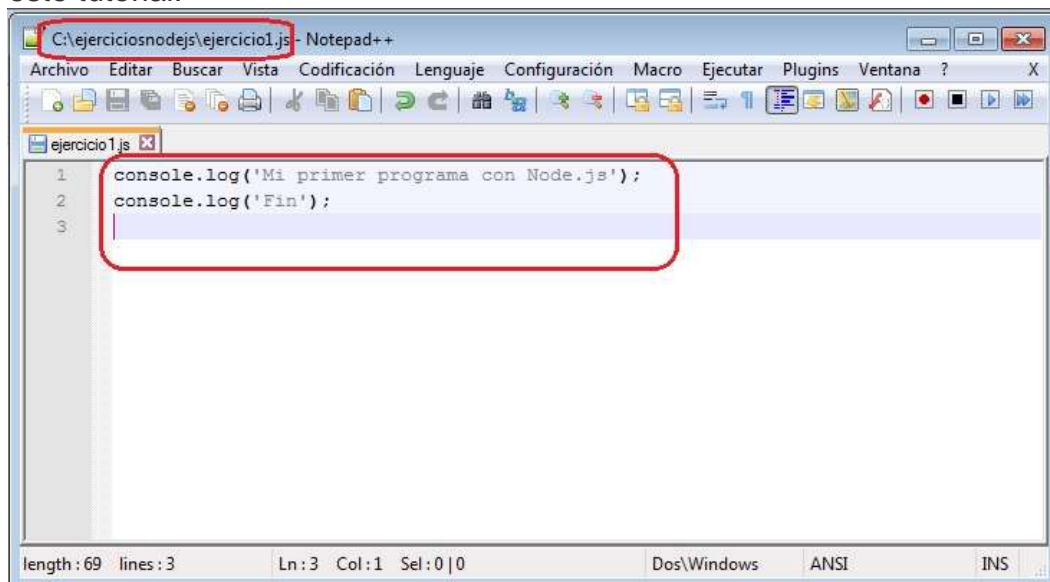
Desde la línea de comandos de node.js podemos ejecutar instrucciones de Javascript pero si lo que necesitamos es implementar un problema más complejo debemos codificar con un editor de texto un programa y almacenarlo con extensión js para luego ejecutarlo llamando al programa node.

Podemos ahora salir de la línea de comandos de node escribiendo el comando .exit:



Ahora nuevamente nos encontramos en la línea de comandos pero del sistema operativo.

Codifiquemos con nuestro editor de texto favorito (por ejemplo [Notepad++](#)) un pequeño programa y lo almacenemos en una carpeta de nuestro sistema operativo donde guardaremos los distintos ejercicios que iremos codificando en este tutorial:

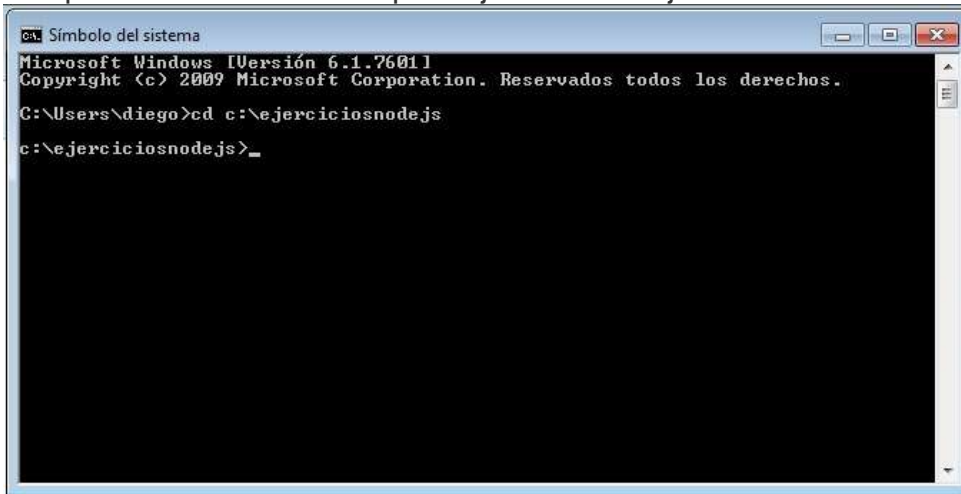


Hemos codificado el programa:

```
console.log('Mi primer programa con Node.js');  
console.log('Fin');
```

