TAREA UT5

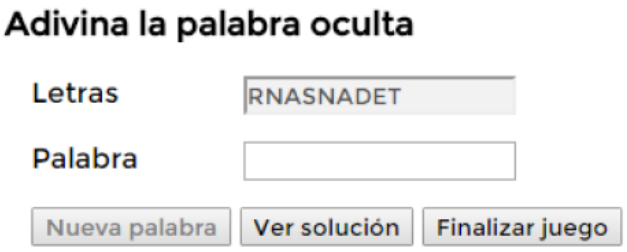
GESTIÓN DE EVENTOS Y FORMULARIOS

La tarea consiste en la implementación de varios scripts en lenguaje Javascript. Se valorará positivamente la claridad del código, el uso de las convenciones pedidas para el nombrado de ficheros, funciones, métodos, clases y variables y el uso de comentarios en el código, así como la corrección de los resultados obtenidos.

Es obligatorio implementar las funciones que se detallan en cada uno de los ejercicios, así como ceñirse a los nombres propuestos para las mismas, sus argumentos y tipos de entrada y valores de salida esperados. Todos los eventos se definirán mediante el modelo de eventos del W3C

Salvo que se indique lo contrario, todos los mensajes se mostrarán por la consola de depuración del navegador.

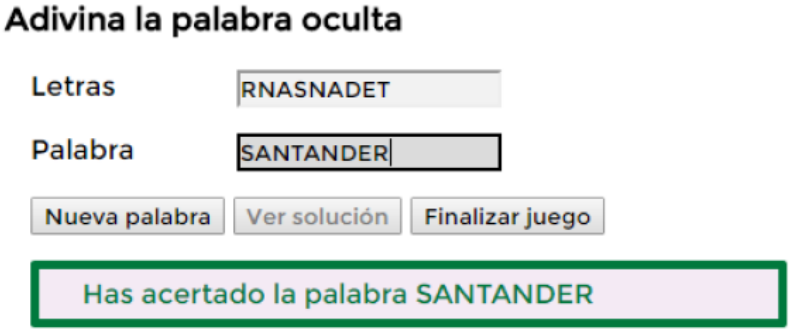
1. [2 puntos] Queremos implementar una página que muestre un pequeño juego que consiste en adivinar una palabra a partir de sus letras. En el campo de texto "Letras" (no editable) se mostrarán las letras de una palabra desordenada, mientras que en el campo de texto "Palabra" (editable) podremos intentar adivinar de qué palabra se trata:



Además tenemos 3 botones que nos permiten:

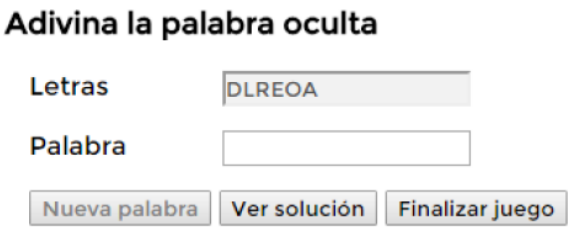
- Generar una nueva palabra (inicialmente deshabilitado).
- Ver la solución (palabra ordenada)
- Finalizar el juego.

Cada vez que introduzcamos una letra en el campo "*Palabra*", además de pasar la misma a mayúsculas, se comprobará si la palabra introducida coincide con la original. En caso afirmativo, se mostrará un mensaje indicativo, además de habilitar el botón para generar una nueva palabra y deshabilitar el que nos permite ver la solución de la palabra (pues la acabamos de mostrar).

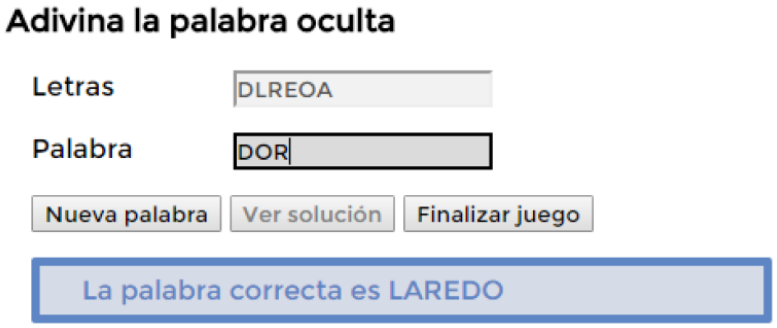


Al pulsar "*Nueva palabra*" se producirán las siguientes acciones:

