Ejercicios prácticos y exámenes para las asignaturas

*Aplicaciones Telemáticas*
*y*
*Diseño de Aplicaciones Telemáticas*



Material docente en abierto de la Universidad Rey Juan Carlos
Escuela de Ingeniería de Fuenlabrada

# Ejercicios prácticos

**Prácticas de HTML**
Aplicaciones Telemáticas
Grado en Ingenieria Telemática, 2023-2024
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

# Formato de la memoria de prácticas

La memoria de estas prácticas debes escribirla en un fichero en formato markdown. El fichero estará en tu cuenta del laboratorio. El dia del examen, entregarás automáticamente tus prácticas mediante un script que te indicará el profesor.

1. Es imprescindible que respetes al pie de la letra los nombres de los ficheros especificados en el guión. Una letra mal puesta equivale a una práctica no presentada (o un examen no presentado). Si bien dispondrás de un script que verificará que has usado los nombres correctos

2. Redacta la memoria describiendo escuetamente todo lo que haces. Esta memoria te será muy útil para preparar el examen práctico (recuerda que podrás llevarla al examen).

   No merece la pena que te preocupes de tener una redacción muy cuidada: copia y pega órdenes y resultados, describe telegráficamente lo que haces. Vete preparando la memoria a la vez que trabajas, no lo dejes para el final.

   *a*) Cuando un paso del guión te pida una orden de shell o similar, es recomendable que primero ejecutes la orden y luego copies y pegues en la memoria. Si resulta algún texto, pégalo también.

   *b*) Cuando el guión pida que escribas o edites un fichero, basta con que lo modifiques en su sitio. No es necesario que copies y pegues en la memoria.

# Práctica 1.0. Preparación de los directorios

1. Crea el directorio `~/at`. Aquí guardarás la mayoría de tu trabajo de prácticas en la asignatura

2. Ponle permisos `rwx------`

   Debes mantenerlo así todo el curso. Recuerda que debes ser autor del 100% de tus prácticas y no permitir que ningún compañero tuyo tenga acceso a ningún fragmento.

3. Crea el directorio

   `~/at/practica01`

4. Crea el fichero

   `~/at/practica01.md`

   Donde escribirás esta memoria de prácticas en formato markdown

   https://gsyc.urjc.es/~mortuno/at/markdown.pdf

   https://www.markdownguide.org/basic-syntax

   https://guides.github.com/features/mastering-markdown

   Observa que el nombre de este fichero **no** es

   ```
   ~/at/practica01/practica01.md       # ¡Este no es el nombre correcto!
   ```

   Esta memoria de prácticas la escribirás en formato markdown

5. Escribe dentro tu nombre, apellidos y login.

6. Crea el directorio

   `~/at/images`

   En él podrás guardar imágenes para tus memorias de prácticas. Tal vez no lo uses, bien porque tu memoria no tenga imágenes, bien porque las dejes *online* en algún servidor. Pero crea el directorio igualmente. Observa que nos referimos ahora a las imágenes de las memorias de prácticas. Durante el curso escribirás también diversos ficheros HTML, que tendrán su propio directorio de imágenes.

7. Seguiremos este convenio en todas las prácticas, un directorio para cada práctica y un fichero de texto plano para cada práctica, fichero que cuelga de `~/at`

# Práctica 1.1. Markdown

Escribe un fichero en formato *markdown* con el nombre `~/at/markdown.md`. Prueba los elementos básicos de markdown que hemos visto en clase. Genera una versión en HTML con pandoc y css. Usa pandoc para *limpiar* el markdown.

# Práctica 1.2. W3C Validator

Analiza con el *W3C Markup Validation Service* el código de 3 sitios web. Han de pertenecer a organismos o empresas de cierta relevancia.

En el fichero `~/at/practica01.md` escribe un pequeño informe con los resultados, explicando en qué consisten y cómo se arreglarían. Si son muchos no es necesario que los describas todos, es suficiente con 10 errores (entre las tres páginas).

# Práctica 1.3. Elementos HTML

- Prepara un fichero HTML llamado `~/at/practica01/ej01.html` que contenga un ejemplo de cada uno de los elementos HTML vistos en clase:

  1. Comentario (con tu nombre, apellidos y login)
  2. p, br, em, pre
  3. h1, h2 (y algún otro h si quieres)
  4. Enlaces a otra página
  5. Enlace a un apartado de la página actual
  6. div, span
  7. ol, ul, dl
  8. img
  9. img dentro de un enlace

  No especifiques colores, fuentes, tipos de letra ni nada similar. En el siguiente tema veremos cómo hacerlo de forma correcta.

  Guarda las imágenes en el directorio `~/at/practica01/images`

  Puedes usar el editor que prefieras, pero debería ser uno con soporte para HTML. Por ejemplo *atom*.

- El contenido no es relevante, usa cualquier texto de prueba.
- Ignora todo aquello que es propio de HTML 4 y obsoleto en HTML 5. Recuerda que salvo indicación en contrario, todo el HTML que manejamos en la asignatura es HTML 5.
- Comprueba con el W3C validator que tu código no provoca avisos ni errores.
- Recuerda que en prácticas como esta donde lo único que tienes que hacer es escribir un fichero html, no es necesario añadir nada en el fichero de memoria de prácticas (*practica01.md*).

## Práctica 1.4. FreeFileSync

Para esta práctica necesitas tu propio ordenador, posiblemente tendrás que hacerla en casa (a menos que hayas llevado el portátil a clase).

- Crea en algún lugar de tu ordenador de casa una carpeta llamada *at*. No importa si es Linux, Windows o macOS. Usando FreeFileSync, haz que sea una réplica (bidireccional) de tu directorio `~/at`
- Provoca un conflicto, intencionadamente
- Soluciona el conflicto
- Durante todo el curso, mantén sincronizadas estas dos carpetas.

Sobre esta práctica no es necesario que escribas memoria. Pero es una práctica **imprescindible**. No hacerla correctamente podría suponer un suspenso en la asignatura: si se perdieran tus prácticas del laboratorio (algo difícil pero no imposible), podrás entregar la copia que realizas con este método. Si no tuvieras una copia al día, tendrías que repetirlo todo. Si no te diera tiempo, suspenderías.

## Práctica 1.5. Formularios

En este ejercicio practicarás el uso de formularios HTML.
Prepara dos ficheros HTML llamados `~/at/practica01/alta.html` y `~/at/practica01/operacion.html`

- Contendrán formularios para darse de alta y para operar una aplicación web. Puede ser un banco, una compañía de seguros (hogar, automóvil...), una compañía telefónica, una agencia de transporte, de paquetería, viajes, empleo, alquiler de vehículos, etc. Ambas páginas se corresponderán al mismo negocio.
- El formulario de la página `alta.html` solicitará al usuario los datos necesarios para darse de alta en esta compañía imaginaria.
- El formulario de la página `operacion.html` permitirá al usuario introducir la información necesaria para una operación cualquiera de este negocio. Por ejemplo, en el caso del banco la página de operación puede ser para realizar una transferencia bancaria, revisar recibos, revisar cargos en cuenta o en tarjeta de crédito. Si es una compañía de seguros, la operación puede ser dar un parte. En una agencia de transporte, encargar una recogida de un paquete para un envío, etc
- Haz que los campos sean razonablemente realistas, aunque no te preocupes por nada relativo al aspecto gráfico o el formato. Procura usar el mayor tipo de elementos de formulario posibles.

