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.

Condicionales

Conceptos

En la vida real es muy normal tomar decisiones lógicas:

- ¿Voy en tren o en coche?
- ¿Me llevo el paraguas?
- ¿Me quedo en casa o salgo a cenar?
- ...

En programación también es muy común tomar decisiones lógicas, por ejemplo:

- ¿Son correctos los datos y puedo grabar la información?
- ¿El usuario tiene permisos para acceder a la información?
- ¿El usuario ha acertado a una pregunta en un juego?

Para tomar decisiones lógicas en JavaScript utilizamos los condicionales.

¿Cómo funciona esto?

- Planteamos una condición.
- Si esa condición se cumple se ejecuta un bloque de código.
- Opcionalmente, si no se cumple la condición se ejecuta otro bloque de código.

Es decir algo así como:

```
if (condición) {
  // código que se ejecuta si la condición se cumple
} else {
  // Esta parte es opcional
  // código que se ejecuta si la condición no se cumple
}
```

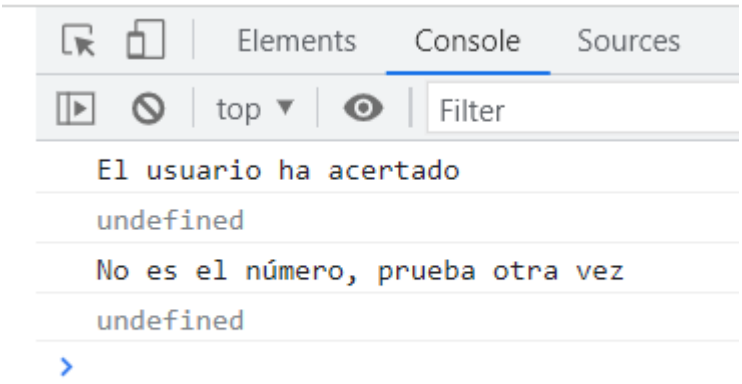
Para ver si se cumple una condición utilizamos los operadores de comparación: `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`.

Si tenemos más de una condición (por ejemplo que el usuario además de acertar el número tiene que hacerlo en menos de 3 intentos), utilizamos operadores lógicos: `&&` (y), `||` (o), `!` (no).

Imagínate que implementamos un juego en el que el ordenador elije un número al azar entre 1 y 100 y el usuario tiene que adivinarlo ¿Cómo podemos comprobar si el usuario ha acertado?

```
const haAcertadoElNumero = (numeroAAcertar, numeroDelUsuario) => {
  if (numeroAAcertar === numeroDelUsuario) {
    console.log("El usuario ha acertado");
  } else {
    console.log("No es el número, prueba otra vez");
  }
};

console.log(haAcertadoElNumero(50, 50)); // Mensaje de consola: El usuario ha
acertado
```

```
console.log(haAcertadoElNumero(50, 51)); // Mensaje de consola: No es el número, prueba otra vez
```



> Lo hemos encapsulado en una función para que sea más fácil de leer. Este código se puede escribir de una manera más corta, pero ya lo veremos más adelante.

¿Qué tenemos aquí?

- Primero la condición: `numeroAAcertar === numeroDelUsuario` aquí estamos comparando si el número a acertar es igual al número del que ha tecleado el usuario *¿Por qué usamos* `===`*?* En JavaScript `=` está reservado para asignar valores, y `==` compara dos valores pero no tiene en cuenta el tipo de datos, por ejemplo "*1*" `==` *1* daría true, es mejor usar `===` que compara el valor y el tipo de dato, por ejemplo "*1*" `===` *1* daría false.
- Segundo, el bloque de código que se ejecuta si la condición se cumple: `console.log("El usuario ha acertado");`, aquí lo hemos separado entre llaves, si no ponemos llaves y tomaría como condición del *if* la siguiente línea, pero es mala práctica hacer esto, es mejor ser explícito y siempre envolver entre llaves.
- Después tenemos un *else* y otro bloque de código, esto sirve para definir un caso alternativo *¿Oye y si la condición no se cumple? ¿Qué pasa?* Pues que se ejecuta este bloque de código, esto no siempre hace falta implementarlo.

Una forma equivalente de escribir esta función sería:

```
const haAcertadoElNumero = (numeroAAcertar, numeroDelUsuario) => {
-  if(numeroAAcertar === numeroDelUsuario) {
+  if(numeroAAcertar !== numeroDelUsuario) {
-    console.log("El usuario ha acertado");
+    console.log("No es el número, prueba otra vez");
  } else {
-    console.log("No es el número, prueba otra vez");
+    console.log("El usuario ha acertado");

  }
}
```

Aquí estamos usando la negación, es decir estamos preguntado si NO es igual, por eso hemos tenido que cambiar el orden de los *console.log*.

Cómo hemos comentado antes tenemos más operadores (mayor que, menor que, menor o igual que...).

Te Toca

Implementa una función en la que pasamos un número de intentos, y el máximo de intentos que se permita, y nos diga por consola si el usuario ha superado el número de intentos o no.

Pistas:

- Crear una función *haSuperadoElNumeroDeIntentos* que reciba dos parámetros: *numeroDeIntentos* y *maximoDeIntentos*.
- Dentro añade un *if* y comprueba si *numeroDeIntentos* es mayor que *maximoDeIntentos*, para ello puedes utilizar el operador >.
- En el bloque en que es mayor que *maximoDeIntentos* imprime por consola "*Has superado el número de intentos*".
- En el bloque en que no es mayor que *maximoDeIntentos* imprime por consola "*Aún no has superado el número de intentos*".
- Invoca a la función con diferentes parámetros y comprueba por la consola que ha tenido éxito.

Adivina el número ejemplo completo

Una de las dudas que suele surgir cuando nos ponemos a implementar condicionales es... ¿Y dónde lo pongo en el código? Vamos a implementar un ejemplo completo de un juego de adivinar el número.

¿Qué flujo de aplicación vamos a implementar?

- Cuando se carga la página se calcula un número aleatorio entre 1 y 100.
- Una vez que el juego ha empezado se muestra un input para que el usuario introduzca un número y un botón para ver si el usuario ha acertado.
- Cuando el usuario pulse el botón se comprueba si el número introducido es igual al número aleatorio, si es así se muestra un mensaje de felicitación y se oculta el input y el botón y vuelta a empezar.