Y lo hemos guardado en un archivo llamado ejercicio1.js en la carpeta c:\ejerciciosnodejs\

Veamos ahora como lo ejecutaremos a nuestro programa. Primero abrimos nuevamente una consola ("Símbolo del sistema") en nuestro sistema operativo nos posicionamos en la carpeta ejerciciosnodejs:

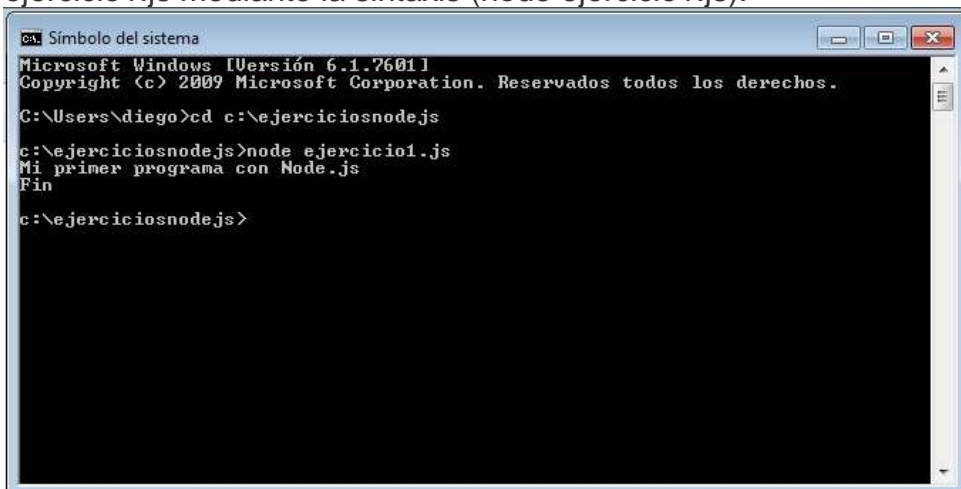


```
ca. Símbolo del sistema
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.
C:\Users\diego>cd c:\ejerciciosnodejs
c:\ejerciciosnodejs>
```

Debemos cambiar y posicionarnos en la carpeta donde almacenamos nuestro programa mediante el comando del sistema operativo Windows "cd" y el path (camino) de la carpeta:

```
cd c:\ejerciciosnodejs
```

Finalmente ejecutamos el programa que almacenamos en el archivo ejercicio1.js mediante la sintaxis (node ejercicio1.js):



```
ca. Símbolo del sistema
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.
C:\Users\diego>cd c:\ejerciciosnodejs
c:\ejerciciosnodejs>node ejercicio1.js
Mi primer programa con Node.js
Fin
c:\ejerciciosnodejs>
```

El resultado del programa aparece en la misma consola (ya que nuestro programa solo muestra dos mensajes utilizando el objeto console)

El resultado de nuestro primer programa es sumamente sencillo pero hemos aprendido a codificar un programa Nodejs, almacenarlo en una carpeta de nuestro sistema y luego ejecutarlo. Este proceso será idéntico a medida que avancemos con este tutorial.

## 2 - Módulos en Node.js

La primera gran diferencia en el JavaScript que se utiliza en Node.js es el concepto de módulo. Sabemos que cuando nuestra aplicación comienza a crecer un único archivo es imposible de manejar todas las funcionalidades, lo mismo ocurre con librerías desarrolladas por otros programadores.