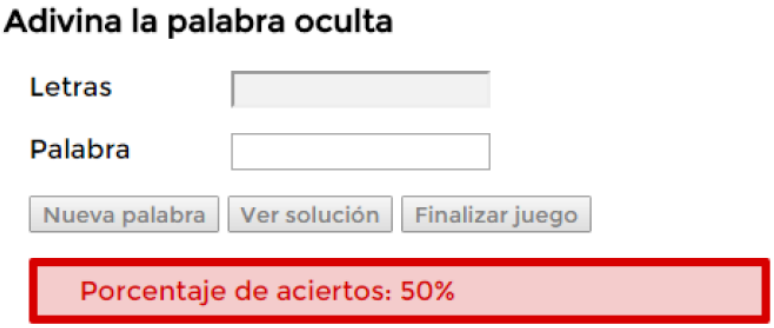
- Se generará una nueva palabra que se mostrará en la caja de texto "Letras" (la palabra se tomará aleatoriamente de entre al menos 10 valores.).
- Se borrará el contenido de "Palabra".
- Se restablecerán los botones a su estado inicial.
- Se ocultará el mensaje indicativo.



En cualquier momento podemos pulsar el botón "*Ver solución*". Este botón muestra un mensaje con la palabra ordenada, cambiando de nuevo el estado de los botones. Nótese que el formato del mensaje es distinto al del mensaje anterior.



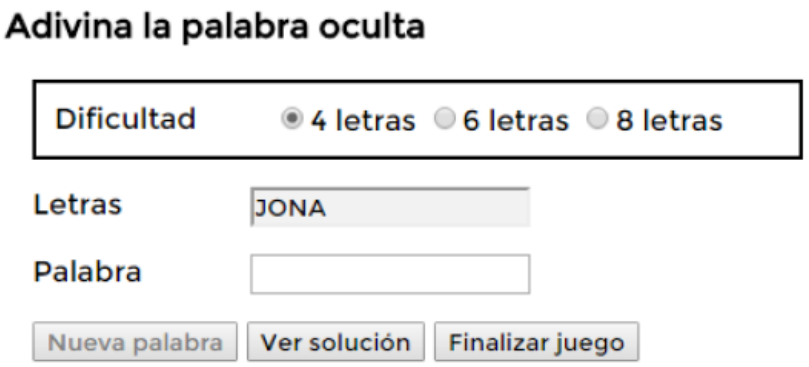
Por último, si mostramos el botón "Finalizar juego", se mostrará un mensaje (de formato distinto al demás) que nos mostrará el porcentaje de aciertos, es decir, el total de partidas que hemos acertado frente al total de partidas jugadas, expresado de manera porcentual.
Nótese que una vez que hemos pulsado este botón el resto de botones se deshabilitan, finalizando efectivamente el juego.



Para resolver el ejercicio deben usarse los siguientes archivos:

- **`adivinapalabra.js`**: Contiene la clase **`PalabraOculta`**, que contiene la lógica necesaria para manejar el juego. Será necesario determinar antes de comenzar a programar qué métodos y propiedades se necesitan (aunque puede ser adecuado focalizarnos en las propiedades y los métodos principales, para luego ir añadiendo los métodos a medida que los necesitemos).
- **`palabraoculta_prueba.js`**: Contiene la prueba de la clase **`PalabraOculta`**
- **`ejercicio1.js`**: Contiene los manejadores de eventos y el código de acceso al DOM.

2. [2 puntos] Modificar el ejercicio anterior para añadir una línea de botones de radio que nos permitan escoger la longitud de la palabra.



La palabra se tomará de manera aleatoria de un array de 3 filas por 10 columnas, donde cada fila contendrá las palabras de una de las longitudes (por ejemplo, la primera fila contiene 10 palabras de 4 letras, la segunda fila contiene 10 palabras de 6 letras y la tercera fila contiene 10 palabras de 8 letras).

Además, vamos a guardar una cookie sin caducidad con el valor del mejor porcentaje de partidas hasta la fecha. Esto implica que al pulsar el botón "Finalizar partida" realizaremos las siguientes acciones:

- Leeremos el valor de la cookie.
- Compararemos con el porcentaje obtenido en la partida actual. Si el porcentaje obtenido es más alto que el porcentaje leído, se actualizará el valor de la cookie y además de mostrar la información de la partida actual se informará al usuario de que ha batido el record. En caso contrario, simplemente informaremos de la puntuación obtenida.