Número aleatorio

¿Cómo generamos un número entre 0 y 100? En JavaScript tenemos una función llamada *Math.random* este función cuando lo llamamos nos devuelve un número aleatorio entre 0 y 1 (ojo no incluye el 1), es decir un número con decimales (0.2, ó 0.99999).

¿Qué podemos hacer con esto?

- Si lo multiplicamos por 100 nos dará un número entre 0 y 99.9999
- Si a este paso le hacemos un redondeo con *floor* (esto quita los decimales), tendría un número entre 0 y 99.
- Si queremos estar entre 0 y 100 tendríamos que multiplicar por 101 y hacer el redondeo.

> Otra alternativa sería usar *round* en vez de *floor*, esto te puede dar para un buen ejercicio de investigación.

Vamos a generar el número entre 0 y 100:

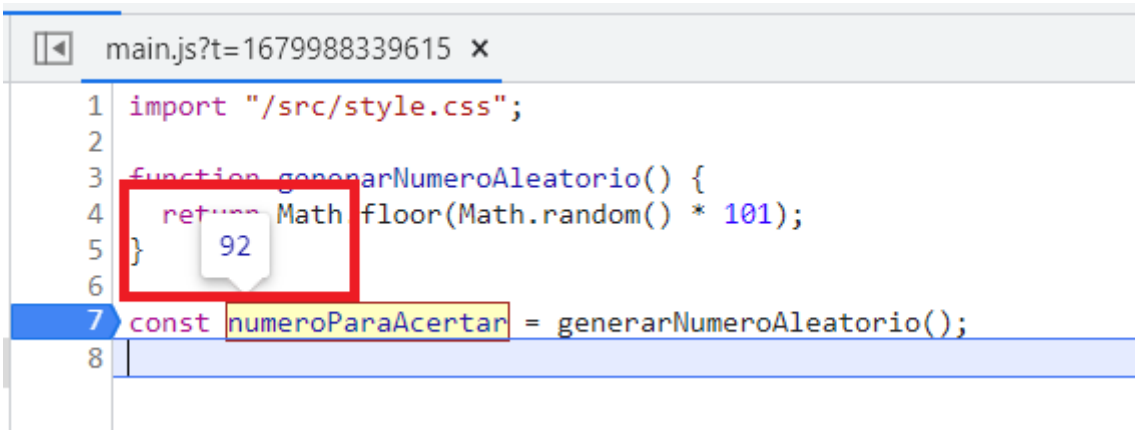
```
function generarNumeroAleatorio() {
  return Math.floor(Math.random() * 101);
}
```

> Y si quieres rizar el rizo, prueba a crear una función que te de un número aleatorio pasándole un mínimo y un máximo por parámetros en la función.

Vamos a crear una variable para tener el número secreto, y vamos a llamar a la función para que nos genere un número aleatorio y lo guardemos en la variable.

```
const numeroParaAcertar = generarNumeroAleatorio();
```

> Abriendo la consola del navegador y accediendo al tab de source para depurar, podemos ver el número que se ha generado 😆, interesante forma de hacer trampas 😜.



Layout del juego

Vamos ahora a generar el HTML del juego:

```
<body>
  <div id="app">
    <h1>Adivina el número</h1>
    <div>
      <input type="number" id="numero" />
      <button id="comprobar">Comprobar</button>
    </div>
    <div id="resultado"></div>
  </div>
  <script type="module" src="/src/main.js"></script>
</body>
```

Y para que se vea más bonito, vamos a enlazar el CSS de estilos:

```
<!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>
```

Interacción con el usuario

Esta parte te puede servir de repaso del módulo anterior.

Ahora vamos a:

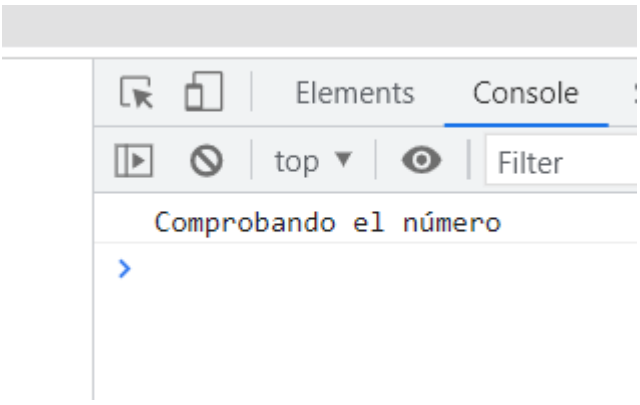
- Crear una función para comprobar el resultado (de momento la dejaremos vacía, sólo pondremos un console.log para ver que se está invocando).
- Enlazarla con el botón de comprobar.

Si te animas... TE TOOOCAAAA !!!

Que haríamos, añadimos al final de nuestro fichero js lo siguiente:

```
function comprobarNumero() {
  console.log("Comprobando el número");
}

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

Ahora vamos a comprobar que funciona, para ello abrimos el fichero HTML en el navegador y pulsamos el botón de comprobar, si todo ha ido bien veremos por consola el mensaje "*Comprobando el número*".



> Esta forma de programar: primero hacer un paso pequeño, comprobar que funciona, y después ir a por otro, es muy útil cuando programamos, yo la llamo la técnica del escalador, subimos un trecho de pared, ponemos una chapa, aseguramos cuerda y seguimos subiendo, así si nos caemos sólo es una

> distancia controlada (ya sabes que el resto del código si funcionaba y el código que puede fallar está más acotado)

Comprobando el número

Vamos a comprobar el número, para ello:

- Convertimos el valor del input a un número (si da NaN, sabemos que el usuario no ha introducido un número).
- Comprobamos si el número es igual al número secreto.
- En caso de que no sea igual mostramos un mensaje indicando que no es el número secreto.
- En caso de que sea igual mostramos un mensaje indicando que ha acertado.

Empezamos por la parte más sencilla, convertir el valor del input a un número:

