Boilerplate

Los ejemplos que vamos a codificar los puedes probar en la sandbox de JavaScript, si no sabes cómo funciona, échale un vistazo al módulo de setup y si te quedan dudas contacta con tu mentor.

Imports

Introducción

En los últimos programas que hemos hecho te habrás dado cuenta de que:

- Empezamos a tener mucho código en un solo fichero.
- Se hace complicado encontrar algo.
- No hay una manera clara de agrupar código y funciones por tipo.

¿Qué puede pasar si nos ponemos a desarrollar una aplicación un poco más compleja? esto se nos iría de madre...

¿Qué sueles hacer cuando por ejemplo te tienes que preparar el temario de una asignatura? Sueles dividirlo en un fichero por tema.

¿Y si tuvieras que impartir varias asignaturas? Puede que incluso crearías una carpeta por cada asignatura.

¿Y si ya das clases en diferentes facultades o planes de estudio? Pues te crearías unas carpetas padre por centro / plan de estudio.

¿Tiene esto sentido? Pues cuando programamos hacemos algo parecido:

- Podemos arrancar a programar en un fichero.
- Conforme esto se complica y se hace más grande, podemos dividir ese código en varios ficheros siguiendo criterios que nos permitan organizarlo mejor.
- Si esto sigue haciéndose más grande, podemos estructurarlo en una jerarquía de carpetas y ficheros.
- Si esto sigue creciendo, o tenemos código que puede servir para varias aplicaciones podemos sacarlo a proyectos comunes (librerías).

También hay veces que podemos tener idea de cómo va a crecer una aplicación, y en ese caso podemos ya partir de una estructura predefinida (Esto no siempre es buena idea, ya que muchas veces conforme se va desarrollando puede resultar que módulos que creíamos iban a tener muy poco peso, se convierten en elefantes).

> Hay una máxima en el mundo de la programación y la arquitectura de software y es que "nunca hay bala de plata", es decir no existe la solución óptima de estructura de ficheros / carpetas que valga para cualquier tipo de proyecto, otra expresión que suele enfadar muchos a los cargos de gestión (jefes de proyecto etc...) es cuando te piden algo que contestes "depende"... pero es que es así 😋.

Hola Import

Punto de partida

Vamos a tomar como punto de partida la aplicación de *adivina un número* que desarrollamos en el módulo anterior (el corte limpio 😋), ¿Qué tenemos aquí?

El código original es este:

HTML

```
<!DOCTYPE html>
<html lang="en">
```

```
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <link rel="stylesheet" href="/src/style.css" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
  </head>
  <body>
    <div id="app">
      <h1>Adivina el numero</h1>
      <div>
        <input type="number" id="numero" />
        <button id="comprobar">Comprobar</button>
      </div>
      <div id="resultado"></div>
      <div id="intentos"></div>
    </div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

JavaScript

```
const generarNumeroAleatorio = () => Math.floor(Math.random() * 101);

const numeroParaAcertar = generarNumeroAleatorio();

const NO_ES_UN_NUMERO = 0;
const EL_NUMERO_ES_MAYOR = 1;
const EL_NUMERO_ES_MENOR = 2;
const ES_EL_NUMERO_SECRETO = 3;
const GAME_OVER_MAXIMO_INTENTOS = 4;

const MAXIMO_INTENTOS = 5;
let numeroDeIntentos = 0;

const hasSuperadoElNumeroMaximoDeIntentos = () =>
  numeroDeIntentos >= MAXIMO_INTENTOS;

const muestraNumeroDeIntentos = () => {
  document.getElementById(
    "intentos"
  ).innerHTML = `${numeroDeIntentos} de ${MAXIMO_INTENTOS}`;
};

document.addEventListener("DOMContentLoaded", muestraNumeroDeIntentos);

const gestionarGameOver = (estado) => {
  if (estado === GAME_OVER_MAXIMO_INTENTOS) {
    document.getElementById("comprobar").disabled = true;
  }
};
```

```
const muestraMensajeComprobacion = (texto, estado) => {
  let mensaje = "";
  switch (estado) {
    case NO_ES_UN_NUMERO:
      mensaje = `"${texto}" no es un numero 😦, prueba otra vez`;
      break;
    case EL_NUMERO_ES_MAYOR:
      mensaje = `UUYYY ! El número ${texto} es MAYOR que el número secreto`;
      break;
    case EL_NUMERO_ES_MENOR:
      mensaje = `UUYYY ! El número ${texto} es MENOR que el número secreto`;
      break;
    case ES_EL_NUMERO_SECRETO:
      mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
      break;
    case GAME_OVER_MAXIMO_INTENTOS:
      mensaje = `🪦 GAME OVER, has superado el número máximo de intentos`;
      break;
    default:
      mensaje = "No se que ha pasado, pero no deberías estar aquí";
      break;
  }

  document.getElementById("resultado").innerHTML = mensaje;
};

const comprobarNumero = (texto) => {
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);

  if (!esUnNumero) {
    return NO_ES_UN_NUMERO;
  }

  if (numero === numeroParaAcertar) {
    return ES_EL_NUMERO_SECRETO;
  }

  if (hasSuperadoElNumeroMaximoDeIntentos()) {
    return GAME_OVER_MAXIMO_INTENTOS;
  }

  return numero > numeroParaAcertar ? EL_NUMERO_ES_MAYOR : EL_NUMERO_ES_MENOR;
};

const handleCompruebaClick = () => {
  const texto = document.getElementById("numero").value;
  const estado = comprobarNumero(texto);
  muestraMensajeComprobacion(texto, estado);
  numeroDeIntentos++;
  muestraNumeroDeIntentos();
  gestionarGameOver(estado);
};
```