3. [3 PUNTOS] Necesitamos una aplicación Web que nos permita almacenar discos de música de las pocas tiendas de discos que quedan.

   Cada disco de música va a almacenar los siguientes datos:

   - Título del disco
   - Cantante
   - Fecha de publicación
   - Tipo de música (puede ser "rock", "pop", "punk" o "indie")
   - Estantería: Almacena un número de estantería.
   - Prestado: Almacena un valor booleano.

Para almacenar un disco de música se usarán objetos literales. Por tanto, no se definirá la clase **`Disco`** (se crearán los objetos de manera literal bajo demanda.

NOTA: Para simular un tipo enumerado en Javascript podemos definir la estructura

```
const TIPO_MUSICA={
      "ROCK":1,
      "POP":2,
      "PUNK":3,
      "INDIE":4
};
```

En el archivo **`tienda.js`** crearemos una pseudoclase **`Tienda`** con la estructura detallada en la tabla siguiente:

Clase **`Tienda`** [Definida mediante función constructora]			
Propiedades			
Nombre	**Tipo**	**Visibilidad**	**Descripción**
`Discos`	`Disco[]`	Privada	-
Métodos			
Nombre	**Parámetros**	**Devuelve**	**Descripción**
`Tienda()`	-	-	Constructor sin parámetros
`getNumeroDiscos()`		entero	Devuelve el número total de discos que tiene la tienda.
`getNumeroDiscosDisponibles()`		entero	Devuelve el número de discos cuyo estado es disponible.

`addDisco(titulo, cantante, tipo, fecha estantería, prestado)`	`titulo:cadena`
	`cantante:cadena`
	`tipo:TIPO_MUSICA`
	`fecha:Date`
	`estanteria:entero`
	`prestado:booleano`
		`booleano`	Añade un disco a la tienda. Devuelve verdadero si se ha podido insertar correctamente y falso en caso contrario (por ya existir un disco con el mismo título).

`getTituloDisco(pos)`	`pos:entero`	`cadena`	Devuelve el título del disco que está en la posición **`pos.`** En caso de no existir el disco se devuelve **`null`**.
`getDisco(pos)`	`pos:entero`	`Disco`	Devuelve el disco que está en la posición **`pos`**
`deleteDisco(titulo)`	`titulo:cadena`	`booleano`	Elimina el disco con el título **`titulo`**. Devuelve verdadero si se ha podido eliminar correctamente y falso si no se ha podido eliminar (por no existir). **El método es independiente de mayúsculas/minúsculas**
`existeDisco(titulo)`	`titulo:cadena`	`booleano`	Devuelve verdadero si existe el disco con el título proporcionado falso en caso contrario. **El método es independiente de mayúsculas/minúsculas**

`modificaDisco(titulo, cantante, [tipo], [fecha], [estantería], [prestado])`	`titulo:cadena`
	`cantante:cadena`
	`tipo:TIPO_MUSICA`
	`fecha:Date`
	`estanteria:entero`
	`prestado:booleano`
		`booleano`	Modifica el disco con el título proporcionado. Los parámetros son opcionales de izquierda a derecha, es decir, podemos llamar al método de las siguientes maneras:
			`modificaDisco(titulo,cantante)`
			`modificaDisco(titulo, cantante, tipo)`
			`modificaDisco(titulo,cantante,tipo,fecha)`
			`modificaDisco(titulo,cantante,tipo,fecha,estanteria)`
			`modificaDisco(titulo,cantante,tipo,fecha,estanteria,prestado)`
			
			Devuelve verdadero si se ha podido modificar el disco y falso en caso contrario (por no existir).
			El método es independiente de mayúsculas/minúsculas

En el archivo **`tienda_prueba.js`** crearemos un objeto de tipo tienda donde insertaremos varios discos y se probará que la tienda funciona correctamente.

El archivo **`ejercicio3.html`** contiene el siguiente formulario, destinado a manejar datos de discos:

NO deberías tocar el código HTML del archivo. Nótese que en la parte superior hay una capa con el id `informacion` que está inicialmente oculta.

El archivo `formulario.js` debería contener:

- Una instancia de Tienda (puede tratarse como una variable global).
- Los manejadores de eventos pertinentes (**usando el modelo W3C**) para que se traten correctamente las pulsaciones de botones.
- Además el archivo podrá contener todas las funciones auxiliares que se juzgue necesario.