  Sugerencia: entre en alguna página web de una empresa real y tómalo como referencia.
- Usa siempre el elemento `<label>`
- Si lo deseas, puedes añadir más de una página de operaciones, con los nombres

  `~/at/practica01/operacion02.html`, `~/at/practica01/operacion03.html`, etc.
- Comprueba con el W3C validator que tu código no provoca errores. Podrá dar avisos relativos al uso de *input color* o *input date*, pero verifica que no da ningún otro aviso.

## Revisión de los nombres de los ficheros

Ejecuta el script `~mortuno/revisa practicas at` para comprobar que los nombres de tus ficheros son correctos.

**Prácticas con CSS y Bootstrap**
Aplicaciones Telemáticas
Grado en Ingeniería Telemática, 2023-2024
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

# Práctica 2.0. Fichero con la memoria de prácticas

Prepara un fichero de nombre `~/at/practica02.md` y contesta en él este apartado y todos los posteriores (excepto cuando la memoria de prácticas te pida que escribas un fichero distinto)

# Práctica 2.1. Análisis de selectores

Observa los siguientes ejemplos de código html con css y explica por qué cada línea tiene el formato que tiente.

Ejemplo 1
Ejemplo 2
Ejemplo 3
Ejemplo 4.

# Práctica 2.2. Uso de selectores

En esta práctica escribirás un fichero HTML con reglas CSS, similar a los ejemplos de la práctica anterior.

1. Prepara un fichero HTML llamado `~/at/practica02/ej22.html`

   y define en él al menos 5 reglas CSS para probar los distintos tipos de selector que vimos en clase (los que aparecen en las transparencias).

   Escribe en el fichero HTML diversos elementos donde se apliquen las reglas. Para cada regla, debe haber

   - Un elemento donde se aplique la regla.
   - Un elemento donde no se aplique la regla porque no se cumplan todos los requisitos del selector, pero sí alguno. En otras palabras: un elemento donde un programador *despistado* pueda pensar que la regla se aplica, pero que en realidad no se aplica.

2. Explica brevemente en el fichero `~/at/practica02.md` cada uno de los casos.

3. Comprueba con el W3C validator que tu código no provoca avisos ni errores.

# Práctica 2.3. Atributos del texto

Prepara un fichero HTML llamado
`~/at/practica02/ej230.html`
que contenga al menos 6 cajas de prueba, donde el texto tenga distintos atributos relacionados con alineación, subrayado, tamaño y estilo. Hazlo usando clases.

## Práctica 2.4. Atributos de las cajas

Prepara un fichero HTML llamado
`~/at/practica02/ej240.html`
que contenga al menos 4 párrafos de prueba, donde las cajas y los bordes de cada caja tengan distintos atributos relacionados con ancho, estilo y color. Hazlo usando clases. El color del texto también deberá ser adecuado, aunque solo sea blanco o negro. Añade imagen de fondo al menos a un elemento. Selecciona los colores usando algún generador de paletas como `https://coolors.co`.

## Práctica 2.5. Horario

En este ejercicio empezarás a usar la rejilla de Bootstrap.

- El nombre del fichero será `~/at/practica02/horario.html`
- En su interior codifica:
  - En un contenedor de tipo fluid, tu horario del primer cuatrimestre de este curso.
  - En un contenedor ordinario (no fluid), tu horario del segundo cuatrimestre de este curso.
- Emplea la rejilla de Bootstrap, con el tipo de cuadricula que creas más oportuno.
- Usando CSS, añade color a toda la página. Por ejemplo, que las celdas de la misma asignatura tengan el mismo color de fondo, o alguna otra organización de tipo lógico que te parezca adecuada. Selecciona la paleta usando algún generador de paletas como `https://coolors.co`

Recuerda que es necesario que revises este documento con el W3C validator, como todos los de la asignatura.

## Práctica 2.6. Rejilla de Bootstrap

Escribe un fichero `~/at/practica02/grid.html` que, basado en Bootstrap, tenga la siguiente estructura:

```
xxxxxxxxxxxx
xxx xxx xxx xxx
xxx xxx xxx xxx
xxx xxx xxx xxx
xxxx xxxx xxxx
```

Esto es,

- Primera fila: una columna de 12 casillas (con texto H1)
- A continuación, 3 filas con 4 columna iguales
- Finalmente, una fila de 3 columnas iguales

El contenido será texto de prueba. En las filas de el medio deberá haber al menos 4 imágenes (en total). Usa el tipo de contenedor y el tipo de cuadrícula que creas más oportuno.

## Práctica 2.7. CV

Prepara una versión de tu currículum en HTML usando Bootstrap 5 y su rejilla. Ponle color mediante CSS. El fichero se llamará `~/at/practica02/cv.html` Incluye todas tus asignaturas aprobadas con su calificación. Si por motivos de privacidad prefieres no incluir datos personales reales en este ejercicio, puedes hacer una versión imaginaria (irónica, sarcástica o simplemente falsa), con tal de que quede claro que no es real.

## Práctica 2.8. Carousel

Prepara un carrusel de Bootstrap en un fichero con nombre `~/at/practica02/carousel.html`

- Contendrá entre 4 y 6 fotos hechas por tí o donde salgas tú. Pon cualquier cosa que te guste y que no tengas inconveniente en enseñar: comida, viajes, mascotas, personas, aficiones, libros, discos... Pon títulos realistas.
- Recuerda que debes revisar siempre el código con el W3C validador.

## Práctica 2.9. Formularios Bootstrap

- Copia en `~/at/practica02/` los ficheros de práctica de formularios de la práctica 1 (`alta.html`, `operacion.html` y los ficheros optativos `operacion02.html` y `operacion03.html` si los tienes)
- Modifícalos para que utilicen Bootstrap.
- Recuerda que el código no debe generar ningún error en el W3C validator.

## Práctica 2.10. Landing Page

Prepara una *landing page* con Bootstrap. Su nombre será `~/at/practica02/landing.html` y todos los ficheros que necesite estarán dentro de `~/at/practica02/`. Organiza los ficheros en subdirectorios como te parezca conveniente.

- Puede ser sobre un organismo o negocio real o inventado, una página en la que te ofrezcas como *freelance* o como empleado... cualquier cosa similar, tienes libertad.
- La página tiene que ser razonablemente realista, puedes poner algún *lorem ipsum* en los párrafos pero la organización y los títulos tienen que tener sentido. Puedes reutilizar apartados anteriores de esta práctica.
- Puedes inspirarte en alguna plantilla gratuita para Bootstrap. Encontrarás muchas en `https://startbootstrap.com`, aunque desde cualquier buscador podrás encontrar páginas similares usando las palabras clave *free bootstrap templates* o *free bootstrap themes*.

  Estas plantillas normalmente usan características de Bootstrap un poco más avanzadas que las que hemos visto en clase, aquí esperamos que tu ejercicio sea más sencillo.
- No *piratees* fotos. Aunque técnicamente nada impide copiar cualquier foto de cualquier página web, desde el punto de vista legal se entiende que, salvo indicación en contrario, el autor mantiene todos sus derechos y no permite la reproducción.

  Para este ejercicio es necesario que todas las imágenes que incluyas o bien sean tuyas o bien tengan una licencia que permita su reutilización, de tipo *Creative Commons* o similar.
- Te será útil algún *stock* de imágenes gratuitas. Por ejemplo `https://www.pexels.com`. Podrás encontrar muchas otras páginas similares en cualquier buscador usando las palabras clave *free photo stock*.
- PRACTICA OPTATIVA: Si lo deseas puedes usar como base alguna plantilla de las indicadas anteriormente. Ten en cuenta que tendrás que documentarte sobre aspectos de Bootstrap fuera del ámbito de la asignatura. Por otro lado, el código de estas plantillas es de calidad muy variable, algunas son fáciles de personalizar, otras no tanto.