```
const botonComprobar = document.getElementById("comprobar");
botonComprobar.addEventListener("click", handleCompruebaClick);
```

Aquí hay demasiado código y cuesta de seguir, nos podemos plantear romper esto en varios ficheros, peeerooo... ¿Qué criterios seguimos?

- Tenemos que buscar una forma de romper el código que haga que sea fácil encontrar lo que buscamnos: por ejemplo saber en que fichero va a estar una función, no vale llevarse cualquier función a un fichero y que después veamos el mismo y no sepamos que puede tener.
- Por otro lado debemos de evitar las dependencias circulares ¿Eso qué es?
    - Imagínate que tengo un ficheroA que depende funciones o datos de un ficheroB.
    - Y que ese ficheroB depende de funciones del ficheroA
    - Esto da problemas de compilación.
- Tampoco merece la pena si acabamos teniendo un montón de ficheros con muy poco contenido, ya que esto puede acabar tan fragmentado que puede ser complicado encontrar tu código.

Es decir, ya tenemos la famosa frase del desarrollador "Depende" (o en inglés "It depends").

Particionando

Cuando partimos en ficheros debemos de seguir una estructura lógica que se justifique, en este caso podemos distinguir en dos grandes bloques:

- Hay funciones que definen las reglas del juego y no tienen nada que ver con interactuar con el HTML (el interfaz de usuario).
- Hay funciones que si se usan para interactuar con la interfaz de usuario (actualizar el HTML, interactuar con eventos como pinchar en un botón, etc...).

¿Y por qué esa distinción? Al principio teníamos todo el código mezclado en modo Spaghetti, e incidimos en la importancia de separar responsabilidades, ahora que lo hemos hecho tenemos la siguiente ventaja:

- El código que define las reglas del juego, está totalmente aislado del interfaz de usuario, y está escrito en lo que se llama *plain vanilla javascript* es decir no está atado a ninguna librería o framework, con lo que ese motor de juego se podría reaprovechar para implementar una web en React, Angular, Vue o incluso React Native...
- Por otro lado el código que define las reglas de juego es muy fácil de probar ya que no tiene interacción con el interfaz de usuario (ya veremos en próximos módulos que son las pruebas unitarias).
- El código de interfaz de usuario lo tenemos encapsulado y especializado en ese área.

¡Anda la leche! Y ahora viene la división natural, ¿Y si dividiéramos el código de la aplicación en dos ficheros?

`motor.js` `ui.js`

En motor.js tendríamos todas las reglas del juego.

En ui.js toda la lógica de presentación (interfaz de usuario).

`ui.js` dependería de `motor.js` ya que tiene que pedir que calcula estados etc... Para después mostrarlo:

motor.js <===== ui.js

¿Oye y no se puede dar que ui.js necesite cosas de motor.js y viceversa que ui.js necesite cosas de motor.js? Eso es muy común que pase, es este caso

```
<======= motor.js ui.js="======">
```

Se llamada *dependencia circular* y es un señor castañazo lo que te da cuando vas a compilar (el compilador va montando un grafo con las relaciones de *imports* que se encuentra y cuando se encuentra una circular es como si se metería en el problema de la pescadilla que se muerde la cola).

¿Cómo se suele resolver esto? Rompiendo la dependencia circular, una de las soluciones es crear un tercer fichero que aglutine las dependencias circulares (así ya no hay, están en un tercer fichero).

```
motor.js <==== ui.js ↓ └——→ c.js < pre>
```

Modelo

Si nos fijamos en el código, tenemos una parte común entre *motor.js* e *ui.js* y son las definiciones de datos y estructuras que nos hace falta para almacenar el estado de la partida, a grosso modo:

```
const numeroParaAcertar = generarNumeroAleatorio();

const NO_ES_UN_NUMERO = 0;
const EL_NUMERO_ES_MAYOR = 1;
const EL_NUMERO_ES_MENOR = 2;
const ES_EL_NUMERO_SECRETO = 3;
const GAME_OVER_MAXIMO_INTENTOS = 4;

const MAXIMO_INTENTOS = 5;
let numeroDeIntentos = 0;
```

A esto lo podemos llamar *modelo*, así pues tendríamos la siguiente propuesta de estructura de ficheros:

```
motor.js <==== ui.js ↓ modelo.js < pre>
```

Vamos a arrancarnos con este refactor inicial.

Lo primero empezamos con modelo:

./src/modelo.js