Al pulsar el botón ***Guardar disco*** deberá añadirse un nuevo disco a la `Tienda`, teniendo en cuenta las siguientes consideraciones:

- Las validaciones de los campos se realizarán al pulsar el botón.
- Todos los campos son obligatorios. Esto implica que si algún campo está vacío no se creará el disco, y se mostrará un mensaje indicativo en el div `informacion`, por ejemplo:

  Se considera un campo de texto vacío cuando no se ha introducido texto o se han introducido únicamente espacios.
  El año debe estar en formato DD/MM/AAAA, y se validará mediante una expresión regular. En caso de que el dato no sea correcto, tampoco se creará el disco y se mostrará otro mensaje indicativo.
- Estantería debería ser un valor numérico entre 1 y 10. Se validará (por el método que se prefiera) y en caso de error se mostrará el mensaje pertinente.
- Si los datos son correctos, pero ya existía un disco con el título que se quiere insertar, también se mostrará un mensaje indicativo del error:

- Si no estamos en las situaciones anteriores el disco se insertará, mostrando un mensaje indicativo en la capa con id `informacion`:

  Además, si procede se actualizará el listado de discos **disponibles** (no prestados) de la capa `listado`, tomando los datos de la instancia de `Tienda`:

Al pulsar el botón ***Eliminar disco*** se borra de la tienda el disco con el nombre insertado en el campo "*Nombre del disco*" (el resto de campos no se tendrán en cuenta al pulsar este botón). En caso de no existir ningún disco con el nombre, se mostrará el mensaje de error correspondiente:

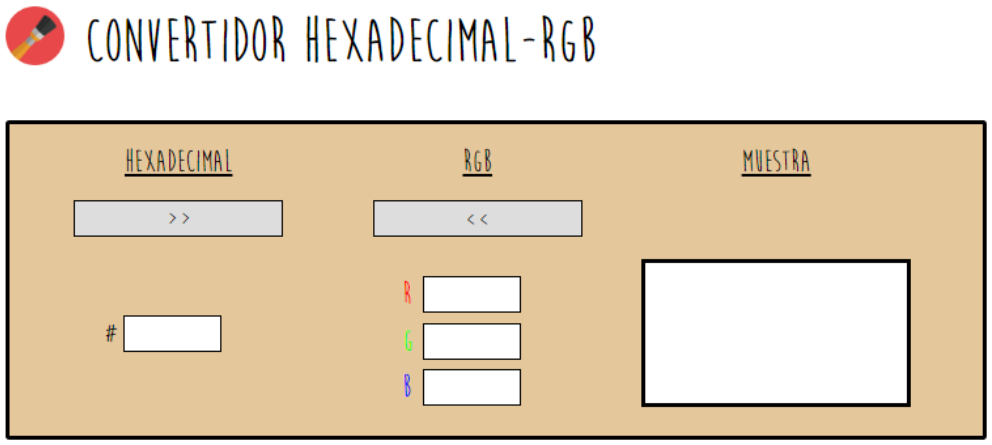
En caso de que el disco exista, se mostrará un mensaje indicativo de que se ha borrado y se actualizará si procede el listado de discos disponibles.

Al pulsar el botón ***Ver disco*** se mostrarán en el formulario los datos del disco con el nombre insertado en el campo "*Nombre del disco"*. En caso de que el disco no exista, se mostrará un mensaje indicativo como en el caso anterior. Si el disco existe, se ocultará el mensaje de información (pues no hay nada que mostrar) y se mostrarán los datos del disco correspondiente.

Al pulsar el botón ***Modificar disco*** se actualizarán los datos del disco con el nombre insertado en el campo "*Nombre del disco*", entendiendo que dicho nombre no puede ser modificado. En caso de intentar modificar el nombre del disco (o introducir un nombre no existente), se mostrará un mensaje indicativo como en los casos anteriores. Con el resto de campos se obrará de manera análoga a la creación de un ***Nuevo disco*** (campos obligatorios, formato de los campos y mensajes de error/éxito).

Una vez modificado el disco se actualizará el listado de discos disponibles (si procede).

4. [2 puntos] Vamos a implementar una aplicación Web que al cargarse tiene el siguiente aspecto. El archivo **`ejercicio4.html`** contiene el siguiente formulario:



El formulario permite especificar un color en modelo hexadecimal y convertirlo a RGB o viceversa, mostrando en el panel derecho una muestra de color.