Un módulo contiene funciones, objetos, variables donde indicamos cuales serán exportados para ser utilizados por otros programas.

### Módulo de un único archivo

Vamos a crear un programa muy sencillo que nos permita sumar, restar y dividir números y mostrarlos por la consola. Las funcionalidades las dispondremos en un módulo de archivo y veremos como la consumimos en nuestro programa principal.

Primero creemos nuestro módulo llamado `matematica.js` con el siguiente código:

`matematica.js`

```
var PI=3.14;

function sumar(x1,x2)
{
    return x1+x2;
}

function restar(x1,x2)
{
    return x1-x2;
}

function dividir(x1,x2)
{
    if (x2==0)
    {
        mostrarErrorDivision();
    }
    else
    {
        return x1/x2;
    }
}

function mostrarErrorDivision() {
    console.log('No se puede dividir por cero');
}
```

```
exports.sumar=sumar;  
exports.restar=restar;  
exports.dividir=dividir;  
exports.PI=PI;
```

En este archivo podemos definir variables, funciones, objetos etc. y los que necesitamos que sean accedidos desde otro archivo los exportamos agregándolos al objeto exports:

```
exports.sumar=sumar;  
exports.restar=restar;  
exports.dividir=dividir;  
exports.PI=PI;
```

Aquello que no necesitemos llamarlo desde otra archivo como en este ejemplo pasa con la función mostrarErrorDivision simplemente no la agregamos al objeto exports.

Codifiquemos ahora nuestra aplicación principal que la llamaremos ejercicio2.js y también la guardamos en la misma carpeta donde tenemos el archivo matematica.js:

```
var mat=require('./matematica');  
  
console.log('La suma de 2+2='+mat.sumar(2,2));  
console.log('La resta de 4-1='+mat.restar(4,1));  
console.log('La          división          de  
6/3='+mat.dividir(6,3));  
console.log('El valor de PI='+mat.PI);
```

El JavaScript que se ejecuta en un navegador web actualmente no puede hacer referencia a otro archivo, en Node.js esto se hace llamando a la función require e indicando el path donde se encuentra en este caso el archivo matematica.js (no es necesario indicar la extensión):

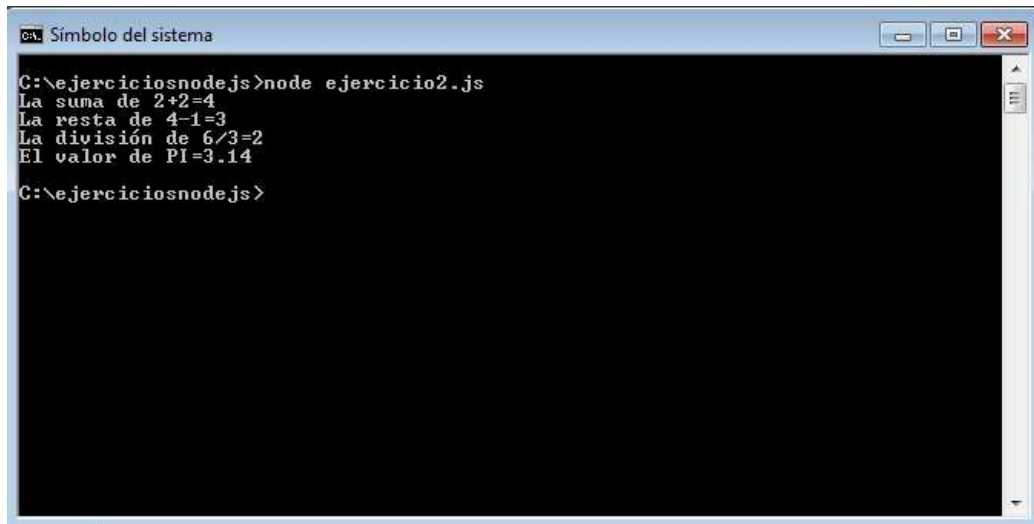
```
var mat=require('./matematica');
```

Luego la variable mat tiene acceso a todas las variables, funciones y objetos exportados.