```
function comprobarNumero() {
-  console.log("Comprobando el número");
+  const texto = document.getElementById("numero").value;
+  const numero = parseInt(texto);
}
```

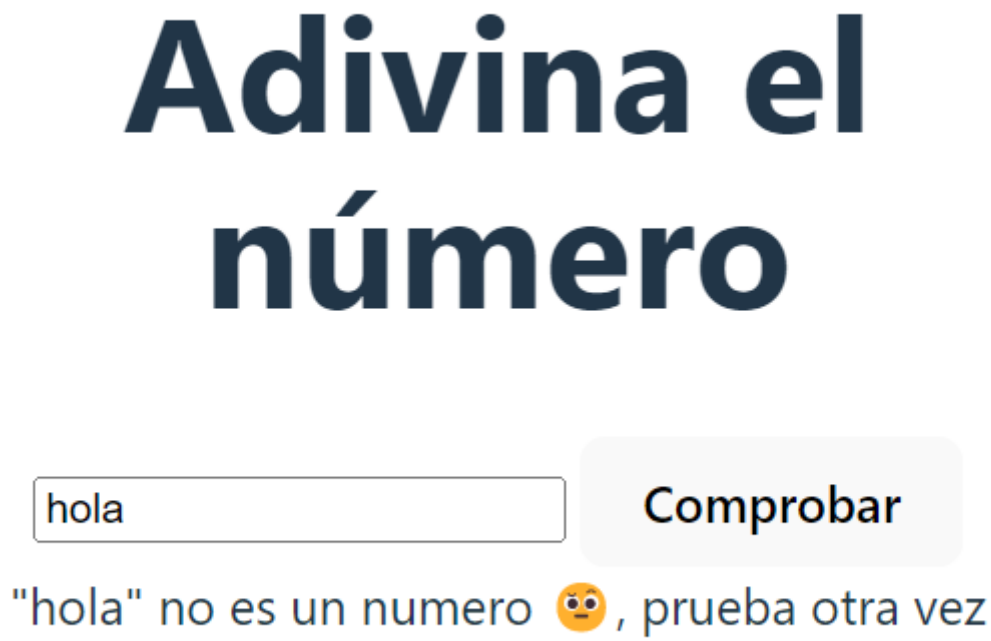
¿Cómo comprobamos si el valor del input es un número y no has introducido una cadena de texto (por ejemplo "hola")? Utilizando la función *isNaN* que nos devuelve true si el valor que le pasamos no es un número.

```
function comprobarNumero() {
  const texto = document.getElementById("numero").value;
  const numero = parseInt(texto);
+ const esUnNumero = !isNaN(numero);
}
```

Y ahora viene el primer condicional... si lo que hemos introducido no es un número, mostramos un mensaje de error:

```
function comprobarNumero() {
  const texto = document.getElementById("numero").value;
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);

+ if (!esUnNumero) {
+   document.getElementById("resultado").innerHTML = `"${texto}" no es un número
😟, prueba otra vez`;
+ }
}
```

Vamos a probarlo, arranca la aplicación y prueba a introducir "hola" donde pide el número (vas a tener que eliminar temporalmente el *type=number* del input en el HTML), te aparecerá el mensaje de error.



Ahora toca la siguiente condición:

- Si el número es igual al número secreto, mostramos un mensaje de felicitación.
- Si el número no es igual al número secreto, mostramos un mensaje indicando que no es el número secreto.

Una duda que te puede salir es ¿Dónde ponemos esta condición? Párate a pensar... ¿Cuándo puedes evaluar si el número es el secreto? Pues una vez que sabes que al menos el usuario ha introducido un número y no una cadena de texto 😃, vamos a hacer uso de la palabra reservada *else* que nos permite ejecutar un bloque de código si la condición del *if* no se cumple.

```
function comprobarNumero() {
  const texto = document.getElementById("numero").value;
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);

 if (!esUnNumero) {
   document.getElementById("resultado").innerHTML = `"${texto}" no es un número
😉, prueba otra vez`;
 }
+ else {
+  // Si es un número, comprobamos si es el número secreto
+ }
}
```

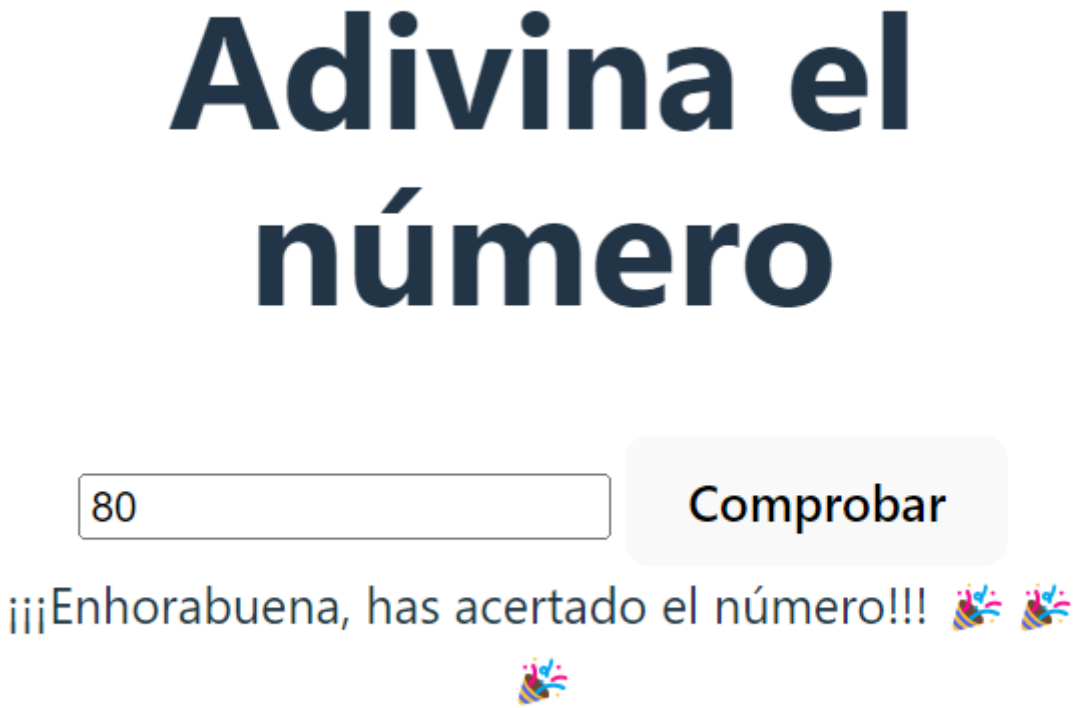
Y ¿Qué haríamos aquí? Pues con otro if comprobamos si el número es igual al número secreto, y mostramos un mensaje de felicitación o de error.

¿Te Toca? ¿Te animas a implementarlo tú y después comprobamos la solución?

Vamos a hacer lo que se llama un *if anidado* es decir un *if dentro de un if*:

```
 if (!esUnNumero) {
   document.getElementById("resultado").innerHTML = `"${texto}" no es un numero
😟, prueba otra vez`;
 } else {
  // Si es un número, comprobamos si es el número secreto
+  if (numero === numeroParaAcertar) {
+    document.getElementById("resultado").innerHTML = `¡¡¡Enhorabuena, has
acertado el número!!! 🎉🎉🎉`;
+  } else {
+    document.getElementById("resultado").innerHTML = `Lo siento ${texto}, el
número no es el correcto 😦, prueba de nuevo`;
+  }
 }
```

Vamos a probar este código, y para no desesperarnos viendo si funciona o no, vamos a depurar el código para sacar el número secreto 😉, inténtalo 😄.



Ahora mismo tenemos un código que funciona y que más o menos se entiende, pero es un poco porquería, veamos porqué:

- De primeras tenemos dos if anidados, esto así tal cual no está del todo mal, pero imagínate que empezamos a meter más complejidad, por ejemplo darle una pista al usuario de si el número está por debajo o por encima del que ha dicho... igual tendríamos que meter un *if* más y podemos acabar con un *Hadouken* (formalmente se llama complejidad ciclomática), y es que muchos *if* / *for* anidados hacen que el código sea difícil de entender y de mantener.

```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

- Por otro lado estamos mezclando: tenemos un código que sirve para ver si el nombre es válido, y otro para mostrarlo por pantalla, esto es un problema, porque si en un futuro queremos cambiar la forma de mostrar el mensaje tenemos un sphaguetti mezclado de comprobación y mostrar mensaje.

¿Vaaaleee pero he picado este código mal, porque no tengo experiencia verdad? cuando ya tenga más práctica lo haré todo bien de primeras ¿No? La respuesta es NO, la mayoría de las veces vas a empezar a codificar algo, y luego vas a ir refactorizando, es como si estuvieras esculpiendo una piedra, es un proceso iterativo

Refactorizando el código

Lo dicho, antes de que esto se vaya de madre es bueno refactorizar el código, para ello vamos a crear una función que se encargue de mostrar el mensaje por pantalla, y que reciba como parámetro el mensaje que queremos mostrar.

Vamos a enumerar los tres casos que podemos tener:

- 0 - El usuario no ha introducido un número.

- 1 - El usuario ha introducido un número, pero no es el número secreto.
- 2 - El usuario ha introducido el número secreto.

Para ello definimos esto como constantes (así evitamos poner números mágicos sueltos por el código), en la primera línea del fichero JS ponemos lo siguiente:

```
const NO_ES_UN_NUMERO = 0;
const NO_ES_EL_NUMERO_SECRETO = 1;
const ES_EL_NUMERO_SECRETO = 2;
```

Vamos a hacer una función que reciba como parámetro el texto introducido, y el estado de la comprobación, y que se encargue de mostrar el mensaje por pantalla.

```
const muestraMensajeComprobacion = (texto, estado) => {
  let mensaje = "";
  if (estado === NO_ES_UN_NUMERO) {
    mensaje = `"${texto}" no es un numero 😐, prueba otra vez`;
  } else {
    if (estado === NO_ES_EL_NUMERO_SECRETO) {
      mensaje = `Lo siento ${texto}, el número no es el correcto 😢, prueba de
nuevo`;
    } else {
      if (estado === ES_EL_NUMERO_SECRETO) {
        mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
      }
    }
  }
  document.getElementById("resultado").innerHTML = mensaje;
};
```

Esta pieza de código huele a *Hadouken*, volveremos a ella para mejorarla, pero por ahora vamos a ver como se usa.

Vámonos ahora a crear una función que se encargue de comprobar si el número es el secreto, y que devuelva el estado de la comprobación.

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

  if (esUnNumero) {
    // Si es un número, comprobamos si es el número secreto
    if (numero === numeroParaAcertar) {
      resultado = ES_EL_NUMERO_SECRETO;
    } else {
      resultado = NO_ES_EL_NUMERO_SECRETO;
    }
```

```
  }
  return resultado;
};
```

> ¿Cómo puedes saber si ese código funciona como es debido? Más adelante nos ayudaremos de las pruebas unitarias para poder probar bloques de código como este.

Y ahora vamos a crear una función que será la que se enganche con el evento de click del botón, y que invoque a las dos funciones anteriores.

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

Vamos a enganchar esta función con el evento de click del botón.

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

- Vamos a repasar este código y ver porque es buena idea escribirlo así:
  - De primeras, si no tuviéramos que desarrollar más igual podríamos haber dejado el código como estaba (era relativamente corto, y se entendía), pero... si vamos a tener que desarrollar sobre (suele pasar que se tiene que extender, por ejemplo dar pistas si el número es mayor o menor) o si a futuro quisiéramos mejorar el interfaz de usuario y cómo mostramos el error, es mejor reorganizarlo.
  - Empezamos por el *handleCompruebaClick* leyendo esta función de una pasada sabemos lo que hace:
    - Lee el valor del input.
    - Comprueba si el número es el secreto.
    - Muestra el resultado por pantalla.
  - Si queremos comprobar cómo funciona cada una de la funciones, podemos hacerlo de forma independiente y sólo tenemos que hacer un *go to definition*.
  - Hemos separado lo que es interfaz de usuario, de lo que podemos llamar *lógica de negocio* (la comprobación de los números), esto de separar es importante, basado en un caso real, me encontré en un proyecto en el que tenían mezclado una validación de NIF con interfaz de usuario, y estaba repetida en 15 sitios esta comprobación para adaptarla a cada UI,... cuando lo ideal hubiera sido sacar la validación de NIF a una función e invocarla desde esos 15 sitios.

> Al final del módulo implementaremos el ejemplo completo sin aplicar código limpio para que veáis la diferencia.

Switch

Vamos a seguir limpiando el código, nos habíamos quedado con una sensación rara con este código:

```
const muestraMensajeComprobacion = (texto, estado) => {
  let mensaje = "";
  if (estado === NO_ES_UN_NUMERO) {
    mensaje = `"${texto}" no es un numero 😐, prueba otra vez`;
  } else {
    if (estado === NO_ES_EL_NUMERO_SECRETO) {
      mensaje = `Lo siento ${texto}, el número no es el correcto 😦, prueba de
nuevo`;
    } else {
      if (estado === ES_EL_NUMERO_SECRETO) {
        mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
      }
    }
  }
  document.getElementById("resultado").innerHTML = mensaje;
};
```

En este caso podíamos discutir si todavía es legible o no, pero a poco que añadiéramos más casos se podría volver un poco lioso, y a fin a de cuentas, tenemos un lista de condiciones, en la que si no cumple una se cumplirá la otra, y todas dependen del valor de una variable que va tomando valores discretos (es decir que no hay decimales ni nada) ¿No hay una forma más fácil se escribir esto? Te presento a tu amigo el *switch*