NO está permitido tocar el código fuente correspondiente al formulario. Nótese que en la parte inferior hay dos capas con id **`errores`** y **`listado`** respectivamente que inicialmente están ocultas.

Todos los eventos se definirán e implementarán en el archivo **`convertidor.js`** usando el modelo W3C.

NOTA: Para una explicación más detallada sobre los modos de color y cómo convertir entre ellos, consultar el Anexo I: Colores Web.

Para realizar las validaciones se pide implementar en el archivo **`funciones_auxiliares.js`** las siguientes funciones auxiliares:

Función	Descripción
`esNumeroValido(numero):booleano`	Devuelve verdadero si recibe un valor numérico entre 0 y 255 y falso en caso contrario.
`esHexadecimal(cadena):booleano`	Devuelve verdadero si la cadena representa un hexadecimal de 6 cifras y falso si no es así. Para realizar la validación se pueden utilizar expresiones regulares o funciones de tratamiento de cadenas, a preferencia del alumno.

En el archivo **`convertidor_prueba.js`** se realizarán varias invocaciones a cada una de las funciones (al menos una con datos correctos y otra con datos incorrectos), mostrando los resultados por consola para comprobar que las funciones están correctamente implementadas.

Para almacenar los colores y realizar las conversiones se pide implementar en el archivo **color.js** una clase llamada **Color** que nos permite gestionar un color tanto en formato RGB numérico como en Hexadecimal

Clase **Color** [Definida mediante prototipos]			
Propiedades			
Nombre	Tipo	Visibilidad	Descripción
valorRGB	entero[3]	Privada	Array de 3 elementos que contiene la representación en RGB del color. El primer elemento del array es el componente rojo, el segundo es el elemento verde y el tercero es el componente azul.
valorHexadecimal	Cadena	Privada	Cadena de caracteres que contiene la representación en hexadecimal del color.
Métodos			
Nombre	Parámetros	Devuelve	Descripción
Constructor() **Constructor(hex)** **Constructor (r,g,b)**	hex:cadena r:entero g:entero b:entero	-	Constructor sobrecargado: • En caso de no recibir ningún parámetro, el color se inicializará a negro (#000000). • En caso de recibir un único parámetro, éste representa una cadena hexadecimal. • En caso de recibir tres parámetros, estos representan los colores rojo, verde y azul respectivamente. Independientemente del número de parámetros recibido, las propiedades **valorRGB** y **valorHexadecimal** tendrán que inicializarse correctamente.
setRGB(rgb)	rgb:entero[3]	-	Establece el valor de la propiedad **valorRGB** Nótese que el valor de la propiedad **valorHexadecimal** debería actualizarse.
setHex(cadena)	hex:cadena	-	Establece el valor de la propiedad **valorHexadecimal** Nótese que el valor de la propiedad **valorRGB** debería actualizarse.
getRojo()	-	entero	Devuelve el valor correspondiente al rojo.
getVerde()	-	entero	Devuelve el valor correspondiente al verde.
getAzul()	-	entero	Devuelve el valor correspondiente al azul.
getHex()	-	cadena	Devuelve el valor hexadecimal.
RGB2Hex(rgb)	rgb:entero[3]	cadena	Método privado. Recibe 1 array con los 3 valores en RGB y calcula su equivalente en hexadecimal, devolviéndolo en forma de cadena.
hex2RGB (cadena)	hex:cadena	entero[3]	Método privado. Recibe 1 cadena que representa un color en formato hexadecimal y calcula su equivalente en RGB, devolviéndolo en forma de array de 3 elementos.

No es necesario realizar las validaciones de parámetros, pues estas se realizarán antes de la invocación de los métodos correspondientes.

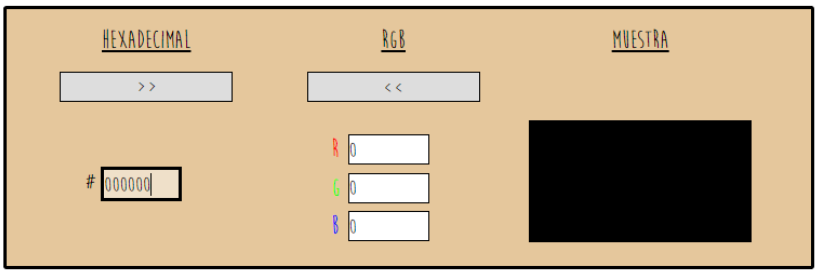
El archivo **color_prueba.js** contendrá una pequeña prueba donde se creen 2 instancias:

- Una que use el constructor con 1 parámetro.
- Otra que use el constructor con 3 parámetros.