## Revisión de los nombres de los ficheros

Ejecuta `~mortuno/revisa practicas at` para comprobar que los nombres de los programas son los correctos.

**Prácticas con JavaScript**
Aplicaciones Telemáticas, 2023-2024
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

# Instrucciones generales

- Estos ejercicios deberán estar escritos en ECMAScript 6, preparado para ejecutarse en node.js, en modo estricto.
- Los enunciados te pedirán una o varias funciones. Para probarlas, incluye en el propio fichero diversas llamadas a cada función, con diversos valores. Muestra por salida estándar el resultado. Ejemplo:

```
'use strict'
function suma(x,y){
    return(x+y);
}

let a,b;
a=2;
b=2;
console.log(suma(a,b));

a=0.15;
b=0.50;
console.log(suma(a,b));

a=-1;
b=0;
console.log(suma(a,b));

a=undefined;
b=0;
console.log(suma(a,b));

// (etc)
```

- Naturalmente, cuando un enunciado pida *una función* que haga algo, esa función a su vez podrá llamar a todas las funciones necesarias para tener un diseño razonable.

# Práctica 3.1. Tipos de datos

Escribe en un fichero `~/at/practica03/test01.js` un programa JavaScript similar a los ejemplos de las transparencias (pero no igual) que haga conversiones entre distintos tipos de datos y que detecte el valor NaN.

# Práctica 3.2. Expresiones

Escribe en un fichero `~/at/practica03/test02.js` un programa JavaScript similar a los ejemplos de las transparencias (pero no igual) para probar ejemplos básicos de expresiones aritméticas y expresiones booleanas, con el doble igual y con el triple igual.

# Práctica 3.3. Funciones

Escribe en un fichero `~/at/practica03/test03.js` un programa JavaScript similar a los ejemplos de las transparencias (pero no igual) con funciones que ilustren que en JavaScript el paso de valores a

las funciones es por valor y no por referencia. También el uso de valores por omisión de un parámetro (al estilo ECMAScript 6).

## Práctica 3.4. Funciones flecha

Escribe en un fichero `~/at/practica03/test04.js` un programa JavaScript similar a los ejemplos de las transparencias (pero no igual) con una función sencilla con notación tradicional, y una función equivalente con notación *flecha*. Prueba la función flecha usándola como argumento de otra función.

## Práctica 3.5. Funciones básicas

Escribe un fichero `~/at/practica03/js01.js` que contenga:

- Una función que reciba una distancia en metros y la devuelva en centímetros.
- Una función que reciba una distancia en metros y la devuelva en pulgadas. (Una pulgada son 0.0254 metros)
- Una función que reciba una distancia en metros y la devuelva en yardas. (Un yarda son 0.9144 metros)

## Práctica 3.6. Funciones.

Escribe en un fichero `~/at/practica03/convierte.js` un programa JavaScript que tenga una función que reciba dos parámetros:

- Una distancia expresada en metros. Este parámetro podrá ser tanto un número como una cadena
- Una cadena que podrá tomar los valores `m`, `cm`, `in` o `yd`.

  Los valores han de ser exactamente estos y no otros similares. No se admiten variaciones en mayúculas/minúsculas o espacios. Si el programador omite este parámetro, tu programa deberá detectarlo.

La función devolverá

- Una cadena (no un número) con la distancia de entrada, convertida a las unidades expresadas en el segundo argumento (metro, centímetro, pulgada o yarda).

  Si ha habido algún problema con los parámetros de entrada, la función devolverá una cadena que empezará por `Error:` (La primera letra en mayúscula), y que a continuación explicará la causa del error: que falten argumentos, que los argumentos sean incorrectos, etc

## Práctica 3.7. Búsqueda en arrays

Escribe un fichero `~/at/practica03/buscaArrays.js` que contenga una función que reciba dos argumentos

- Un array de arrays
- Un elemento

La función devolverá un booleano indicando si el elemento está incluido o no en alguno de los arrays. Como ya sabes, esta función podrá llamar a todas las funciones que necesite.

## Práctica 3.8. Sustitución de espacios

Escribe un programa en JavaScript 6 llamado `~/at/practica03/quitaEspacios.js` que tenga una función que reciba una cadena, y que devuelva esa misma cadena, pero reemplazando los espacios por barras bajas (_). Observa que el nombre del programa es `quitaEspacios.js`, en notación *dromedario*.

## Práctica 3.9. Plain objects

Escribe en un fichero `~/at/practica03/buscaObjeto.js` un programa en JavaScript que:

- Tenga una función que cree y devuelva un *plain object*. Iniciando sus propiedades a partir de los parámetros de esa función, como los ejemplos siguientes:

  ```
  crea_alumno(nombre, apellidos, dni, expediente);
  crea_coche(marca, modelo, matricula);
  crea_asignatura(id, nombre, titulacion, cuatrimestre);
  ```

  Piensa una función similar a estas, pero no idéntica. Basta con una. Observa que el nombre del programa es `buscaObjeto.js`, en *notación dromedario*, con una mayúscula (solo una) separando las palabras. Como en todos los ficheros de esta asignatura, si lo escribes de otra forma, aunque solo cambie una mayúscula, tu práctica será un *no presentado*.

- Tenga otra función que reciba un array de objetos, y haga alguna búsqueda sobre ellos, devolviendo los que cumplan cierta condición. O la lista vacía si ninguno la cumple. Ejemplo:

  - A partir de una lista de objetos alumno, devolver una lista con todos los que tengan el dni 01234 (solo deberia ser uno, pero la función no lo tiene en cuenta).
  - A partir de una lista de coches, devoler todos los de marca *Seat*.
  - A partir de una lista de asignaturas, devolver todas las del primer cuatrimestre.

En estos ejemplos, *01234*, *seat* o *1* serían un parámetro de la función de búsqueda, no un valor constante en esa función.

La función puede devolver una lista de los *plain objects*, enteros, tal cual. O puede devolver un identificador del objeto. Seguramente lo segundo sería preferible en un programa real, puedes hacerlo como desees.

## Práctica 3.10. Comprobación del número de parámetros

En JavaScript, como sabes, si una función recibe un número de parámetros diferente al previsto, no se dispara ninguna excepción. Para estar seguros de que la función se invoca con el número correcto de parámetros, es necesario que el programador se ocupe de ello.

Es lo que harás en este ejercicio. Escribe en un fichero `~/at/practica03/compruebaParametros.js` un programa que tenga una función llamada *compruebaParametros* (o *comprueba_parametros*, según la notación que prefieras). Recibirá dos argumentos: el número de parámetros que tiene una función en su declaración y el número de parámetros con el que se invoca. Si ambos no coinciden, generará una excepción (un objeto). Escribe también unas cuantas funciones para probar esta función.

Recuerda que la propiedad *length* del objeto predefinido *arguments* permite conocer el número de parámetros con el que se ha invocado a una función.

Ejemplo de uso: sea la función *f*, con los parámetros *x* e *y*. La función *f* lo primero que hará es llamar a *compruebaParametos* (o *comprueba_parametros*) para asegurarse de que realmente ha sido invocada con 2 parámetros.

