9.3 MÓDULOS

9.3.1 USO DE MÓDULOS Y PAQUETES

Hay que entender que un módulo es un conjuntode funciones, constantes, objetos, etc, que se usan como una librería que podremos reutilizar para facilitar el desarrollo de aplicaciones.

Los módulos permiten de forma eficaz tanto utilizar módulos de librerías de terceros, como crear módulos propios que pueden reutilizarse en todas nuestras aplicaciones. Esto permite condensar más nuestras aplicaciones y facilitar la reusabilidad del código.

Las aplicaciones de node.js hacen uso de módulos desde hace mucho tiempo. Pero la cuestión es ¿qué pasa con el JavaScript creado para ser ejecutado en un navegador? Es decir ¿qué pasa con el JavaScript del lado del cliente?

Lo cierto es que en el lado del cliente no se usaban módulos, estaban fuera de la norma. Si deseamos usar librerías de funciones y otros elementos, estas se cargaban mediante etiquetas script:

```
<script      src="1ibrerial.js"></script>
<script      src="1ibreria2.js">                  </script>
<script      src="1ibreria3.js"> </script>

<script>
... código que usa funciones de las librerías anteriores....
</script>
```

El código anterior muestra la forma de trabajar cuando queremos reutilizar código JavaScript o utilizar librerías de terceros (como **jQuery,** por ejemplo). La instrucción script, cuando carga archivos externos, genera una petición http por cada librería y eso puede ralentizar la ejecución de la aplicación.

Especialmente preocupante es el hecho de que, de algunas librerías estándar, solo usemos ciertas funcionalidades, ya que la etiqueta script carga el archivo entero. Y esto no es nada eficiente.

En definitiva, con los años, muchos desarrolladores empiezan a echar mucho de menos un sistema de módulos para el desarrollo de aplicaciones en el lado del cliente.

9.3.2 CARGA Y CREACIÓN DE MÓDULOS

La norma ES2015 (también llamada ES6) al fin incorporó el uso de módulos en JavaScript. Con esta norma aparecieron las palabras claves **export** e **import.**

9.3.2.1 CREACIÓN DE MÓDULOS

Un módulo no es más que un conjunto de funciones, variables, objetos y todo tipo de elementos que puedan ser reutilizados en otro código. Cuando deseemos en nuestro código alguno de los elementos del módulo bastará con indicar qué queremos importar el módulo y qué elementos concretos deseamos de él (o bien importar todo).

Por otro lado, cuando se crea el módulo también hay que indicar qué cosas son importables. Supongamos que estamos creando un módulo llamado ***geometria.js*** en el que deseamos colocar funciones de cálculo de áreas y perímetros de figuras. En ese archivo hemos creado la función **areaCirculo** en base a lo siguiente:

```
export function areaCirculo(radio){
     return Math.PI * radio * pradio;
}
```

La palabra **export** hace referencia a que esa función es exportable en otro archivo JavaScript.

Cada función o variable que queremos exportar debe tener por delante la palabra export:

```
export const PI_CUADRADO=Math.PI*Math.PI;

export function areaCirculo(radio){
      return radio * PI_CUADRADO;
}
export function areaCuadrado(lado){
      return lado ** 2;
}
```

También podemos acumular todo lo que queremos exportar en una sola instrucción **export:**

```
const PI_CUADRADO=Math.PI*Math.PI;

function areaCirculo(radio){
      return radio * PI.CUADRADO;
}
function areaCuadrado(lado){
      return lado ** 2;
}
export{
      PI.CUADRADO,
      areaCirculo,
      areaCuadrado
}
```

Aquellos elementos del módulo que no estén en la instrucción **export** se considerarán privados, y, por lo tanto, no exportables.

93.2.2 CARGA DE MÓDULOS

La importación de módulos se realiza con la instrucción **import.** Esta instrucción debe de ser la primera (puede haber varias instrucciones **import)** del código JavaScript. Si queremos cargar un elemento del módulo, podremos hacer lo siguiente:

```
import { areaCirculo } from "./geometría.js";
consolé.log(areaCirculo(5));
```

El código anterior carga la función ***areaCirculo*** del módulo que se ha configurado en el archivo ***geometría.js.*** Si queremos importar varios elementos del módulo, estos se separan con comas:

```
import { areaCirculo, PI.CUADRADO } from /geometría.js";
```

Podemos renombrar los elementos importados:

```
import { areaCirculo as circulo, areaCuadrado as cuadrado } from "./geometría.js";

console.log(circulo(9)); //Escribe 88.82643960980423
console.log(cuadrado(4)); //Escribe 16
```

También podemos importar todos los elementos de un módulo, pero tenemos que asignar un nombre que se utilizará que usar como **espacio de nombre.** Los espacios de nombre son un identificador que se usa como prefijo delante del nombre de cada elemento importado (función, método, variable, etc.) La razón de su uso es diferenciar los nombres de elementos pertenecientes a dos módulos distintos.

```
import * as geom from /geometría.js";
consolé.log(geom.areaCirculo(9));
consolé.log(geom.areaCuadrado(4));
```

Un detalle muy importante es que, en el código HTML, si nuestro código JavaScript utiliza módulos, la etiqueta script que contiene ese código o que carga el código desde un archivo debe utilizar el atributo **type** con el valor **module.** Es decir, el código anterior completo, sería:

```
<script type="module">
import * as geom from "./geometría.js";
consolé.log(geom.areaCirculo(9));
consolé.log(geom.areaCuadrado(4));
</script>
```