Tras crear las instancias se comprobará que los métodos funcionan correctamente

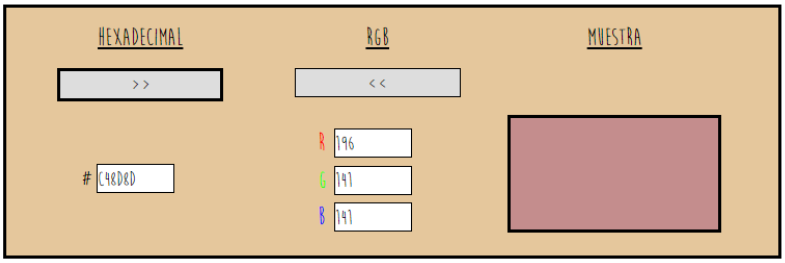
Los valores iniciales de los campos deberían ser los correspondientes al color negro. Para ello, en el evento de carga de página tendremos que instanciar un nuevo objeto de tipo **`Color`** (usar constructor sin parámetros) y rellenar los campos de texto de la página a partir de los getters del objeto.

Además, el foco debería estar situado en la caja de texto correspondiente al valor hexadecimal.

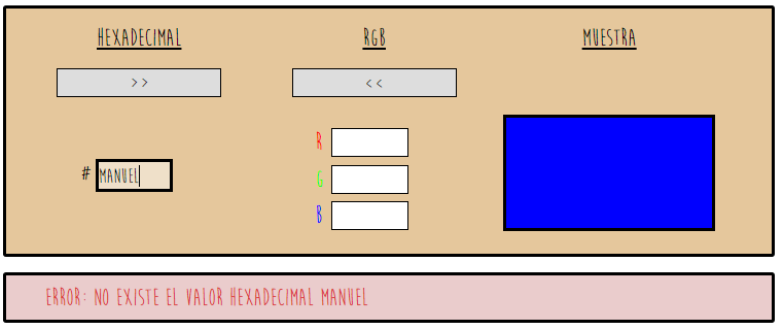


La funcionalidad de los botones >> y << debería ser la siguiente:

- Al pulsar sobre el botón >> el valor hexadecimal se convertirá a RGB y se muestra el color de fondo sobre la capa de muestra. Para realizar las conversiones se usará una nueva instancia de **`Color`** que crearemos en el evento de pulsación sobre el botón.

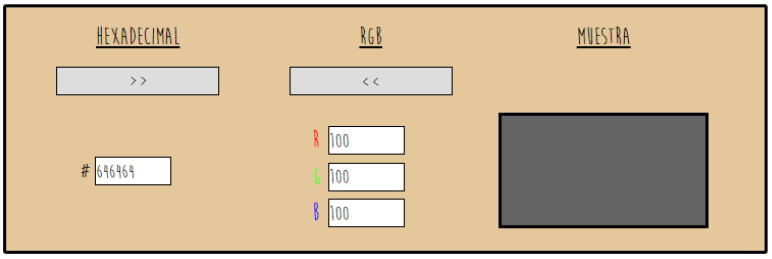


  En caso de que al pulsar el botón el campo hexadecimal tenga un valor no válido (habrá que realizar la validación previamente a crear la instancia de **`Color`**), aparecerá un mensaje en el campo con id **`errores`** y el **foco pasará al campo hexadecimal**:

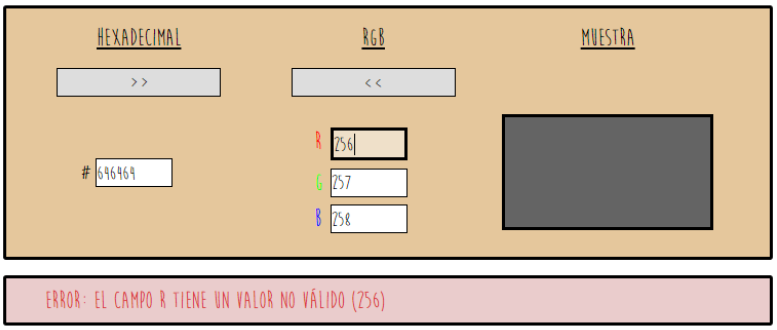


  Ten en cuenta que la capa de errores no debería mostrarse si el valor introducido es correcto.