## Práctica 3.11. Busqueda de caracteres

Escribe en un fichero `~/at/practica03/buscaCadena.js` un programa en JavaScript que tenga una función que:

- Reciba dos parámetros, que serán cadenas. La primera cadena de longitud 1, la segunda de longitud arbitraria (lo que incluye la longitud 1)

- Eleve las excepciones adecuadas si los parámetros no cumplen lo especificando en el punto anterior (no son dos, no son cadenas, la primera no es de longitud 1). Utiliza la función que has escrito en la práctica anterior.
- Devuelve el número de veces que la primera cadena aparece en la segunda. Ejemplos
  - Para los parámetros `"x", "xyz"` devolvería 1
  - Para los parámetros `"1", "xyz"` devolvería 0
  - Para los parámetros `"a", "ababA"` devolvería 2

# Práctica 3.12. Validación de contraseña

Escribe en un fichero `~/at/practica03/valida.js` un programa JavaScript que tenga una función que indique si una contraseña tiene la fortaleza requerida

Recibirá los siguientes parámetros:

1. La contraseña.
2. La longitud mínima que deberá tener la contraseña para que se considere válida.
3. El número mínimo de minúsculas exigidas. Puede ser 0. Por *letra minúscula* entenderemos todas las letras inglesas minúsculas, mas la eñe minúscula. Las letras con tilde o con diéresis no las consideramos letras a estos efectos.

   Ejemplo: si el número mínimo de minúsculas exigido es 2, la cadena *ESPAña* sería válida. Pero *CAñóN* no, porque la *ó* con tilde no se considera en este caso correcta (podría ser un carácter especial si así lo especificara el usuario de la librería)
4. El número mínimo de mayúsculas. Puede ser 0. Como en el apartado anterior, consideramos letra mayúscula todas las letras inglesas, más la eñe mayúscula. Y no consideramos letra las letras con tilde o con diéreses.

   Del mismo modo, si se exigieran por ejemplo al menos 2 mayúsculas, *espAÑA* sería correcto tendría 3 mayúsculas. Pero *ÑandÚ* no, porque la *Ú* no la contaríamos como letra mayúscula.
5. El número mínimo de dígitos exigidos. (Puede ser 0)
6. El número mínimo de caracteres especiales exigido. (Puede ser 0)
7. Una cadena que contendrá todos los caracteres que serán considerados especiales. P.e.

   ```
   ,.-{}[]!"·$%&/()=?¿¡'
   ```

   Si la contraseña contiene algún carácter especial no incluido en esta lista, se considerará incorrecta.

   Si esta cadena que especifica lo que se considera especial contiene números o letras ordinarias, puedes hacer lo que prefieras: comprobarlo o no, dar un error, ignorarlo, considerarlo un carácter especial, etc

Devolverá

- La cadena *ok*, en minúscula, si todos los parámetros son correctos, todos los caracteres de la contraseña son correctos y la contraseña tiene la fortaleza requerida.
- En otro caso, una cadena de error. La cadena empezará por *Error:* (primera con mayúscula), y a continuación describirá el problema: la contraseña no tiene suficientes números, no tiene suficientes minúsculas, tiene un carácter especial no permitido, etc

Observaciones adicionales:

- La función debe detectar si el programador que la usa se ha equivocado y alguno de los 7 parámetros es incorrecto
  - Por no ser del tipo adecuado. Por ejemplo que el programador que usa la librería exija una contraseña con *jkrr* minúsculas. (Esto no tiene sentido, *jkrr* es una cadena, no un número)
  - Por no ser un valor correcto. Por ejemplo negativo, o indefinido.

  En estos casos, el programa levantará una excepción.
- Observa que en este programa, como en cualquier otro que use las excepciones de forma razonable:
  - Cuando el usuario se equivoca, cosa bastante frecuente y no especialmente problemática, esta situación se describe en un parámetro de salida. No salta ninguna excepción.
  - Cuando quien se equivoca es el programador que usa la función, sí es un problema severo, que no debería suceder, y que debe provocar una excepción.
- Para el punto 7 tendrás que hacer una función que cuente cuántas veces aparece en la contraseña cada uno de los caracteres de una cadena. Esta misma función te servirá para contar números, usando como cadena 1234567890. También para contar minúsculas, usando como cadena *abcdefg...* y mayúsculas con *ABCDEFG...* Reutiliza el código de la práctica anterior.

## Práctica 3.13. Paradoja del cumpleaños

Sea un conjunto aleatorio de personas. ¿Cuál es la probabilidad de que al menos dos de ellas cumplan años el mismo día? La llamada *paradoja del cumpleaños* (`https://es.wikipedia.org/wiki/Paradoja_del_cumplea%C3%B1os`) nos dice que si hay al menos 23 personas, la probabilidad es superior al 50 %. Si hay al menos 41, la probabilidad es superior al 90 % y si son 57, superior al 99 %.

Escribe un programa llamado `~/at/practica03/paradoja.js` que tenga:

- Una función que tome como entrada una lista de fechas de cumpleaños, y añada a esa lista una nueva fecha aletoria. Puedes hacerlo de forma destructiva o no destructiva, como prefieras con tal de que funcione correctamente.
- Una función que a partir de una lista de fechas de cumpleaños, indique cuántas personas tienen el mismo cumpleaños.
- Una función que vaya generando estas listas, de tamaño incremental. Esto es: dos personas, tres, cuatro.... Y que se detenga cuando haya dos con el mismo cumpleaños.

Observaciones:

- Naturalmente, puedes usar las funciones adicionales que veas conveniente.
- Ignora los años bisiestos.
- Aunque las personas indicamos nuestro cumpleaños en formato mes y dia, para este ejercicio es mucho más cómodo que, internamente, las funciones siempre manejen una fecha como un entero entre 1 y 365. Ejemplo: el 1 sería el 1 de enero, el 32 el 1 de febrero y el 365, el 31 de diciembre.
- Escribe ejemplos que muestren el funcionamiento de tu código. Pero usando el formato mes y dia. Por tanto, necesitarás una función que apartir de una fecha como dia del año, devuelva mes y día. Esto es algo muy común en cualquier programa: usar cierto formato para el proceso interno, y un formato distinto para presentarlo a las personas. Esta conversión debe hacerse en el *último momento*, esto es, inmediatamente antes de presentase al usuario. Esta función puedes programarla a partir de lo visto en clase, o simplemente *googlear* un poco, hay muchos ejemplos en internet. En este caso, asegúrate de comprender bien el código que copias.

## Revisión de los nombres de los ficheros

Ejecuta `~mortuno/revisa practicas at` para comprobar que los nombres de los programas son los correctos.

**Prácticas con el DOM.**
Aplicaciones Telemáticas, 2023-2024
Grado en Ingeniería Telemática
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

## Instrucciones generales

1. Crea un directorio `~/at/practica04` para los ejercicios de esta práctica
2. Ahora escribirás ficheros .html que utilizarán ficheros .js de la práctica 3.
   - En el fichero .html de cada práctica 4, añade una referencia al fichero .js correspondiente indicando su trayecto relativo. Ejemplo `<script src="../practica03/blabla.js"><script>`

## Práctica 4.1. Manejadores

Escribe una página web con Bootstrap 5 y código JavaScript contemporáneo en los ficheros
`~/at/practica04/botones.html`
`~/at/practica04/botones.js` para probar los manejadores sobre botones y la escritura de contenido en elementos.

Ponle al menos tres botones, y haz que al pulsar cada uno, apareza un texto cualquiera. P.e. botones con el texto *Café solo*, *Café con leche*, *Chocolate*. Al pulsarlos, en algún lugar de la pantalla aparecerá diferentes textos. P.e. *Seleccionado café con leche*.

No uses este ejemplo, piensa cualquier otra cosa.

## Práctica 4.2. Escritura y lectura de datos HTML-JavaScript

Escribe los ficheros ficheros `~/at/practica04/unidades.html` `~/at/practica04/unidades.js` en HTML 5 y JavaScript contemporáneo que contengan el interfaz de usuario del programa que escribiste en la práctica 3.6.

La distancia se supone que la indicará el servidor, por tanto, será un valor escrito en el HTML, tu programa no podrá modificarlo. Para tus pruebas, cambia este valor a mano.