```
const muestraMensajeComprobacion = (texto, estado) => {
  let mensaje = "";
-  if (estado === NO_ES_UN_NUMERO) {
-    mensaje = `"${texto}" no es un número 😐, prueba otra vez`;
-  } else {
-    if (estado === NO_ES_EL_NUMERO_SECRETO) {
-      mensaje = `Lo siento ${texto}, el número no es el correcto 😦, prueba de
nuevo`;
-    } else {
-      if (estado === ES_EL_NUMERO_SECRETO) {
-        mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
-      }
-    }
-  }
+  switch(estado){
+    case NO_ES_UN_NUMERO:
+      mensaje = `"${texto}" no es un número 😐, prueba otra vez`;
+      break;
+    case NO_ES_EL_NUMERO_SECRETO:
+      mensaje = `Lo siento ${texto}, el número no es el correcto 😦, prueba de
nuevo`;
+      break;
+    case ES_EL_NUMERO_SECRETO:
+      mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
+      break;
+   default:
```

```
+      mensaje = "No se que ha pasado, pero no deberías estar aquí";
+      break;
+   }
   document.getElementById("resultado").innerHTML = mensaje;
};
```

¿Qué estamos haciendo aquí?

- Indicamos con *switch* que queremos evaluar valores de la variable estado.
- Por cada valor de *estado* que queramos evaluar, ponemos un *case* y ejecutamos el código que queramos.
- Para indicar que termina el caso y se salga del *case* le indicamos con *break* que salga.
- Tenemos un caso *cajón de sastre* (*default*) en el que si no viene una de los valores esperados, podemos gestionar esta situación no esperada (en módulos posteriores veremos cómo lanzar excepciones cuando esto suceda)

Ternarios

Vamos ahora a revisar *comprobarNumero* y ver si podemos mejorarla.

Fíjate en estas líneas de código:

```
if (numero === numeroParaAcertar) {
  resultado = ES_EL_NUMERO_SECRETO;
} else {
  resultado = NO_ES_EL_NUMERO_SECRETO;
}
```

Aquí tenemos un patrón:

- Tenemos una condición
- En ambos casos de la condición lo que se acaba haciendo es asignar una valor a la variable resultado.
- Tenemos una línea de código para cada caso.

¿Hay una forma más simple de escribir esto? Te presento a los ternarios 😃:

```
- if (numero === numeroParaAcertar) {
-  resultado =  ES_EL_NUMERO_SECRETO;
- } else {
-  resultado =  NO_ES_EL_NUMERO_SECRETO;
- }
+ resultado = (numero === numeroParaAcertar) ? ES_EL_NUMERO_SECRETO :
NO_ES_EL_NUMERO_SECRETO;
```

¿Qué estamos haciendo aquí?

- Primero decimos que vamos a asignar a *resultado* un valor.

- Después ponemos la condición que queremos y evaluar, y para separar la condición de las acciones ponemos un interrogante.
- La primera parte de código será la que se ejecute si se da la condición (es decir asignará a la variable resultado el valor ES_EL_NUMERO_SECRETO).
- Para separar al código a ejecutar si no se da la condición ponemos ":".
- Si no se da la condición se ejecuta la segunda parte (es decir asignará a la variable resultado el valor NO_ES_EL_NUMERO_SECRETO).

A este *if abreviado* se le llama operador ternario ¿Por qué ese nombre tan raro?

- Es porque tiene 3 operandos (ternarios).
- La primera parte de la expresión, (condición), es una expresión booleana que se evalúa como verdadera o falsa.
- La segunda el código a ejecutar si se da la condición.
- La tercera el código a ejecutar si NO se da la condición.

Oye... pues yo estoy más cómodo con un IF No tienes por qué usar ternarios, quizás poco a poco te vayas a acostumbrando y lo empieces a usar, lo que sí es importante es que sepas como funciona porque te va a tocar leer código de otros que si lo usen.

Más de un return

Un punto que genera mucha polémica es que podemos tener más de un *return* en una función, hay casos en los que puede estar justificado, y otros que puede hacer que la función sea difícil de seguir, como regla general no usaremos más de un *return* a no ser que veamos claro que queda más legible la función.

Vamos a hacer otra refactor y tú me dices que te parece, tenemos esta función:

Primer paso, si no es número, directamente hago un *return* de *NO_ES_UN_NUMERO* y dejo de ejecutar el resto de la función:

```
function comprobarNumero() {
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);
  const resultado = NO_ES_UN_NUMERO;

+ if(!esUnNumero) {
+   return NO_ES_UN_NUMERO
+}

-  if (esUnNumero) {
    resultado = (numero === numeroParaAcertar) ? ES_EL_NUMERO_SECRETO :
NO_ES_EL_NUMERO_SECRETO;
-  }
  return resultado;
}
```

Ya que hemos simplificado esto, no nos hace falta tener una variable *resultado* directamente devolvemos el valor en el ternario:

```
function comprobarNumero() {
  const numero = parseInt(texto);
  const esUnNumero = !isNaN(numero);
-  const resultado = NO_ES_UN_NUMERO;

+ if(!esUnNumero) {
+   return NO_ES_UN_NUMERO
+}

-  resultado = (numero === numeroSecreto) ? ES_EL_NUMERO_SECRETO :
NO_ES_EL_NUMERO_SECRETO;
-  return resultado;
+  return (numero === numeroParaAcertar) ? ES_EL_NUMERO_SECRETO :
NO_ES_EL_NUMERO_SECRETO;
}
```

Resumamos que hemos hecho aquí:

- Por un lado chequeamos, si no es un número directamente devolvemos el valor de *NO_ES_UN_NUMERO* y se acabó.
- Por otro ya simplificamos, y directamente podemos devolver si es o no el número secreto.

El que esto sea bueno o malo, depende mucho de tus preferencias ¿Qué te parece? Es más limpia para ti esta aproximación... tanto si sí, como si no, es bueno que la conozcas porque te vas a encontrar código de otros escritos de esta manera.

Adivina con Pistas

Adivinar un número entre 0 y 100 sin que te den pistas es un poco aburrido, ¿Y si cada vez que digas un número, te responde el ordenador si este número es menor o mayor?

Vamos a pensar que tendríamos que hacer:

Por un lado tenemos esta enumeración:

Sería bueno cambiar NO_ES_EL_NUMERO_SECRETO por EL_NUMERO_SECRETO_ES_MAYOR, EL_NUMERO_SECRETO_ES_MENOR

Eso nos hará actualizar *muestraMensajeComprobación*: eliminamos el caso de *NO_ES_EL_NUMERO_SECRETO* y añadir el mensaje para los dos nuevos casos que hemos definido.

En *comprobarNumero* nos toca ahora cubrir los dos nuevos casos.

¿Qué no cambia? *handleCompruebaClick* se queda tal cual.

Vamos a realizar este refactor, primero añadimos los nuevos casos a las constantes de casos:

- Como hemos usado constantes y no hemos repetido el número por todo el código (lo que se dice evitar números mágicos), podemos modificarlos tranquilamente.
- Añadimos los dos casos y movemos el id del *ES_EL_NUMERO_SECRETO* al valor 3.