Llamamos luego a las funciones y accedemos a las variables mediante la variable mat:

```
console.log('La suma de 2+2='+mat.sumar(2,2));  
console.log('La resta de 4-1='+mat.restar(4,1));  
console.log('La división de 6/3='+mat.dividir(6,3));  
console.log('El valor de PI='+mat.PI);
```

En nuestra consola al ejecutar nuestro programa ejercicio2.js tenemos como resultado:

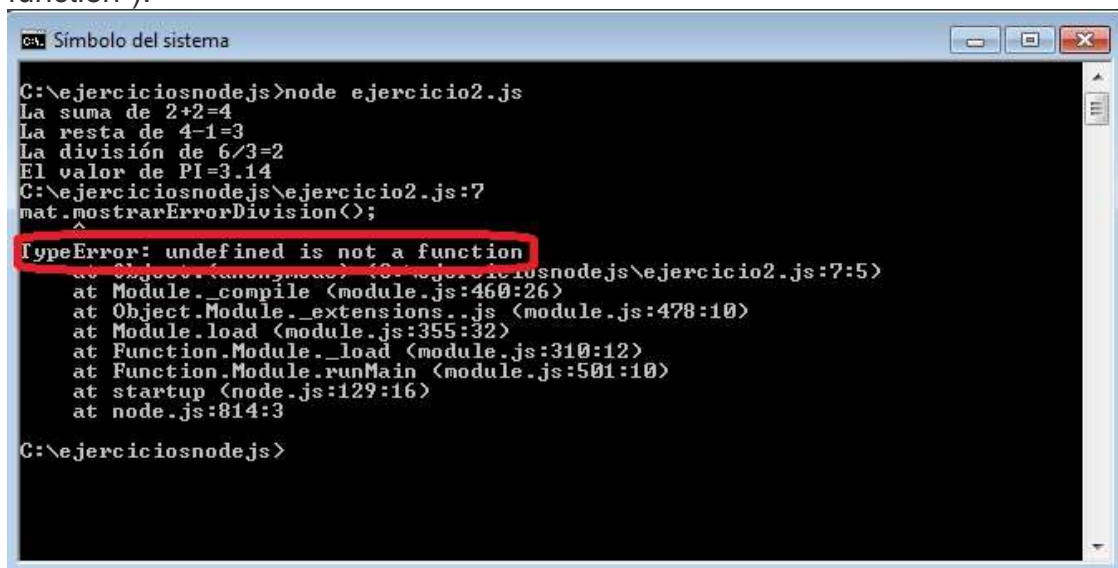


```
C:\ejerciciosnodejs>node ejercicio2.js
La suma de 2+2=4
La resta de 4-1=3
La división de 6/3=2
El valor de PI=3.14
C:\ejerciciosnodejs>
```

Tengamos en cuenta que solo podemos acceder del módulo a aquellos elementos exportados, si por ejemplo tratamos de acceder a la función `mostrarErrorDivision` del módulo `matematica.js`:

```
mat.mostrarErrorDivision();
```

Tendremos como resultado un error en tiempo de ejecución ("undefined is not a function"):



```
C:\ejerciciosnodejs>node ejercicio2.js
La suma de 2+2=4
La resta de 4-1=3
La división de 6/3=2
El valor de PI=3.14
C:\ejerciciosnodejs\ejercicio2.js:7
mat.mostrarErrorDivision();
^
TypeError: undefined is not a function
    at Object.<anonymous> [C:\ejerciciosnodejs\ejercicio2.js:7:5]
    at Module._compile (module.js:460:26)
    at Object.Module._extensions..js (module.js:478:10)
    at Module.load (module.js:355:32)
    at Function.Module._load (module.js:310:12)
    at Function.Module.runMain (module.js:501:10)
    at startup (node.js:129:16)
    at node.js:814:3
C:\ejerciciosnodejs>
```

Esta pequeña introducción de módulos es para entender en los próximos temas cada vez que consumamos módulos que ya vienen con Node.js y de otras librerías que descarguemos de internet.