```
const numeroParaAcertar = generarNumeroAleatorio();

const NO_ES_UN_NUMERO = 0;
const EL_NUMERO_ES_MAYOR = 1;
const EL_NUMERO_ES_MENOR = 2;
const ES_EL_NUMERO_SECRETO = 3;
const GAME_OVER_MAXIMO_INTENTOS = 4;

const MAXIMO_INTENTOS = 5;
let numeroDeIntentos = 0;
```

Al estar en JavaScript, salvo que tengamos algún plugin instalado no nos avisa de ningún error... pero si tenemos uno serio: *numeroParaAcertar* está invocando a *generarNumeroAleatorio* ¿Qué pasa aquí?

- Si nos traemos *generaNumeroAleatorio* estamos trayendo parte del motor de reglas del juego a nuestro modelo... mala cosa.
- Si hacemos un import de main (o del motor cuando lo tengamos separado), íbamos a tener una temida *dependencia circular* (motor tira de modelo y modelo tira de motor).

¿Cómo solucionamos esto?

- Vamos a dejar la definición de variable de *numeroParaAcertar* inicializada a un valor por defecto (cero, o menos uno lo que mejor te venga).
- Vamos a pasar la inicialización de partida (número aleatorio al motor), así pues hacemos el siguiente cambio en el modelo:

./src/modelo.js

```
- const numeroParaAcertar = generarNumeroAleatorio();
+ const numeroParaAcertar = 0; // Valor por defecto en el motor se inicializa al
arrancar partida

const NO_ES_UN_NUMERO = 0;
const EL_NUMERO_ES_MAYOR = 1;
const EL_NUMERO_ES_MENOR = 2;
const ES_EL_NUMERO_SECRETO = 3;
const GAME_OVER_MAXIMO_INTENTOS = 4;

const MAXIMO_INTENTOS = 5;
let numeroDeIntentos = 0;
```

Por otro lado todo lo que hemos definido en este fichero es *privado*, es decir otro fichero no puede consumir esta información, ¿Cómo puedo hacer para que sí se puedan consumir estas variables desde otros ficheros? Añadiendo *export* a cada variable o función que queramos exportar:

./src/modelo.js

```
- const numeroParaAcertar = 0; // Valor por defecto en el motor se inicializa al
arrancar partida
+ // Cambiamos a un let porque esta variable se asignara en el motor o main js
cuando se empiece nueva partida
+ export let numeroParaAcertar = 0; // Valor por defecto en el motor se inicializa
al arrancar partida

- const NO_ES_UN_NUMERO = 0;
+ export const NO_ES_UN_NUMERO = 0;

- const EL_NUMERO_ES_MAYOR = 1;
+ export const EL_NUMERO_ES_MAYOR = 1;

- const EL_NUMERO_ES_MENOR = 2;
+ export const EL_NUMERO_ES_MENOR = 2;

- const ES_EL_NUMERO_SECRETO = 3;
+ export const ES_EL_NUMERO_SECRETO = 3;

- const GAME_OVER_MAXIMO_INTENTOS = 4;
+ export const GAME_OVER_MAXIMO_INTENTOS = 4;

- const MAXIMO_INTENTOS = 5;
+ export const MAXIMO_INTENTOS = 5;

-let numeroDeIntentos = 0;
+ export let numeroDeIntentos = 0;
```

Ahora toca consumir este fichero de modelo desde main JS, podemos hacerlo de dos formas:

Una que importándolo con nombre y poniendo *modelo* y lo que sea después:

No copiar este código

```
import modelo from "./modelo.js";

modelo.numeroParaAcertar = generarNumeroAleatorio();
```

Y otra forma es aplicando *destructuring* ¿Esto qué es? Resulta que con el import nos traemos un objeto con todo lo que se exporta, al ser un objeto podemos descomponerlo, antes de importar vamos a cómo funciona destructuring:

NO COPIES Y PEGUES ESTE CODIGO EN EL EJEMPLO

Imagínate que tengo algo así:

```
const cliente = {
  nombre: "Pepe",
```

```
  apellidos: "Perez",
};
```

Yo puedes acceder a nombre y apellidos de esta manera *cliente.nombre* o *client.apellidos* pero también puedo hacer lo siguiente:

NO COPIES Y PEGUES ESTE CODIGO EN EL EJEMPLO

```
const cliente = {
  nombre: 'Pepe',
  apellidos: 'Perez'
}

+ const {nombre} = cliente;
+ console.log(nombre);
```

¿Qué estoy haciendo aquí?

- Parto de que cliente es un objeto que tiene la propiedad *nombre*
- Le estoy diciendo que me desarme el objeto, y me cree una variable nombre que tenga exactamente el valor que tiene cliente.nombre.
- Así ya puedes usar *nombre* directamente en vez de tener que repetir en todos sitios *cliente.nombre*

¿Qué solución es mejor? .... te va a entrar la risa, la respuesta es "Depende" (It Depends) 😊, en nuestro caso vamos a tirar por destructuring para no tener que ir añadiendo el prefijo *modelo.* en todos sitios en el fichero *main.js*, más sabiendo que cuando pasemos a *TypeScript* esto se va a simplificar bastante, ahora sí... vamos a picar código en firme:

./src/main.js

```
+ import {numeroParaAcertar,
+  NO_ES_UN_NUMERO,
+  EL_NUMERO_ES_MAYOR,
+  EL_NUMERO_ES_MENOR,
+  ES_EL_NUMERO_SECRETO,
+  GAME_OVER_MAXIMO_INTENTOS,
+  MAXIMO_INTENTOS,
+  numeroDeIntentos
+ } from './modelo'; // Esto se simplificará bastante con la solución en
TypeScript

const generarNumeroAleatorio = () => Math.floor(Math.random() * 101);