```
const NO_ES_UN_NUMERO = 0;
- const NO_ES_EL_NUMERO_SECRETO = 1;
+ const EL_NUMERO_ES_MAYOR = 1;
+ const EL_NUMERO_ES_MENOR = 2;
- const ES_EL_NUMERO_SECRETO = 2;
+ const ES_EL_NUMERO_SECRETO = 3;
```

Vamos a cambiar la función que cambia el mensaje de error:

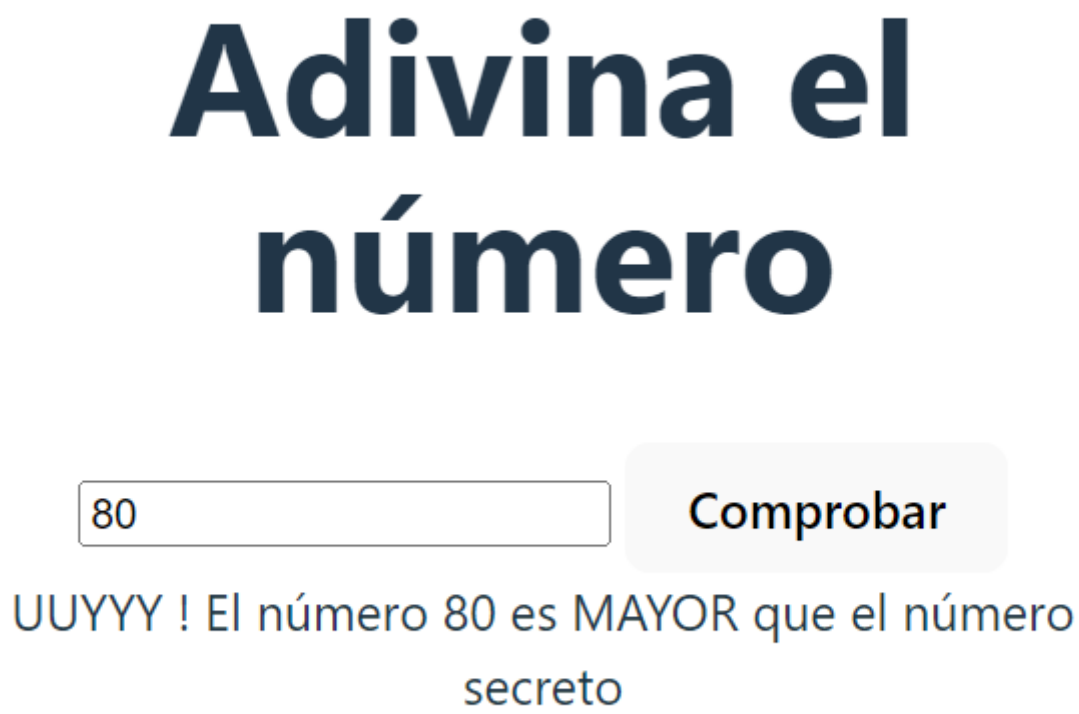
```
const muestraMensajeComprobacion = (texto, estado) => {
  let mensaje = "";
  switch (estado) {
    case NO_ES_UN_NUMERO:
      mensaje = `"${texto}" no es un numero 😐, prueba otra vez`;
      break;
-    case NO_ES_EL_NUMERO_SECRETO:
-      mensaje = `Lo siento ${texto}, el número no es el correcto 😢, prueba de
nuevo`;
-      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;
    default:
      mensaje = "No se que ha pasado, pero no deberías estar aquí";
      break;
  }

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

Y vamos ahora a por la comprobación:

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

  if (!esUnNumero) {
    return NO_ES_UN_NUMERO;
  }

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

```
+ return (numero > numeroParaAcertar ) ? EL_NUMERO_ES_MAYOR : EL_NUMERO_ES_MENOR

-  return numero === numeroAAcertar
-    ? ES_EL_NUMERO_SECRETO
-    : NO_ES_EL_NUMERO_SECRETO;
}
```

Vamos a probar que tal ha quedado la cosa 😃 (*npm run dev*).



¿Y si no hubiésemos hecho refactor?

Hasta ahora hemos estado puliendo un código, partimos de una versión en bruto y conforme avanzamos vimos la forma de ir mejorándolo, realizando refactors. A veces te podrás preguntar ¿Y no sería mejor hacerlo todo a lo burro y terminar antes? En este caso simple puede que quedara un código medio mantenible, pero a poco se complicará más (un caso real se suele complicar más y más 😅) y no habría por donde pillarlo.

Así pues vamos a realizar este mismo ejercicio a lo *bruto* y compararemos las diferencias:

OJO ESTO QUE VAMOS A PROGRAMAR ES MALA PRACTICA

Borramos todo y vuelta a empezar (ponte el delantal porque va a salpicar sangre...).