Veremos más adelante que los módulos pueden ser una carpeta que contiene un conjunto de archivos y de subcarpetas (esto es muy útil si la funcionalidad del módulo es muy compleja y por lo tanto no debería estar en un único archivo)

No nos preocuparemos por ahora de crear módulos propios más complejos ya que la forma de consumir módulos con la función `require` es idéntica ya sea que el módulo sea un único archivo o una carpeta.



### 3 - Consumir módulo del núcleo de Node.js

Hemos visto que para utilizar la funcionalidad de un módulo en Node.js debemos llamar a la función `require` pasando entre comillas el nombre del módulo.

La plataforma Node.js incorpora una serie de módulos de uso muy común en los programas que los incorpora el núcleo de Node.js y que se encuentran codificados en C++ para mayor eficiencia.

Para utilizar los módulos del núcleo también utilizamos la función `require` indicando el nombre del módulo.

Algunos de los módulos del núcleo de Node.js son: `os`, `fs`, `http`, `url`, `net`, `path`, `process`, `dns` etc..

#### Problema

Emplear el módulo "os" que provee información básica del sistema donde se ejecuta la plataforma del Node.js.

`ejercicio3.js`

```
var os=require('os');

console.log('Sistema operativo:'+os.platform());
console.log('Versión del sistema
operativo:'+os.release());
console.log('Memoria total:'+os.totalmem()+
'bytes');
console.log('Memoria libre:'+os.freemem()+
'bytes');
```

Para hacer uso de un módulo del núcleo de Node.js solo hacemos referencia a su nombre:

```
var os=require('os');
```

Recordemos que cuando incorporamos un módulo que codificamos nosotros indicamos el path: `var mat=require('./matematica');`

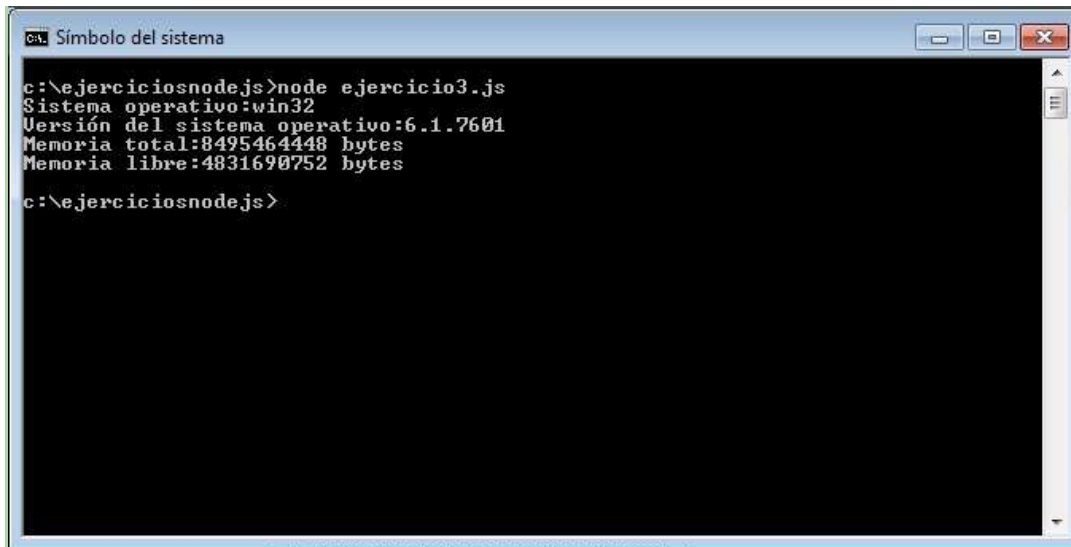
La variable `os` almacena una referencia al módulo 'os' y mediante esta referencia podemos acceder a las variables, funciones, objetos etc. que el módulo exporta (debemos tener la documentación del módulo para poder utilizar su funcionalidad)

Como dijimos el módulo 'os' tiene una serie de métodos que nos informan el ambiente donde se está ejecutando Node.js como puede ser el sistema operativo donde está instalado, la versión del sistema operativo, la cantidad de memoria ram disponible, memoria libre etc.