- const numeroParaAcertar = generarNumeroAleatorio();
+ // TODO: Esto deberíamos de plantearlo mejor
+ numeroParaAcertar = generarNumeroAleatorio();


- const NO_ES_UN_NUMERO = 0;
```

```
- const EL_NUMERO_ES_MAYOR = 1;
- const EL_NUMERO_ES_MENOR = 2;
- const ES_EL_NUMERO_SECRETO = 3;
- const GAME_OVER_MAXIMO_INTENTOS = 4;

- const MAXIMO_INTENTOS = 5;
- let numeroDeIntentos = 0;

const hasSuperadoElNumeroMaximoDeIntentos = () =>
  numeroDeIntentos >= MAXIMO_INTENTOS;
```

> Aquí hemos puesto un comentario que empiezar por *TODO:* Esto viene del inglés *TO DO* (por hacer), se suelen añadir estos comentarios cuando hay algo que se puede mejorar y debería de implementarse, esto es un estándar que se sigue y hay herramientas que buscan y te marcan esto cuando vas a subir tu código para que lo limpies antes (o bien implementes lo que haga falta o borres el TODO)

Bueno... parece que está todo listo ¿Intentamos compilar esto y ver cómo tira?

```
npm run dev
```

💣💣 CAAAASTAAAÑAAAZO 💣💣

El error que nos da:

```
✗ [ERROR] Cannot assign to import "numeroParaAcertar"

    src/main.js:15:0:
      15 │ numeroParaAcertar = generarNumeroAleatorio();
         ╵ ~~~~~~~~~~~~~~~~~

  Imports are immutable in JavaScript. To modify the value of this import, you
must export a
  setter function in the imported file (e.g. "setNumeroParaAcertar") and then
import and
  call that function here instead.
```

¿Qué quiere decir esto? Qué tenemos varias variables con *let* (*numeroParaAcertar* y *numeroDeIntentos*) y no las puede modificar directamente fuera del fichero *modelo*, de ahí que diga que un *import* es inmutable.

¿Qué posibles soluciones tenemos?

A) La que nos aconseja el compilador es crear una función para asignar el valor a la variable desde el módulo(es lo que se llama un *setter*):

./src/modelo.js

```
export let numeroParaAcertar = 0; // Valor por defecto en el motor se inicializa
al arrancar partida
+ export const setNumeroParaAcertar = (nuevoNumero) => numeroParaAcertar =
nuevoNumero;

export const NO_ES_UN_NUMERO = 0;
export const EL_NUMERO_ES_MAYOR = 1;
export const EL_NUMERO_ES_MENOR = 2;
export const ES_EL_NUMERO_SECRETO = 3;
export const GAME_OVER_MAXIMO_INTENTOS = 4;

export const MAXIMO_INTENTOS = 5;
export let numeroDeIntentos = 0;
+ export const setNumeroDeIntentos = (nuevoNumeroDeIntentos) => numeroDeIntentos =
nuevoNumeroDeIntentos;
```

B) Otra opción es encapsular *numeroParaAcertar* y *numeroDeIntentos* en un objeto, así podemos modificarlo sin problemas (el objeto completo es inmutable, pero no las propiedades que tiene dentro), ESTA OPCION NO LA VAMOS A IMPLEMENTAR EN ESTE CASO (Si la implementaremos cuando estamos con la solución en *TypeScript* ¿Por qué? Porque es más fácil realizar el refactor teniendo a nuestro viejo amigo *TS* avisándonos de todo lo que se nos ha pasado).

ESTA OPCIÓN NO LA COPIES Y PEGUES EN EL CODIGO

```
-export let numeroParaAcertar = 0; // Valor por defecto en el motor se inicializa
al arrancar partida
- export const setNumeroParaAcertar = (nuevoNumero) => numeroParaAcertar =
nuevoNumero;

export const NO_ES_UN_NUMERO = 0;
export const EL_NUMERO_ES_MAYOR = 1;
export const EL_NUMERO_ES_MENOR = 2;
export const ES_EL_NUMERO_SECRETO = 3;
export const GAME_OVER_MAXIMO_INTENTOS = 4;

export const MAXIMO_INTENTOS = 5;
- export let numeroDeIntentos = 0;
- export const setNumeroDeIntentos = (nuevoNumeroDeIntentos) => numeroDeIntentos =
nuevoNumeroDeIntentos;

+ export const partida = {
+  numeroParaAcertar: 0,
+  numeroDeIntentos: 0,
+ }
```

Como hemos comentado elegimos la opción A) y vamos ahora actualizar el código de *main.js*

Primero los imports

./src/main.js