Cumpliendo lo anteriormente indicado, tienes libertad para organizar el HTML como prefieras.

## Práctica 4.3. Tabla aleatoria

Escribe los ficheros ficheros `~/at/practica04/tablaRandom.html` `~/at/practica04/tablaRandom.js` en HTML 5 y JavaScript contemporáneo según la siguiente especificación:

Queremos una aplicación web (de momento sin Bootstrap) que, usando número aleatorios, genere una tabla simulando los resultados de extraer al azar una series de valores, por ejemplo

- 7 Tiradas de una máquina tragaperras
- 10 Cartas al azar de la baraja francesa (o un subconjunto)
- 4 Cartas al azar de alguna baraja de tarot (o un subconjunto)
- 5 Fichas de dominó
- 4 Piezas de ajedrez
- Algún otro ejemplo similar. Pero que no sea la baraja española.

Por *ejemplo similar* entendemos *algo* que dependa de al menos dos valores aleatorios, como carta y palo, número superior y numero inferior (en el dominó), etc. La extracción de p.e. una bola de bingo no sería válido porque es un único número.

De momento, tu aplicación simplemente escribirá el valor. Por ejemplo Reina picas". Pero en una práctica posterior, tendrás que generar una imagen. En este ejemplo, una imagen de una reina de picas. O una "qƶ unas picas.

Por tanto, busca ahora las imágenes que usarás después. Sugerencia: usa palabras clave como *free tarot cliparts*, *free slot machine clip art*, *free playing card icons*, *playing card images*.

De momento cada tirada es independiente de la anterior, esto es, el mismo valor puede repetirse. Cada tirada debe añadir una nueva fila a la tabla.

Ya sabes que primero tienes que escribir la lógica de negocio en node.js, y luego insertarla en el HTML.

## Práctica 4.4. Tabla aleatoria, con imágenes, sin repetición

Guarda una copia de tu práctica anterior (por si *rompes algo*) y ahora modifícala de forma que:

1. Tenga un botón que permita elegir entre resultados en modo texto (como hasta ahora) o resultados gráficos. Por omisión, la aplicación mostrará los gráficos.

2. La aplicación no lanzará varios resultados (7, 5, 10, ...) sino que lanzará 1 resultado cada vez que el usuario pulse un botón. Esto es, cada vez que el usuario pulse un botón, se agregará una nueva fila (o columna) a la tabla.

3. Tenga un botón que permite elegir entre resultados *con repetición* o *sin repetición*. Por omisión, será sin repetición. Esto es, si sale por ejemplo el 3 de bastos, no podrá volver a salir.

   La aplicación debe controlar que ya han salido todos los valores posibles y advertirá que no se puede seguir tirando (implementa esto como prefieras)

4. La aplicación tendrá un botón llamado *nuevo mazo* o similar que olvide los valores que ya han salido para permitir volver a empezar (sin borrar lo ya escrito)

5. Recuerda que cada tirada debe añadir filas a la tabla, sin borrar los valores anteriores.

Sugerencia: para evitar repetidos y controlar si quedan valores por aparecer, genera una lista con todos los valores posibles (por ejemplo todas las cartas del mazo), y vete extrayendo elementos aleatorios de esta lista.

## Práctica 4.5. Mostrar y ocultar controles

En este ejercicio vas a practicas el mostrar y ocultar controles usando el DOM.

Escribe los ficheros `~/at/practica04/ocultar.html` `~/at/practica04/ocultar.js` según la siguiente especificación

- Elige uno de tus ficheros HTML de las prácticas 2.9 o 2.10. Cópia el código en este fichero y modifícalo.

- Añádele algún elemento (botón, tarjeta, etc) si lo crees necesario.

- Añade botones (entre uno y tres) para que el usuario pueda mostrar u ocultar distintas partes del interfaz de usuario. Puedes organizarlo por complejidad (p.e. modo simplificado, modo normal, modo avanzado). También puedes organizarlo por funcionalidad, esto es, mostrar u ocultar los controles que se ocupen de tareas que estén relacionadas.

## Práctica 4.6. Modificación de un botón

Escribe los ficheros `~/at/practica04/cambiaBoton.html` y `~/at/practica04/cambiaBoton.js` para hacer algo similar a quita_pon_02.html. Esto es, tendrá un botón para ocultar o mostrar un elemento html. Pero tendrá una funcionalidad nueva: el botón cambiará su texto según corresponda. Por ejemplo indicará *Ver foto* o *Ocultar foto* según corresponda.

Hay muchas formas de conseguir esto.

- Puedes usar una variable booleana que vaya cambiando de estado. Cada vez que pulses el botón, invierte el estado (con el operador de negación). Según el estado, muestras la foto o no e invocas a *text()* con el valor que corresponda.
- Podrías leer el valor del texto del botón. Pero esto no sería un buen diseño, estarías mezclando lógica de negocio con interface de usuario. Piensa por ejemplo en qué pasaría si quisieras portar este programa a un entorno gráfico diferente, o simplemente traducirlo.

## Práctica 4.7. Selección de figuras

Guarda una copia de tu práctica 4.4 (por si *rompes algo*) y ahora modifícala de forma que use Bootstrap y además:

- El usuario pueda *marcar* o *seleccionar* o *voltear* las imágenes haciendo *clic* sobre ellas. Cuando una imagen reciba un clic, su aspecto cambiará. Por ejemplo, si es una baraja, cambia para mostrar el reverso de la carta. O un comodín, o un cuadro negro o cualquier otra imagen que prefieras. Pero siempre la misma imagen en todos los casos.
- Cuando el usuario haga clic sobre una imagen *marcada*, la imagen se *des-seleccionará* y volverá a su aspecto normal.
- Esta selección solo funciona en modo *imagen*. Si el usuario hace clic sobre la figura en modo texto, no sucederá nada.
- La aplicación guardará en la estructura de datos que prefieras la relación de elementos que están marcados. Esta estructura la mostrarás en la consola, como una traza. Al *marcar* una figura, la añadirás a la estructura. Al *desmarcar* una figura, la quitarás

Práctica opcional: imagen de selección individualizada.

- En el caso básico, todas las figuras pasan a tener el mismo aspecto cuando son seleccionadas. Por ejemplo, con el reverso de una baraja. En esta práctica opcional, cuando la imagen sea seleccionada, se verá una versión modificada de esa imagen en particular. Por ejemplo, en blanco y negro, o en negativo, o con colores más pálidos, o más intensos, etc
- Para ello, busca una librería de procesamiento de imágenes (del lenguaje de programación que prefieras) y prepara una versión *modificada* de cada una de tus figuras. (podría hacerse a mano con un programa de retoque de imágenes como por ejemplo Gimp, pero no lo hagas, busca un método automático)
- Por lo demás, la práctica funcionará igual: al marcar la figura se verá modificada y aparecerá en la estructura de atos. Al volver a hacer clic, volverá a su aspecto normal y saldrá de la estructura

## Práctica 4.8. Interface de la validación de contraseña

Escribe en los ficheros `~/at/practica04/valida.html` y `~/at/practica04/valida.js` el código necesario para ejecutar dentro del navagador tu práctica de validación de contraseñas. No repitas nada, haz llamadas al javascript de tu prática 3. Usa Bootstrap 5.

## Revisión de los nombres de los ficheros

Ejecuta `~mortuno/revisa practicas at` para comprobar que los nombres de los programas son los correctos.