En el sitio [nodejs.org](https://nodejs.org) podemos consultar la documentación de cada uno de los módulos del núcleo: [consulta del módulo 'os'](#)

El resultado de ejecutar este programa en la consola (recordemos que este script se ejecuta en forma local, más adelante veremos las herramientas que nos provee Node.js para desarrollar programas que se ejecutan en un servidor de internet y los podamos consumir desde un navegador por ejemplo):





```
c:\ejerciciosnodejs>node ejercicio3.js
Sistema operativo:win32
Versión del sistema operativo:6.1.7601
Memoria total:8495464448 bytes
Memoria libre:4831690752 bytes
c:\ejerciciosnodejs>
```

### Problema propuesto

1. Confeccionar un programa que requiera el módulo 'os' para recuperar el espacio libre de memoria. Mostrar inicialmente el espacio libre mediante el método `freemem()`  
Luego crear un vector y mediante el método `push` almacenar 1000000 de enteros. Mostrar luego de la creación y carga del vector la cantidad de espacio libre.  
Llamar al archivo `ejercicio4.js`

```
var os=require('os');

console.log('Memoria libre:'+os.freemem());
var vec=[];
for(var f=0;f<1000000;f++) {
    vec.push(f);
}
console.log('Memoria libre después de crear el
vector:'+os.freemem());
```

## 4 - Módulo para administrar el sistema de archivos: fs

Ya sabemos cómo crear un módulo mínimo, como consumirlo a dicho módulo y también como consumir módulos que vienen por defecto en Node.js.

Ahora veremos un segundo módulo que viene implementado en Node.js por defecto y nos permite acceder al sistema de archivos para poder leer sus contenidos y crear otros archivos o carpetas.

Tenemos que poner mucho cuidado en entender el concepto de programación asincrónica que propone la plataforma de Node.js

La programación asincrónica busca no detener la ejecución del programa en forma completa por actividades que requieren mucho tiempo (una analogía es imaginar que nuestro entorno Node.js es un "mozo de restaurante" que va a una mesa y toma el pedido y lo envía a la cocina, la elaboración del pedido toma su tiempo pero el mozo no se queda congelado hasta que la cocina le avisa que el pedido está preparado sino que sigue tomando pedidos en otras mesas)

El módulo de administración de archivos "fs" implementa la programación asincrónica para procesar su creación, lectura, modificación, borrado etc.