```
import {
  numeroParaAcertar,
+ setNumeroParaAcertar,
  NO_ES_UN_NUMERO,
  EL_NUMERO_ES_MAYOR,
  EL_NUMERO_ES_MENOR,
  ES_EL_NUMERO_SECRETO,
  GAME_OVER_MAXIMO_INTENTOS,
  MAXIMO_INTENTOS,
  numeroDeIntentos,
+ setNumeroDeIntentos
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript
```

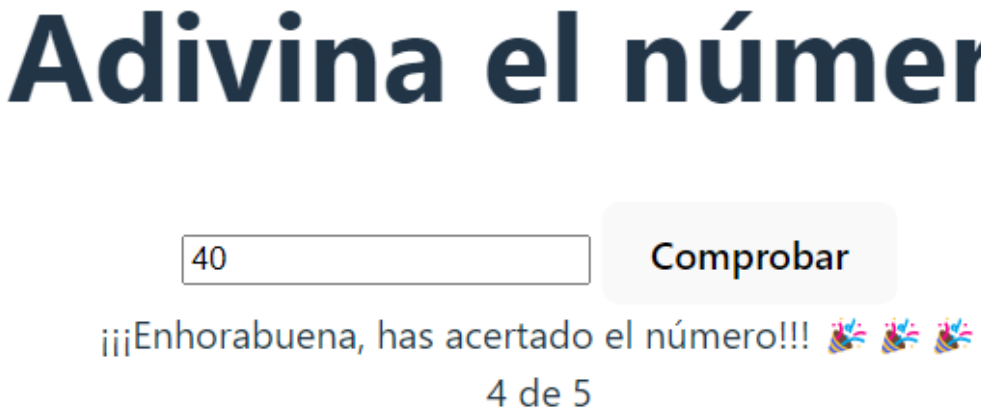
Ahora el código donde intento hacer el set de las variables que traíamos con *let*

```
// TODO: Esto deberíamos de plantearlo mejor
- numeroParaAcertar = generarNumeroAleatorio();
+ setNumeroParaAcertar(generarNumeroAleatorio());
```

```
const handleCompruebaClick = () => {
  const texto = document.getElementById("numero").value;
  const estado = comprobarNumero(texto);
  muestraMensajeComprobacion(texto, estado);
-  numeroDeIntentos++;
+  setNumeroDeIntentos(numeroDeIntentos + 1); // TODO: Aquí se podría añadir una
función _incrementaNumeroDeIntentos_
  muestraNumeroDeIntentos();
  gestionarGameOver(estado);
};
```

Ya lo tenemos andando...

```
npm run dev
```



Vamos a seguir con el refactor.

Motor

Vamos a llevarnos ahora todo lo que son las reglas del juego a un fichero aparte, como hemos comentado así separamos responsabilidades, el código es más fácil de probar (ya veremos lo que son las pruebas unitarias), e incluso podemos aprovechar este motor en otras tecnologías.

./motor.js

```
import {
  numeroParaAcertar,
  setNumeroParaAcertar,
  NO_ES_UN_NUMERO,
  EL_NUMERO_ES_MAYOR,
  EL_NUMERO_ES_MENOR,
  ES_EL_NUMERO_SECRETO,
  GAME_OVER_MAXIMO_INTENTOS,
  MAXIMO_INTENTOS,
  numeroDeIntentos,
  setNumeroDeIntentos,
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript

export const generarNumeroAleatorio = () => Math.floor(Math.random() * 101);

export const hasSuperadoElNumeroMaximoDeIntentos = () =>
  numeroDeIntentos >= MAXIMO_INTENTOS;

export const comprobarNumero = (texto) => {
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);

  if (!esUnNumero) {
    return NO_ES_UN_NUMERO;
  }
```

```
  if (numero === numeroParaAcertar) {
    return ES_EL_NUMERO_SECRETO;
  }

  if (hasSuperadoElNumeroMaximoDeIntentos()) {
    return GAME_OVER_MAXIMO_INTENTOS;
  }

  return numero > numeroParaAcertar ? EL_NUMERO_ES_MAYOR : EL_NUMERO_ES_MENOR;
};
```

Y hacemos limpia en *main.js*

```
import {
  numeroParaAcertar,
  setNumeroParaAcertar,
  NO_ES_UN_NUMERO,
  EL_NUMERO_ES_MAYOR,
  EL_NUMERO_ES_MENOR,
  ES_EL_NUMERO_SECRETO,
  GAME_OVER_MAXIMO_INTENTOS,
  MAXIMO_INTENTOS,
  numeroDeIntentos,
  setNumeroDeIntentos,
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript

+ import {
+  generarNumeroAleatorio,
+  hasSuperadoElNumeroMaximoDeIntentos,
+  comprobarNumero
+ } from './motor';

- const generarNumeroAleatorio = () => Math.floor(Math.random() * 101);

// TODO: Esto deberíamos de plantearlo mejor
setNumeroParaAcertar(generarNumeroAleatorio());

- const hasSuperadoElNumeroMaximoDeIntentos = () =>
-  numeroDeIntentos >= MAXIMO_INTENTOS;

 const muestraNumeroDeIntentos = () => {
  document.getElementById(
    "intentos"
  ).innerHTML = `${numeroDeIntentos} de ${MAXIMO_INTENTOS}`;
};

document.addEventListener("DOMContentLoaded", muestraNumeroDeIntentos);

const gestionarGameOver = (estado) => {
  if (estado === GAME_OVER_MAXIMO_INTENTOS) {
    document.getElementById("comprobar").disabled = true;
  }
```