**Prácticas de web programable**
Aplicaciones Telemáticas, 2023-2024
Grado en Ingeniería Telemática
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

# Práctica 5.1. Posición

Escribe un programa en los ficheros `~/at/practica05/posicion.html` `~/at/practica05/posicion.js` según la siguiente especificación:

- Será similar a `http://ortuno.es/localStorage.html`: preguntará al usuario su nombre y lo guardará en *local storage*. Pero almacenará, además, las coordenadas del usuario y la fecha en la que ejecutó el programa.
- El programa no usará *alert* ni *prompt* para mostrar o pedir información, sino que escribirá los mensajes en la página (sin ventanas emergentes) y consultará el nombre desde un formulario.
- Cuando el programa tenga constancia de una visita anterior del usuario, le mostrará mensajes similar a estos, según proceda o según prefieras:
  - *Hola Juan, tu última visita fue en Tue May 03 2022 12:32:20, tus coordenadas eran (40.2833, -3.8215)*
  - *Hola Ana, no tengo constancia de visitas previas*
  - *Hola Pedro, tu última visita fue el martes 3 de mayo a las 12:32, tus coordenadas eran las mismas que ahora: (40.2833, -3.8215)*

# Práctica 5.2. Geolocalización y mapas

Escribe una página con el nombre `~/at/practica06/mapa.html` que presente un mapa de OpenStreetMaps con la posición actual del usuario.

# Práctica 5.3. ¿Dónde está mi coche?

Escribe una aplicación web en los ficheros `~/at/practica06/dondeEsta.html` y `~/at/practica06/dondeEsta.js` con la funcionalidad típica de las aplicaciones del tipo *Where is my car?*
Esto es, servirá para que el usuario pueda registrar dónde aparcó su coche.

- La página tendrá especial utilidad funcionando en un teléfono móvil. Pero gracias a que HTML es una norma universal, puedes desarrollarlo en un ordenador de escritorio o portátil y probarlo luego en el móvil.
- Toda la información se almacenará usando *Web Storage*.
- La página mostrará sobre un mapa:
  - La posición actual del usuario.
  - Si está almacenada, la posición del coche.
- La página tendrá dos botones.
  - Recordar posición.
    Al pulsarlo, la posición almacenada será la posición actual. Si ya había una posición guardada, la borrará.
  - Borrar posición.
    Al pulsarlo, se borrará la posición almacenada. Si no hay ninguna, el botón estará deshabilitado.

## Revisión de los nombres de los ficheros

Ejecuta `~mortuno/revisa practicas at` para comprobar que los nombres de los programas son los correctos.

**Prácticas Cliente Servidor en JavaScript**
Aplicaciones Telemáticas, 2023-2024
Grado en Ingeniería Telemática

Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

---

# Práctica 6.1. Uso de Express

Prueba en tu ordenador el primer ejemplo completo de las transparencias sobre Express.js (*api_rest.js*). No escribas memoria de este apartado.

# Práctica 6.2. Uso de Express(2)

Prueba en tu ordenador el ejemplo de las transparencias para servir ficheros estáticos (*estaticos.02.js*). No escribas memoria de este apartado.

# Práctica 6.3. Diseño de API REST/ROA

Considera la empresa de tu práctica 1.4. Diseña un interface REST/ROA con al menos 4 direcciones para hacer diversas operaciones: dar de alta un usuario, consultar los datos de un producto a partir de su id, reservar una habitacion en cierta fecha, etc.

Configura Express.js en el fichero `~/at/practica05/empresa.js` implementando ese API, pero devolviendo solamente mensajes de prueba. En otras palabras: escribe un script similar al `api_rest.js` de las transparencias, pero con un API diferente.

# Práctica 6.4. Código en el servidor

Crea un fichero `~/at/practica05/servidor_distancias.js` que ofrezca mediante un interface REST/ROA la función principal de `~/at/practica03/convierte.js`

- Copia el código que necesites de tu práctica 3.
- Si el cliente comete algún error con los parametros (el primero no se puede convertir en número o el segundo no es una unidad adecuada) el servidor devolverá un error `400 Bad Request`
- Puedes diseñar la URL como prefieras, pero observa que debería tener dos parámetros.

# Práctica 6.5. Ficheros estáticos

Crea un fichero `~/at/practica05/estaticos.js` que configure Express.js para exportar via web en el puerto 3000 de *localhost* tus ficheros de las prácticas 1,2, 3 y 4.

- Para construir los *path*, usa siempre `path.join()`

# Práctica 6.6. Hola JSON

Prepara un script en node.js con el nombre `~/at/practica05/json.js`. Define un objeto literal con al menos 8 propiedades, probando todos los tipos de valor de JSON. Convierte el objeto en cadena. Convierte la cadena de nuevo en objeto.

## Práctica 6.7. Prueba de cliente REST

Prueba dobla_server y dobla_client en tu ordenador. No es necesario que escribas memoria.

## Práctica 6.8. Modificación de dobla_server y dobla_client

Copia el fichero dobla_server.js de las transparencias en `~/at/practica05/opera_server.js`.
Copia el fichero dobla_client.html de las transparencias en `~/at/practica05/opera_client.html`.
Copia el fichero dobla_client.js de las transparencias en `~/at/practica05/js/opera_client.js`.
Haz los cambios necesarios para que:

1. El cálculo del doble del número sea una función independiente.
2. El directorio raiz sea `~/at/www/site01`
3. No use *funciones flecha* sino funciones con sintaxis tradicional.
4. Si el cliente pasa un argumento que no es un número, el servidor devuelva un error `400 Bad Request`. El cliente percibirá esto y lo mostrará en el HTML de la forma que te parezca conveniente.
5. El botón cuyo texto actual es *enviar un valor* pasará a tener un texto como *doblar un valor*. Y habrá otro botón con el texto *triplicar un valor* completamente análogo al anterior, que enviará un número aleatorio al servidor via REST/ROA y recibirá el número por triplicado.
6. Añade un nuevo botón al interface, llamado *Introducir Errores*. Cuando se pulse, la aplicación estará en un modo especial en el que, *de vez en cuando*, en vez de enviar un número correcto, enviará un valor ilegal. Por ejemplo una cadena.

   Para implementar este *de vez en cuando*, antes del envio del valor, genera un valor aleatorio, y si está dentro del intervalo que veas oportuno, el cliente enviará un valor correcto, y si no, uno falso. Por ejemplo, genera un número del 1 al 5, si sale 1 envía un error, en otro caso, un valor correcto.

   Cuando la aplicación esté en este modo, el botón tendrá el texto *No Introducir Errores*, al pulsarlo, volverá al modo en el que todos los valores que se envían son correctos.

Exámenes

**Aplicaciones Telemáticas**
**Examen de teoría. 17 de Mayo de 2018**
Grado en Ingeniería Telemática. Universidad Rey Juan Carlos

---

## Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara_at`
- Esto dejará en el ordenador el fichero `~/at/TULOGIN/at.teoria.mayo.18.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.
- Encontrarás los pdf de las transparencias en `~/at/transpas`

# Ejercicio 1

La transparencia 15 del tema 8 , refiriéndose a JSONP y a la *same site policy* dice : *Pero la cadena "f(3)"no es un dato. Es un script. Se puede enviar*
Explíca esta afirmación. Explica por qué no se considera un dato, por qué se puede enviar y para qué puede servir este script, si no es un dato.

# Ejercicio 2

En JavaScript, para averiguar la longitud de una cadena escribimos `x.length`, y no `x.length()` ni `length(x)`.

1. ¿Por qué escribimos `x.length` ?
2. ¿Por qué no escribimos `x.length()` ?
3. ¿Por qué no escribimos `length(x)` ?

# Ejercicio 3

Explica el *problema del gorita y el plátano* en la programación orientada a objetos *tradicional*. Explica por qué no se da en la programación orientada a objetos basada en prototipos. Ten en cuenta que si te limitas a copiar el contenido de la transparencia, tu respuesta no tendrá ningún valor.