### Creación de un archivo de texto.

Creemos un archivo llamado ejercicio5.js:

```
var fs=require('fs');

fs.writeFile('./archivo1.txt','línea 1\nLínea
2',function(error){
    if (error)
        console.log(error);
    else
        console.log('El archivo fue creado');
});

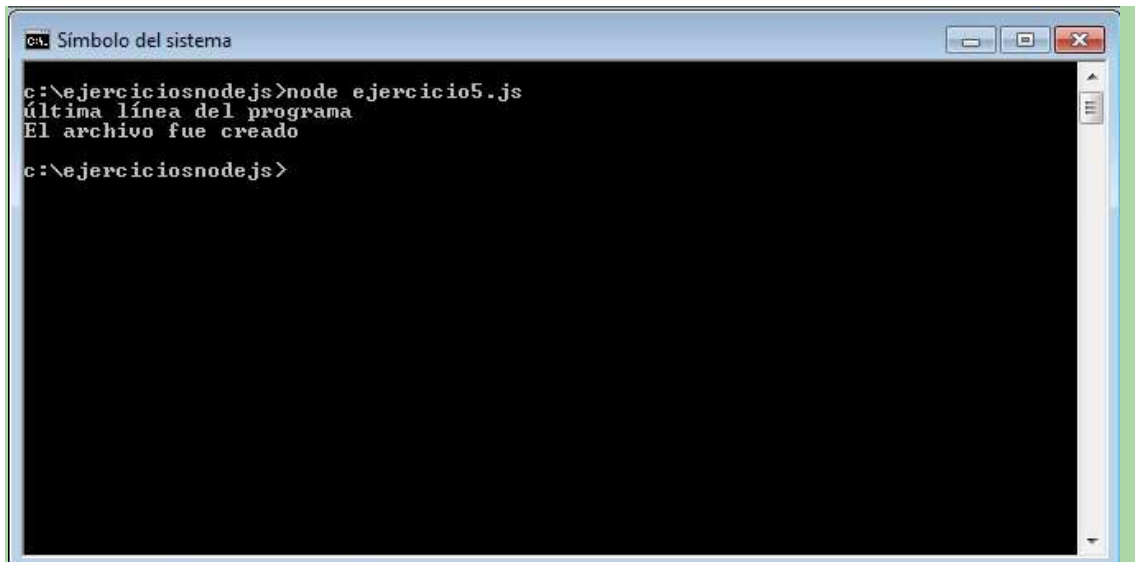
console.log('última línea del programa');
```

Es importante tener en cuenta que cuando ejecutamos este programa aparece en pantalla primero el mensaje:

última línea del programa

antes que:

El archivo fue creado



```
c:\ejerciciosnodejs>node ejercicio5.js
última línea del programa
El archivo fue creado

c:\ejerciciosnodejs>
```

Explicaremos cómo funciona el código de este programa, primero requerimos el módulo 'fs':

```
var fs=require('fs');
```

Llamamos a la función `writeFile` a través de la variable `fs`. Esta función tiene tres parámetros:

- El primer parámetro es el nombre del archivo de texto a crear. Indicamos el path donde debe crearse (en este caso se crea en la misma carpeta donde se encuentra nuestro programa `ejercicio5.js`)
- El segundo parámetro es el string a grabar en el archivo de texto (mediante los caracteres `\n` generamos el salto de línea en el archivo de texto)
- Finalmente el tercer parámetro es una función anónima que será llamada desde el interior de la función `writeFile` cuando haya terminado de crear y grabar el string en el archivo de texto. La función recibe como parámetro un objeto literal con el error en caso de no poderse crear el archivo, en el caso de haber podido crear el archivo retorna en la variable `error` el valor `null`.  
El `if` se verifica verdadero si en la variable `error` viene un objeto sino con el valor `null` almacenado en el parámetro `error` se ejecuta el bloque del `else` donde mostramos el mensaje de que el archivo fue creado.

La programación asíncrona podemos ver que sucede al mostrar el mensaje 'última línea del programa' antes de informarnos que el archivo fue creado. Es decir que cuando llamamos a la función `writeFile` el programa no se detiene en esta línea hasta que el archivo se crea sino que continúa con las siguientes instrucciones.

En este programita en particular no tiene grandes ventajas utilizar la programación asíncrona ya que luego de llamar a la función `writeFile` solo procedemos a mostrar un mensaje por la consola, pero en otras circunstancias podríamos estar ejecutando más actividades que no dependieran de la creación de dicho archivo (por ejemplo ordenando un vector en memoria)