```
};

const muestraMensajeComprobacion = (texto, estado) => {
  let mensaje = "";
  switch (estado) {
    case NO_ES_UN_NUMERO:
      mensaje = `"${texto}" no es un numero 😲, prueba otra vez`;
      break;
    case EL_NUMERO_ES_MAYOR:
      mensaje = `UUYYY ! El número ${texto} es MAYOR que el número secreto`;
      break;
    case EL_NUMERO_ES_MENOR:
      mensaje = `UUYYY ! El número ${texto} es MENOR que el número secreto`;
      break;
    case ES_EL_NUMERO_SECRETO:
      mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
      break;
    case GAME_OVER_MAXIMO_INTENTOS:
      mensaje = `🪦 GAME OVER, has superado el número máximo de intentos`;
      break;
    default:
      mensaje = "No se que ha pasado, pero no deberías estar aquí";
      break;
  }

  document.getElementById("resultado").innerHTML = mensaje;
};

- const comprobarNumero = (texto) => {
-  const numero = parseInt(texto);
-  const esUnNumero = !isNaN(numero);
-
-  if (!esUnNumero) {
-    return NO_ES_UN_NUMERO;
-  }
-
-  if (numero === numeroParaAcertar) {
-    return ES_EL_NUMERO_SECRETO;
-  }
-
-  if (hasSuperadoElNumeroMaximoDeIntentos()) {
-    return GAME_OVER_MAXIMO_INTENTOS;
-  }
-
-  return numero > numeroParaAcertar ? EL_NUMERO_ES_MAYOR : EL_NUMERO_ES_MENOR;
- };

const handleCompruebaClick = () => {
  const texto = document.getElementById("numero").value;
  const estado = comprobarNumero(texto);
  muestraMensajeComprobacion(texto, estado);
  setNumeroDeIntentos(numeroDeIntentos + 1); // TODO: Aquí se podría añadir una
funcion _incrementaNumeroDeIntentos_
  muestraNumeroDeIntentos();
```

```
  gestionarGameOver(estado);
};

const botonComprobar = document.getElementById("comprobar");
botonComprobar.addEventListener("click", handleCompruebaClick);
```

Al hacer el refactor, se nos marca en gris el import de *numeroParaAcertar* (no se usa ahora en este fichero, lo podemos eliminar):

./src/main.js

```
import {
-  numeroParaAcertar,
  setNumeroParaAcertar,
  NO_ES_UN_NUMERO,
  EL_NUMERO_ES_MAYOR,
  EL_NUMERO_ES_MENOR,
  ES_EL_NUMERO_SECRETO,
  GAME_OVER_MAXIMO_INTENTOS,
  MAXIMO_INTENTOS,
  numeroDeIntentos,
  setNumeroDeIntentos,
} from "./modelo";
```

UI

Ya tenemos el motor de reglas, vamos a separar ahora la parte de interfaz de usuario:

Creamos el fichero *ui.js*

```
import {
  setNumeroParaAcertar,
  NO_ES_UN_NUMERO,
  EL_NUMERO_ES_MAYOR,
  EL_NUMERO_ES_MENOR,
  ES_EL_NUMERO_SECRETO,
  GAME_OVER_MAXIMO_INTENTOS,
  MAXIMO_INTENTOS,
  numeroDeIntentos,
  setNumeroDeIntentos,
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript

export const muestraNumeroDeIntentos = () => {
  document.getElementById(
    "intentos"
  ).innerHTML = `${numeroDeIntentos} de ${MAXIMO_INTENTOS}`;
};

export const gestionarGameOver = (estado) => {
```

```
  if (estado === GAME_OVER_MAXIMO_INTENTOS) {
    document.getElementById("comprobar").disabled = true;
  }
};

export const muestraMensajeComprobacion = (texto, estado) => {
  let mensaje = "";
  switch (estado) {
    case NO_ES_UN_NUMERO:
      mensaje = `"${texto}" no es un numero 😐, prueba otra vez`;
      break;
    case EL_NUMERO_ES_MAYOR:
      mensaje = `UUYYY ! El número ${texto} es MAYOR que el número secreto`;
      break;
    case EL_NUMERO_ES_MENOR:
      mensaje = `UUYYY ! El número ${texto} es MENOR que el número secreto`;
      break;
    case ES_EL_NUMERO_SECRETO:
      mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
      break;
    case GAME_OVER_MAXIMO_INTENTOS:
      mensaje = `🪦 GAME OVER, has superado el número máximo de intentos`;
      break;
    default:
      mensaje = "No se que ha pasado, pero no deberías estar aquí";
      break;
  }

  document.getElementById("resultado").innerHTML = mensaje;
};
```

Y vaciamos el *main.js*

```
import {
  setNumeroParaAcertar,
  NO_ES_UN_NUMERO,
  EL_NUMERO_ES_MAYOR,
  EL_NUMERO_ES_MENOR,
  ES_EL_NUMERO_SECRETO,
  GAME_OVER_MAXIMO_INTENTOS,
  MAXIMO_INTENTOS,
  numeroDeIntentos,
  setNumeroDeIntentos,
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript

import {
  generarNumeroAleatorio,
  comprobarNumero,
} from "./motor";

+ import {
+  muestraNumeroDeIntentos,
```

```
+  gestionarGameOver,
+  muestraMensajeComprobacion
+ } from './ui';

// TODO: Esto deberíamos de plantearlo mejor
setNumeroParaAcertar(generarNumeroAleatorio());