**Desarrollo de Aplicaciones Telemáticas**
**Examen de teoría. 8 de Mayo de 2018**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Universidad Rey Juan Carlos

---

## Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara_dat`
- Esto dejará en el ordenador el fichero `~/dat/TULOGIN/dat.teoria.mayo.18.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.
- Encontrarás los pdf de las transparencias en `~/dat/transpas`

# Ejercicio 1

La transparencia 15 del tema 8 , refiriéndose a JSONP y a la *same site policy* dice :
*Se entiende que esto no es especialmente peligroso. Bancofuenla podría enviar una contraseña a una página web, como un dato, esperando que la lea el usuario. Pero en un script nunca debería haber información sensible*
Explíca esto.

## Respuesta

La *same site policy* obliga a que un script incluido en un HTML solo pueda acceder a datos del mismo sitio web donde se descargó el HTML, no de un sitio distinto. Esta política se aplica solo a datos, no a scripts. JSONP es un *truco* que permite que el script acceda a datos de un sitio web diferente. Para ello, los datos se *disfrazan* de funciones. Aunque los datos se *maquillan* siguen siendo datos. Pero en general esto no se considera peligroso, porque si el servidor *participa en el truco*, si acepta enviar los datos de esta forma peculiar, es porque sabe que los va a manejar un script de un sitio distinto, del que no tiene por qué fiarse.

Por tanto, estos datos *disfrazados de función* nunca serán sensibles. Los datos sensibles solo se enviarán como dato *normal* y por tanto estarán protegidos por la *same site policy*.

# Ejercicio 2

En JavaScript, para averiguar la longitud de una cadena escribimos `x.length`, y no `x.length()` ni `length(x)`.

¿Por qué?

### Respuesta

Porque en JavaScript la longitud de una cadena es un atributo. `x.length()` sería un método, y `length(x)` sería una función.

## Ejercicio 3

Explica el problema del *yo-yo* en la programación orientada a objetos *tradicional*. Explica por qué no se da en la programación orientada a objetos basada en prototipos. Ten en cuenta que si te limitas a copiar el contenido de la transparencia, tu respuesta no tendrá ningún valor.

**Aplicaciones Telemáticas**
**Examen de teoría. 15 de Junio de 2018**
Grado en Ingeniería Telemática. Universidad Rey Juan Carlos

---

### Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara_at`
- Esto dejará en el ordenador el fichero `~/at/TULOGIN/at.teoria.junio.18.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.
- Encontrarás los pdf de las transparencias en `~/at/transpas`

# Ejercicio 1

En las páginas web antiguas, el concepto *viewport* era distinto al actual, más sencillo. Explica qué problemas se daban entonces por esta causa. Ten en cuenta que si te limitas a copiar el contenido de la transparencia, tu respuesta no tendrá ningún valor.

# Ejercicio 2

La transparencia 13 del tema javascript-1 dice lo siguiente:

*Se acerca más a un lenguaje interpretado: el motor necesita estar siempre presente, la compilación se hace en cada ejecución y siempre se distribuye el fuente y solo el fuente*

Explica esto.

# Ejercicio 3

Explica el *problema del gorila y el plátano* en la programación orientada a objetos *tradicional*. Explica por qué no se da en la programación orientada a objetos basada en prototipos. Ten en cuenta que si te limitas a copiar el contenido de la transparencia, tu respuesta no tendrá ningún valor.

**Desarrollo de Aplicaciones Telemáticas**
**Examen de teoría. 13 de Junio de 2018**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Universidad Rey Juan Carlos

---

### Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara_dat`
- Esto dejará en el ordenador el fichero `~/dat/TULOGIN/dat.teoria.junio.18.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.
- Encontrarás los pdf de las transparencias en `~/dat/transpas`

# Ejercicio 1

En las páginas web antiguas, el concepto *viewport* era distinto al actual, más sencillo. Explica qué problemas se daban entonces por esta causa. Ten en cuenta que si te limitas a copiar el contenido de la transparencia, tu respuesta no tendrá ningún valor.

# Ejercicio 2

La transparencia 13 del tema javascript-1 dice lo siguiente:

*Se acerca más a un lenguaje interpretado: el motor necesita estar siempre presente, la compilación se hace en cada ejecución y siempre se distribuye el fuente y solo el fuente*

Explica esto.

# Ejercicio 3

Explica el *problema del gorita y el plátano* en la programación orientada a objetos *tradicional*. Explica por qué no se da en la programación orientada a objetos basada en prototipos. Ten en cuenta que si te limitas a copiar el contenido de la transparencia, tu respuesta no tendrá ningún valor.

**Aplicaciones telemáticas**
**Examen de teoría. 9 de mayo de 2019**
Grado en ingeniería telemática. Universidad Rey Juan Carlos

---

### Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara` y contesta sus preguntas.
- Esto dejará en el ordenador el fichero `~/at/TULOGIN/teoria.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (3.3 puntos)

Este fragmento de HTML 4 no es válido en HTML 5. Explica por qué.

```
<p><font size="4" color="red" face="monospace">Hola, mundo</font>
```

Si sabes escribir una versión equivalente correcta en HTML 5, hazlo. Pero lo más importante es que describas por qué esto ya no es correcto y qué técnica se usa para escribir algo equivalente en HTML 5.

# Respuesta

En este ejemplo, los atributos del texto se aplican directamente a un párrafo. En HTML 4 esto era posible, aunque se consideraba una mala técnica. El problema es que si queremos cambiar alguna de estas propiedades, tendríamos que retocar uno a uno todos los párrafos. Ya en HTML 4 se consideraba mucho más adecuado usar estilos CSS. De esa forma, asignamos a los párrafos el estilo, y si luego queremos modificar alguna propiedad del estilo, basta modifica la definición del estilo en un solo sitio, no en todos y cada uno de los párrafos. En HTML 5 ya no es algo desaconsejado, sino directamente incorrecto.

Por otro lado, el párrafo con etiqueta de apertura pero sin etiqueta de cierre tienen la misma consideración tanto en HTML 4 como en HTML 5: es correcto, pero siempre es preferible el cierre explícito. Este es un ejemplo de página similar en HTML 5 (añadiendo el cierre explícito, lo que no es imprescindible):

```
<!DOCTYPE html>
<html lang="es-ES">
<head>
  <meta charset="utf-8">
  <title>Ejemplo de estilos con CSS</title>
  <style>
    .clase01 { color: red;  font-family : courier;   font-size: large  }
  </style>
</head>

<body>
  <p class="clase01">Hola, mundo.</p>
</body>
</html>
```

# Ejercicio 2 (3.4 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué.

1. div.ayuda
2. div .ayuda
3. div #ayuda
4. div ayuda

# Respuesta

1. Correcto. Selecciona los elementos *div* que sean de clase *ayuda*.
2. Correcto. Selecciona los elementos de clase *ayuda* que esté contenidos dentro un elemento *div*. (Contenidos directamente como *hijos* o como *nietos*, *bisnietos*, etc).
3. Correcto. Selecciona el elemento con el identificador *ayuda*, si está contenido dentro de un elemento *div*.
4. Incorrecto. Sería los elementos *ayuda* que estén dentro de elementos *div*, pero en HTML no existen los elementos *ayuda* (pueden existir elementos con esta clase o con este identificador, pero no es un tipo de elemento).

# Ejercicio 3 (3.3 puntos)

¿Qué significa *fuertemente acoplado*? ¿Qué significa *débilmente acoplado*? ¿Cuál es la relación entre estos conceptos y la programación orientada a objetos basada en clases y la programación orientada a objetos basada en prototipos?