### **Lectura de un archivo de texto.**

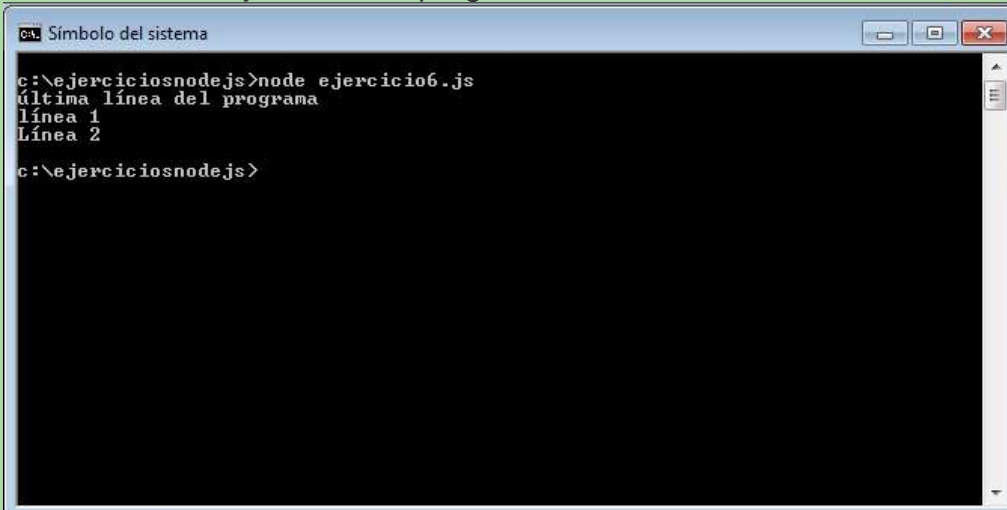
Creemos un archivo llamado ejercicio6.js:

```
var fs=require('fs');

fs.readFile('./archivo1.txt',function(error,dato
s){
    if (error) {
        console.log(error);
    }
    else {
        console.log(datos.toString());
    }
});

console.log('última línea del programa');
```

El resultado de ejecutar este programa es:



```
C:\ejerciciosnodejs>node ejercicio6.js
última línea del programa
línea 1
línea 2
C:\ejerciciosnodejs>
```

Tenemos en pantalla la impresión de las dos líneas del archivo de texto. El módulo 'fs' tiene una función llamada `readFile` que le pasamos como primer parámetro el nombre del archivo a leer y como segundo parámetro una función anónima que se ejecutará cuando se termine de leer el archivo pasando como parámetros un objeto con la referencia del error si lo hubiera y un objeto de tipo Buffer con todos los datos del archivo de texto.

Para mostrar el contenido del Buffer en formato texto llamamos al método `toString()`. Si no hacemos esto en pantalla mostrará los valores numéricos de los caracteres.

Nuevamente estamos implementando la lectura de un archivo en forma asincrónica, con el objeto de no detener el hilo de nuestro programa (esto es muy útil si el archivo a leer es de gran tamaño)

### Otra forma de definir la función que se dispara luego de leer o escribir un archivo.

El empleo de funciones anónimas en JavaScript es muy común pero podemos volver a codificar el problema anterior pasando el nombre de una función:

Modificamos el archivo ejercicio6.js eliminando la función anónima e implementando una función con un nombre explícito:

```
var fs=require('fs');

function leer(error,datos){
    if (error) {
        console.log(error);
    }
    else {
        console.log(datos.toString());
    }
}

fs.readFile('./archivo1.txt',leer);

console.log('última línea del programa');
```

Tengamos en cuenta que el resultado es idéntico a la implementación con la función anónima.

No es obligatorio que la implementación de la función esté definida antes de llamar a readFile, podría estar implementada al final:

```
var fs=require('fs');

fs.readFile('./archivo1.txt',leer);

console.log('última línea del programa');

function leer(error,datos){
    if (error) {
        console.log(error);
    }
    else {
        console.log(datos.toString());
    }
}
```

O inclusive podría estar implementada en otro módulo y requerirla.

El módulo 'fs' tiene muchas funciones más además de crear y leer un archivo como hemos visto como pueden ser borrar, renombrar, crear directorios, borrar directorios, retornar información de archivos etc. para consultar estas funciones podemos visitar [el api de Node.js](#)

En conceptos sucesivos seguiremos utilizando el paquete 'fs' e iremos introduciendo otras funcionalidades.