- const muestraNumeroDeIntentos = () => {
-  document.getElementById(
-    "intentos"
-  ).innerHTML = `${numeroDeIntentos} de ${MAXIMO_INTENTOS}`;
- };

document.addEventListener("DOMContentLoaded", muestraNumeroDeIntentos);

- const gestionarGameOver = (estado) => {
-  if (estado === GAME_OVER_MAXIMO_INTENTOS) {
-    document.getElementById("comprobar").disabled = true;
-  }
- };

- const muestraMensajeComprobacion = (texto, estado) => {
-  let mensaje = "";
-  switch (estado) {
-    case NO_ES_UN_NUMERO:
-      mensaje = `"${texto}" no es un numero 😮, prueba otra vez`;
-      break;
-    case EL_NUMERO_ES_MAYOR:
-      mensaje = `UUYYY ! El número ${texto} es MAYOR que el número secreto`;
-      break;
-    case EL_NUMERO_ES_MENOR:
-      mensaje = `UUYYY ! El número ${texto} es MENOR que el número secreto`;
-      break;
-    case ES_EL_NUMERO_SECRETO:
-      mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
-      break;
-    case GAME_OVER_MAXIMO_INTENTOS:
-      mensaje = `🪦 GAME OVER, has superado el número máximo de intentos`;
-      break;
-    default:
-      mensaje = "No se que ha pasado, pero no deberías estar aquí";
-      break;
-  }
-
-  document.getElementById("resultado").innerHTML = mensaje;
- };

const handleCompruebaClick = () => {
  const texto = document.getElementById("numero").value;
  const estado = comprobarNumero(texto);
  muestraMensajeComprobacion(texto, estado);
  setNumeroDeIntentos(numeroDeIntentos + 1); // TODO: Aquí se podría añadir una
funcion _incrementaNumeroDeIntentos_
  muestraNumeroDeIntentos();
```

```
  gestionarGameOver(estado);
};

const botonComprobar = document.getElementById("comprobar");
botonComprobar.addEventListener("click", handleCompruebaClick);
```

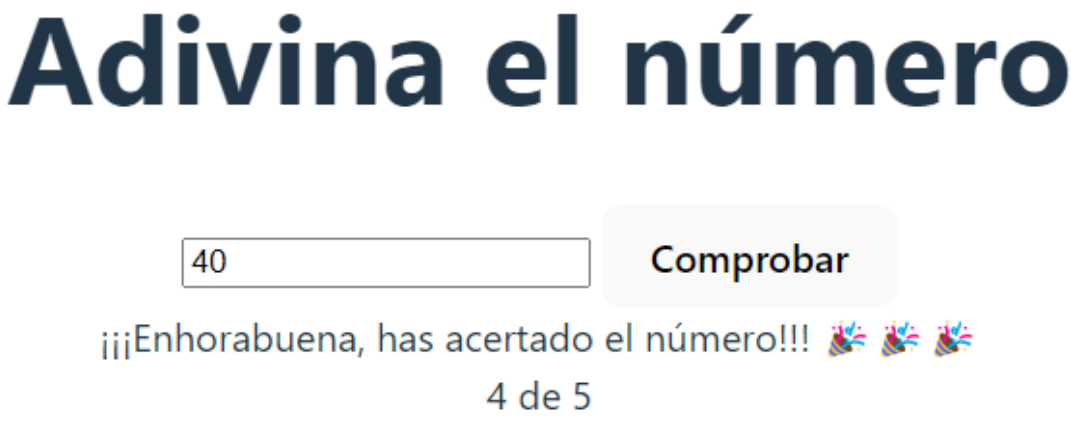
Fíjate que al hacer el refactor, se quedan un montón de imports muertos de *main.js* que podemos eliminar.

./main.js

```
import {
  setNumeroParaAcertar,
-  NO_ES_UN_NUMERO,
-  EL_NUMERO_ES_MAYOR,
-  EL_NUMERO_ES_MENOR,
-  ES_EL_NUMERO_SECRETO,
-  GAME_OVER_MAXIMO_INTENTOS,
-  MAXIMO_INTENTOS,
  numeroDeIntentos,
  setNumeroDeIntentos,
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript
```

Vamos a probar:

```
npm run dev
```



¿A qué ya echabas de menos picar tú? ... pues TE TOCA:

handleCompruebaClick lo puedes pasar al fichero de *ui.js* pásalo y actualiza los *imports*.

Inténtalo y no leas la solución que viene ahora 😃

./ui.js

```
+ export const handleCompruebaClick = () => {
+  const texto = document.getElementById("numero").value;
+  const estado = comprobarNumero(texto);
+  muestraMensajeComprobacion(texto, estado);
+  setNumeroDeIntentos(numeroDeIntentos + 1); // TODO: Aquí se podría añadir una
funcion _incrementaNumeroDeIntentos_
+  muestraNumeroDeIntentos();
+  gestionarGameOver(estado);
+ };

export const muestraNumeroDeIntentos = () => {
  document.getElementById(
    "intentos"
  ).innerHTML = `${numeroDeIntentos} de ${MAXIMO_INTENTOS}`;
};
```

./main.js

```
import {
  muestraNumeroDeIntentos,
  gestionarGameOver,
  muestraMensajeComprobacion,
+ handleCompruebaClick
} from "./ui";