- Al pulsar sobre el botón << los valores R, G y B se convierten a hexadecimal y se muestra el color de fondo de la muestra. Para realizar las conversiones se usará una nueva instancia de **`Color`** que se creará en el manejador del evento de pulsación sobre el botón.



En caso de que al pulsar el botón uno de los campos tenga un valor no válido (habrá que realizar la validación previamente a crear la instancia de **`Color`**), deberá mostrase un mensaje indicativo en el contenedor errores, pasando el foco al primer campo que tiene valor incorrecto:



Se quiere recordar el último valor correcto que hemos introducido antes de cerrar la página. Para ello, vamos a **guardar el valor hexadecimal en una** cookie una vez que se cierra la página. Se usará el evento **`onunload`** de **`window`** para detectar el cierre de página. La cookie tendrá un periodo de caducidad de 1 semana.

Al **cargar la página** de nuevo, obtendremos el valor de la cookie y actualizaremos de manera oportuna los campos de texto y el panel, borrando la cookie a continuación. En caso de que no exista la cookie al cargar la página, se cargará el color por defecto de la página (negro) como se explica en el primer punto.

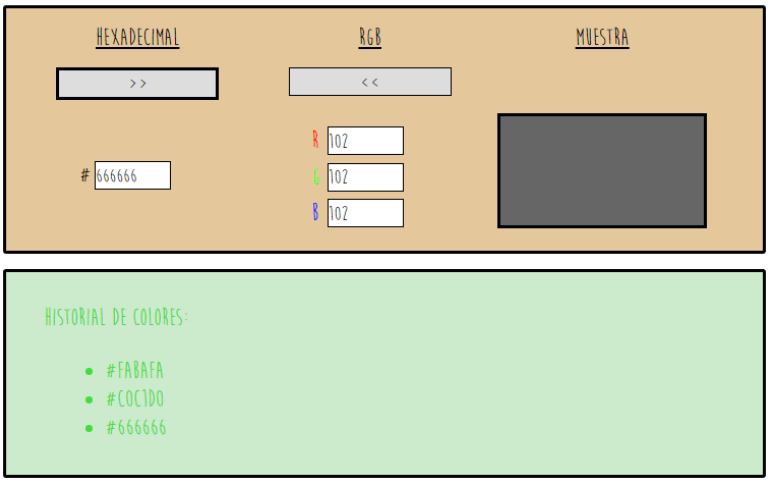
5. [1 punto] Modifica el ejercicio 4 para guardar un historial de los colores que se han convertido correctamente. Para ello, se implementará usado prototipos la clase **`ListaColores`** en el archivo **`lista_colores.js`** con los siguientes elementos:

Propiedad	Descripción
`colores: Array`	Array que contiene un listado de objetos de tipo Color

Método	Descripción
`Constructor()`	Constructor sin parámetros
`addColor(color)`	Añade un objeto de tipo Color al listado de colores si el color no existía previamente. Para saber si el color ya existe se consultará su valor hexadecimal.
`getColor(i): Color`	Devuelve el color existente en la posición i
`getNumeroColores():entero`	Devuelve el número de colores que contiene el listado.

`toHTML():Cadena`	Devuelve una representación en formato HTML de listado de colores, del tipo:
	`<ul>`
	`<li>#FABAFA</li>`
	`<li>#C0C1D0</li>`
	`<li>#666666</li>`
	`</ul>`

En el archivo convertidor.js se instanciará una variable global de tipo **`ListaColores`**. Cada vez que convirtamos un color con éxito se añadirá a dicha instancia el objeto que representa el color convertido. Además, se actualizará el listado de la capa **`listado:`**



La capa **`listado`** inicialmente está oculta, luego habrá que mostrarla la primera vez que se carga un color. Por otra parte, el color que se muestra no se mostrará en el listado.

Al cerrar la página, habrá que guardar un listado de los valores convertidos usando la API **localStorage**. Cuando carguemos la página, tendremos que comprobar si existe dicho listado, y en caso afirmativo cargar los colores en la capa de historial, mostrando el último color en el panel de conversión Hexadecimal / RGB.