Lo primero ponemos la función en línea y recogemos el valor directamente.

```
const botonComprobar = document.getElementById("comprobar");
botonComprobar.addEventListener("click", () => {
  const texto = document.getElementById("numero").value;
});
```

Vamos ahora a ver si el número es válido o no:

```
const botonComprobar = document.getElementById("comprobar");
botonComprobar.addEventListener("click", () => {
  const texto = document.getElementById("numero").value;
+
+ if(isNaN(texto)) {
+   document.getElementById("resultado").innerHTML =`"${texto}" no es un numero
😐, prueba otra vez`;
+ } else {
+  if (numero === numeroParaAcertar) {
+   document.getElementById("resultado").innerHTML = `¡¡¡Enhorabuena, has acertado
el número!!! 🎉🎉🎉`;
+  } else {
+    if(numero > numeroParaAcertar) {
+      document.getElementById("resultado").innerHTML = `UUYYY ! El número
${texto} es MAYOR que el número secreto`;
+    } else {
+      document.getElementById("resultado").innerHTML = `UUYYY ! El número
${texto} es MENOR que el número secreto`;
+    }
+  }
+ }
});
```

Veamos qué pasa con este código:

- De primeras tenemos tres niveles de complejidad ciclomática (¿te acuerdas del pantallazo de Ruy soltando un Hadouken?) se empieza a poner un poco complicado de leer este código, si ahora añadimos contar número de intentos, terminar la partida si ha superado un máximo número de intentos y añadir lógica para empezar nueva partida ¿Cómo crees que quedará esto...?
- Después te tienes que bajar a nivel de detalle para entender este código, de primera vista no ves lo que hace.
- Si ahora en vez de mostrar un sólo texto, quiero involucrar varios elementos HTML y por ejemplo añadir una animación ¿Qué haría? repetiría estos casos en cada sitio del IF ¡Tengo que tocar en 5 sitios y no equivocarme!

Esto que acabas de aprender es algo muy importante, muchos programadores seniors ni lo ven ni lo aplican y son auténticos *carniceros* del software.

Te Toca

Es hora de ponerte manos a la obra, vamos a plantear un par de ejercicios 😊.

Recuperamos el código anterior (la versión limpia) 😁.

```
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 = 3;
const ES_EL_NUMERO_SECRETO = 4;

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;
    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;
  }

  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);
};

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

Máximo de intentos

El primer desafío que planteamos es el siguiente: queremos mostrar al usuario el número de intentos que lleva e indicarle el máximo de intentos que tiene disponible.

¿Te animas con el caso? Si no lo ves claro aquí van unas pistas:

- Te va a hacer falta una variable que lleve el número de intentos (defínela con let porque la irás modificando).
- Te va a hacer falta una variable que indique el número máximo de intentos (defínela con const porque no la vas a modificar), también podrías poner "un número mágico", pero siempre es mejor usar una constante para tenerlo definida en un sólo sitio.
- En el HTML puedes añadir un DIV para mostrar el número de intentos que llevas.
- Puedes crear un método que pinte por pantalla el número de intentos.
- En el click del botón *comprobar números* puedes incrementar el número de intentos y llamar a la función para que actualice el mensaje por pantalla.

En esta ocasión como utilizaremos el código generado como punto de partida para el siguiente ejercicio, aquí va la solución (inténtala tú antes 😊):

Primero vamos a definir la variable que almacenará el número de intentos que lleva el usuario (inicialmente cero) y la que lleva el número máximo de intentos (en este caso 5):

```
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 = 3;
const ES_EL_NUMERO_SECRETO = 4;

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

const muestraMensajeComprobacion = (texto, estado) => {
```

Vamos a añadir el *div* para el número de intento:

index.html

```
    <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>
```

- Ya que tenemos el div, creamos una función que pinte en ese contenedor lo intentos que llevamos:

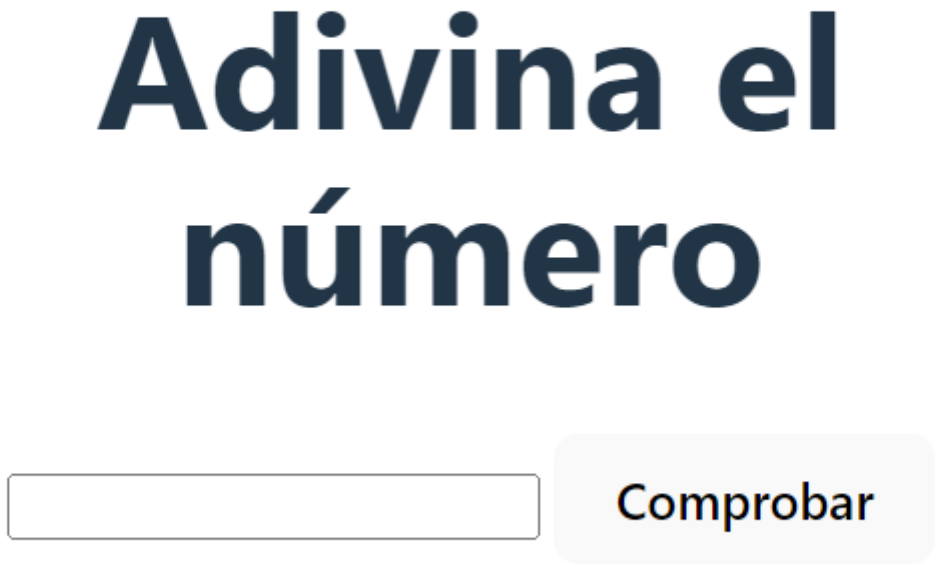
./src/main.js

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

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

const muestraMensajeComprobacion = (texto, estado) => {
```

- En el click del botón *comprobar números* puedes incrementar el número de intentos y llamar a la función para que actualice el mensaje por pantalla.

```
const handleCompruebaClick = () => {
  const texto = document.getElementById("numero").value;
+ numeroDeIntentos++; // Esto es equivalente a numeroIntentos = numeroIntentos +
1;
+ muestraNumeroDeIntentos();
  const estado = comprobarNumero(texto);
  muestraMensajeComprobacion(texto, estado);
};
```

Si te fijas, no se muestra el mensaje de número de intentos la primera vez que se carga la página.