// TODO: Esto deberíamos de plantearlo mejor
setNumeroParaAcertar(generarNumeroAleatorio());

document.addEventListener("DOMContentLoaded", muestraNumeroDeIntentos);

- const handleCompruebaClick = () => {
-  const texto = document.getElementById("numero").value;
-  const estado = comprobarNumero(texto);
-  muestraMensajeComprobacion(texto, estado);
-  setNumeroDeIntentos(numeroDeIntentos + 1); // TODO: Aquí se podría añadir una
funcion _incrementaNumeroDeIntentos_
-  muestraNumeroDeIntentos();
-  gestionarGameOver(estado);
-};
```

Y eliminamos imports muertos de *main.js*

./src/main.js

```
import {
  setNumeroParaAcertar,
-  numeroDeIntentos,
-  setNumeroDeIntentos,
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript
```

otro TE TOCA, *setNumeroParaAcertar* debería de llamarse en el *DOMContentLoaded* es más tiene sentido crear una función *inicializaNuevaPartida* que llamara tanto a *setNumeroParaAcertar* como a *setNumeroDeIntentos* (poniéndolo a cero), como a *muestraNumeroDeIntentos* ya que esto nos podría valer para tener un botón de nueva partida a futuro.

Intenta hacerlo y no leas la solución que viene ahora 😊

./src/main.js

```
import {
  setNumeroParaAcertar,
  numeroDeIntentos,
+  setNumeroDeIntentos,
} from "./modelo"; // Esto se simplificará bastante con la solución en TypeScript

import { generarNumeroAleatorio, comprobarNumero } from "./motor";

import {
  muestraNumeroDeIntentos,
-  gestionarGameOver,
-  muestraMensajeComprobacion,
  handleCompruebaClick,
} from "./ui";

- // TODO: Esto deberíamos de plantearlo mejor
- setNumeroParaAcertar(generarNumeroAleatorio());

+ const inicializaNuevaPartida = () => {
+   setNumeroParaAcertar(generarNumeroAleatorio());
+   setNumeroDeIntentos(0);
+   muestraNumeroDeIntentos();
+ }

- document.addEventListener("DOMContentLoaded", muestraNumeroDeIntentos);
+ document.addEventListener("DOMContentLoaded", inicializaNuevaPartida);

const botonComprobar = document.getElementById("comprobar");
botonComprobar.addEventListener("click", handleCompruebaClick);
```

Si quieres rizar el rizo, podrías partir *inicializaNuevaPartida* en dos *inicializaNuevaPartida* e *inicializaNuevaPartidaUI* y soltarlas en motor e ui.

Shell

Si te fijas lo que se nos ha quedado en la aplicación es el código mínimo para arrancarla, lo que se suele conocer como el *shell* (si eres argentino te vas a reír mucho con la traducción al español de esto).

Vamos a hacer un refactor más que es innecesario, esto pasa al final vamos refactorizando y hacemos una vuelta de tuerca de más, y tenemos que volver atrás es lo que los pescadores dicen *voy a recoger carrete*, pero nos va a servir de excusa para aprender un *import* más, sin parámetros.

Vamos a crear un archivo que llamaremos *shell.js* y vamos a mover el código que nos queda en *main.js* a este nuevo fichero.

./src/shell.js

```
import { setNumeroParaAcertar, setNumeroDeIntentos } from "./modelo"; // Esto se
simplificará bastante con la solución en TypeScript

import { generarNumeroAleatorio, comprobarNumero } from "./motor";

import { muestraNumeroDeIntentos, handleCompruebaClick } from "./ui";

const inicializaNuevaPartida = () => {
  setNumeroParaAcertar(generarNumeroAleatorio());
  setNumeroDeIntentos(0);
  muestraNumeroDeIntentos();
};

document.addEventListener("DOMContentLoaded", inicializaNuevaPartida);

const botonComprobar = document.getElementById("comprobar");
botonComprobar.addEventListener("click", handleCompruebaClick);
```

Y ahora vamos a importarlo en *main.js*

```
+ import "./shell";

- import {
- setNumeroParaAcertar,
- setNumeroDeIntentos,
- } from "./modelo"; // Esto se simplificará bastante con la solución en
TypeScript
-
- import { generarNumeroAleatorio, comprobarNumero } from "./motor";
- -import { muestraNumeroDeIntentos, handleCompruebaClick } from "./ui";
- -const inicializaNuevaPartida = () => {
- setNumeroParaAcertar(generarNumeroAleatorio());
- setNumeroDeIntentos(0);
- muestraNumeroDeIntentos();
- };
-
```

```
- document.addEventListener("DOMContentLoaded", inicializaNuevaPartida);
-
- const botonComprobar = document.getElementById("comprobar");
- botonComprobar.addEventListener("click", handleCompruebaClick);
```

¿Qué hace ese *import*? Importa el fichero y ejecuta el código, en este caso:

- Crea el *DOMContentLoaded* listener
- Crea el *click* listener

Resumen

Qué hemos visto en este módulo:

- Por qué es importante modularizar.
- Que guía seguir para modularizar.
- Como separar en ficheros y como importarlos.
- Como importar todo el contenido o utilizar destructuring.
- Como detectar imports muertos.
- Como importar un fichero sin parámetros.