Una opción que nos podríamos plantear es poner directamente una llamada a *muestraNumeroDeIntentos*, directamente en el código:

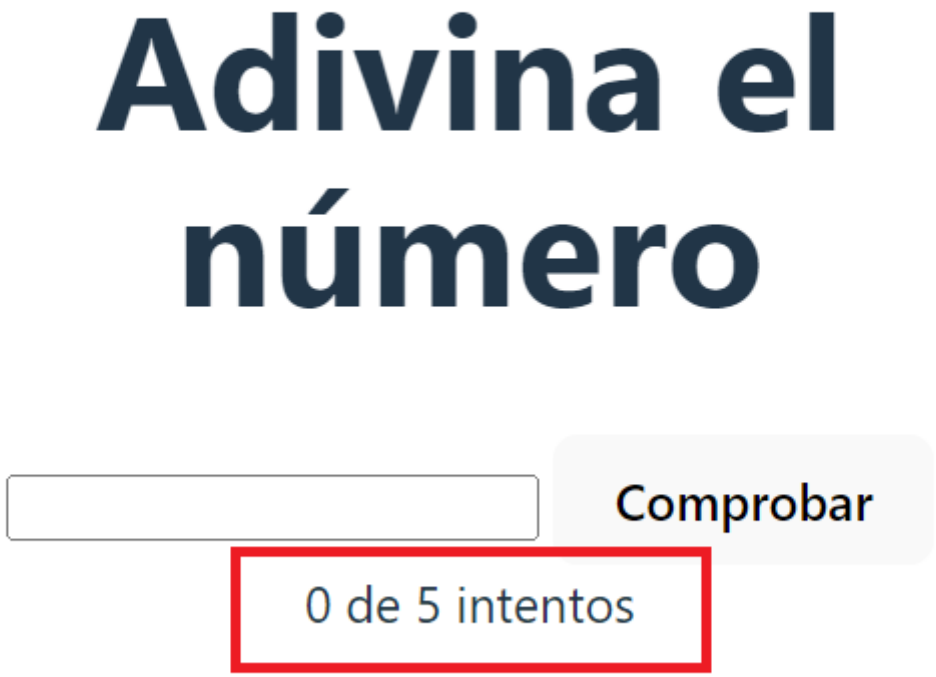
```
const MAXIMO_INTENTOS = 5;
let numeroDeIntentos = 0;

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

+ muestraNumeroDeIntentos();
```

Pero esto nos puede traer problemas ¿Por qué?

- La función *muestraNumeroDeIntentos* tira del DOM (el HTML) para pintar, y puede que cuando se ejecute esta línea el HTML no esté cargado.
- Si el HTML no está cargado, no podemos acceder a los elementos del DOM y nos daría un *castañazo* (si, de eso del tipo *undefined blah blah*).

¿Qué podemos hacer? Pues tenemos un evento que nos dice cuando está el DOM ya cargado, y es el evento *DOMContentLoaded*, nos suscribimos a él, y cuando se dé, que lance la función que necesitamos.

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

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

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



Game over

Siguiente desafío: si el usuario ha hecho más de 5 intentos se acaba el juego (inténtalo y si no.... a por las pistas 😊).

Pistas:

- Crea una función para comprobar si se ha superado el número máximo de intentos.

- Vamos a añadir un estado más GAME_OVER_MAXIMO_INTENTOS.
- En *muestraMensajeDeComprobación* añade un caso para GAME_OVER_MAXIMO_INTENTOS.
- Nos falta deshabilitar el botón de comprobar número si el juego ha terminado (game over):
    - Crea una función que se llame *gestionarGameOver* que reciba el estado.
    - Dentro de esa función chequea si el estado actual es GAME_OVER_MAXIMO_INTENTOS y en ese caso deshabilita el botón.
- Invoca la función *gestionarGameOver* desde *handleCompruebaClick*.

Si te has quedado atrancado, como este ejercicio tiene un poco más de complejidad, aquí va una posible solución.

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

+ const HasSuperadoElNumeroMaximoDeIntentos = () =>
+  numeroDeIntentos > MAXIMO_INTENTOS;
```

- Vamos a añadir el nuevo estado del juego (ojo el número 5 no tiene nada que ver con el del máximo de intentos, simplemente ha coincidido, podría valer 24).

```
const NO_ES_UN_NUMERO = 0;
const EL_NUMERO_ES_MAYOR = 1;
const EL_NUMERO_ES_MENOR = 3;
const ES_EL_NUMERO_SECRETO = 4;
+ const GAME_OVER_MAXIMO_INTENTOS = 5;
```

- Vamos a actualizar *muestra_mensaje*:

```
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 GAME_OVER_MAXIMO_INTENTOS:
+    mensaje = `🪦 GAME OVER, has superado el número máximo de intentos`;
+    break;
    case ES_EL_NUMERO_SECRETO:
      mensaje = `¡¡¡Enhorabuena, has acertado el número!!! 🎉🎉🎉`;
      break;
    default:
```

```
      mensaje = "No se que ha pasado, pero no deberías estar aquí";
      break;
  }
```

- Y en *comprobarNumero*, vemos el caso de que el usuario no haya acertado, si ha pasado el número de intentos para mostrar el mensaje de error.

```
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;
};
```

- Nos falta deshabilitar el botón de comprobar número si el juego ha terminado (game over):
  - Crea una función que se llame gestionarGameOver que reciba el estado.
  - Dentro de esa función si el estado de GAME_OVER_MAXIMO_INTENTOS deshabilita el botón.

```
document.addEventListener("DOMContentLoaded", muestraNumeroDeIntentos);

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

- Invoca la función *gestionarGameOver* desde *handleCompruebaClick*, así cada vez que el usuario intente adivinar un número, se comprueba si ha superado el número máximo de intentos.

```
const handleCompruebaClick = () => {
  const texto = document.getElementById("numero").value;
  numeroDeIntentos++; // Esto es equivalente a numeroIntentos = numeroIntentos +
1;
  muestraNumeroDeIntentos();
  const estado = comprobarNumero(texto);
```

```
    muestraMensajeComprobacion(texto, estado);
+   gestionarGameOver(estado);
};
```

Vamos a probarlo 😃

```
npm run dev
```

Seguramente te estés planteando dos cosas:

- Hay demasiado contenido en un fichero y cuesta de seguir.
- Es muy fácil cometer un fallo tonto en una función y es algo relativamente complejo de detectar.

En los próximos módulos vamos a aprender a:

- Separar código en varios ficheros (y cuando haga falta incluso en carpetas), para así poder organizarlo mejor.
- A implementar pruebas unitarias para poder probar funciones de forma aislada.

Si quieres seguir practicando por tu cuenta, podrías plantearte:

- Que cuando termine la partida también se considere game over y se deshabilite el botón.
- Añadir un botón *nueva partida* que sólo se habilite cuando está en Game Over y que reinicie el juego.

Si te surgen dudas, contacta con tu mentor.

Resumen

En este módulo hemos visto:

- Qué es un condicional.

- Como ir refactorizando una aplicación de forma iterativa para acabar con una base de código limpia y mantenible.
- Qué es la complejidad cilomática (o lo *hadouken*)
- Cuando hacer uso del *switch*
- Cuando hacer uso de un ternario.
- Compara el uso de código limpio vs una versión carnicera.