# Respuesta

*Fuertemente acoplado* y *débilmente acoplado* son conceptos que se aplican a cualquier sistema modular. Si los módulos necesitan mucha información unos de otros para poder ensamblarse, decimos que son *fuertemente acoplados.* Si necesitan poca información unos de otros, son *débilmente acoplados.* Y por supuesto, hay todo tipo de valores posibles intermedios. Así podríamos decir que los componentes de un ordenador tipo torre tienen un acoplamiento mucho más débil que los de un portátil: en los primeros es fácil por ejemplo quitar una tarjeta de vídeo y poner otra, en los segundos, no.

La programación orientada a objetos basada en clases tiene un acoplamiento más fuerte que la basada en prototipos, porque un objeto instanciado a partir de una clase normalmente hereda todas sus propiedades, y las de sus clases padre, abuelo, etc. Mientras que con las técnicas de programación orientada a objetos basadas en prototipos, los objetos se crean a partir de objetos, controlando qué elementos se reaprovechan y cuáles no.

**Desarrollo de Aplicaciones Telemáticas**
**Examen de teoría. 16 de Mayo de 2019**
Grado en Ingeniería Telemática. Universidad Rey Juan Carlos

---

### Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara`.
- Esto dejará en el ordenador el fichero `~/dat/TULOGIN/teoria.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1

En el siguiente fragmento de código jQuery ¿qué es *click*? ¿Un evento? ¿Un método? ¿Una función? ¿Un manejador? ¿Todas estas cosas a la vez? ¿Ninguna de estas cosas? ¿Alguna de estas cosas?

```
$("#boton01").click(function() {
    [...]
});
```

# Respuesta

Es un método. (Y por tanto, una función, un método es una función que pertenece a una clase). Este método click se llama igual que el evento click que recibe cualquier elemento HTML cuando el usuario pulsa sobre él el botón principal del ratón/touchpad. Es el convenio de jQuery: para cada evento, define un método con el mismo nombre que el evento. Este método recibe una función, típicamente anónima, que es el manejador del evento.

Resumiendo: el método click recibe como argumento el manejador para el evento click, click es un método que se llama igual que un evento.

# Ejercicio 2

De los siguientes selectores css, indica si son correctos o no. Si son correctos, indica su significado. Si no son correctos, indica por qué.

1. p.resumen
2. p .resumen
3. p #resumen
4. p resumen

# Respuesta

1. Correcto. Selecciona los elementos *p* (párrafos) que sean de clase *resumen*.

2. Correcto. Selecciona los elementos de clase *resumen* que esté contenidos dentro un elemento *p*. (Contenidos directamente como *hijos* o como *nietos*, *bisnietos*, etc).

3. Correcto. Selecciona el elemento con el identificador *resumen*, si está contenido dentro de un elemento *p*.

4. Incorrecto. Sería los elementos *resumen* que estén dentro de elementos *p*, pero en HTML no existen los elementos *resumen* (pueden existir elementos con esta clase o con este identificador, pero no es un tipo de elemento).

# Ejercicio 3

¿Qué significa *fuertemente acoplado*? ¿Qué significa *débilmente acoplado*? ¿Cuál es la relación entre estos conceptos y la programación orientada a objetos basada en clases y la programación orientada a objetos basada en prototipos?

# Respuesta

*Fuertemente acoplado* y *débilmente acoplado* son conceptos que se aplican al diseño del software y también a cualquier sistema modular. Si los módulos necesitan mucha información unos de otros para poder ensamblarse, decimos que son *fuertemente acoplados.* Si necesitan poca información unos de otros, son *débilmente acoplados.* Y por supuesto, hay todo tipo de valores posibles intermedios. Así podríamos decir que los componentes de un ordenador tipo torre tienen un acoplamiento mucho más débil que los de un portátil: en los primeros es fácil por ejemplo quitar una tarjeta de vídeo y poner otra, en los segundos, no.

La programación orientada a objetos basada en clases tiene un acoplamiento más fuerte que la basada en prototipos, porque un objeto instanciado a partir de una clase normalmente hereda todas sus propiedades, y las de sus clases padre, abuelo, etc. Mientras que con las técnicas de programación orientada a objetos basadas en prototipos, los objetos se crean a partir de objetos, controlando qué elementos se reaprovechan y cuáles no.

**Aplicaciones telemáticas**
**Examen de teoría. 20 de junio de 2019**
Grado en ingeniería telemática. Universidad Rey Juan Carlos

---

## Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara` y contesta sus preguntas.
- Esto dejará en el ordenador el fichero `~/at/TULOGIN/teoria.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (3.3 puntos)

Explica qué hace el siguiente código jQuery

```
$("#j23").on('change keyup paste mouseup', function() {
     blablabla();
});
```

## Respuesta

- El signo de dólar es un alias de la función jQuery. Todo el código en jQuery comienza por una llamada a esta única función, que a su vez recibe otras funciones como argumentos.
- `("#j23")` es el selector. En la sintaxis de jQuery, indica sobre qué elementos se realizará la acción. La almohadilla significa *identificador*, en este caso el identificador *j23*. Por tanto estamos seleccionando el elemento HTML con identificador *j23*.
- `on('change keyup paste mouseup', function() { ...})` es un método que asigna un manejador a una serie de eventos. El método *on* recibe:
  - Un primer argumento que es una cadena que contiene una lista de nombres de eventos, separados por espacios.
  - Un segundo argumento con la función a ejecutar cuando el elemento seleccionado reciba esos eventos.

  Los eventos son: cambio en el elemento, pulsar una tecla, pegar desde el portapapeles y soltar el botón del ratón tras haberlo pulsado.

  Por tanto, estas líneas hacen que se dispare la función cuyo código es *blablabla()* cuando el elemento html con indentificador *j23* cambia, se pulsa sobre él, se pega algo o se hace clic sobre él.

# Ejercicio 2 (3.3 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué.

1) .oferta.precio

2) oferta precio

3) #oferta#precio

4) oferta #precio

## Respuesta

1) Correcto. Selecciona los elementos pertenecientes simultáneamente a las clases *oferta* y *precio*

2) Incorrecto. Sería un elemento HTML con etiqueta *precio* descendiente de un elemento HTML de etiqueta *oferta*. Pero en HTML no existen estas etiquetas.

3) Incorrecto. Seria un elemento HTML con el identificador *oferta* y simultáneamente con el indicador *precio*. Pero en HTML un elemento no puede tener más de un identificador.

4) Incorrecto. Seleccionaría el elemento de identificador *precio*, descendiente de un elemento HTML con etiqueta *oferta*. Pero esta etiqueta no existe en HTML.

# Ejercicio 3 (3.4 puntos)

¿Por qué JavaScript 5 no se pueden definir clases? ¿Por qué JavaScript 6 sí se puede?

## Respuesta

Desde su origen, JavaScript usa el paradigma de programación orientada a objetos basada en prototipos, donde no se definen clases sino que los objetos se construyen a partir de otros objetos.

Esto es así hasta la versión 5, pero la versión actual de JavaScript, la 6, (del año 2015) introduce muchos cambios. Uno de ellos es que, manteniendo la programación orientada a prototipos, añade una sintaxis que permite una programación orientada a objetos tradicional, basada en clases.

**Desarrollo de Aplicaciones Telemáticas**
**Examen de teoría. 18 de Junio de 2019**
Grado en Ingeniería Telemática. Universidad Rey Juan Carlos

---

### Instrucciones:

- Entra en el puesto del laboratorio con el nombre de usuario *examen* y la contraseña *examen*.
- Ejecuta el script `prepara`.
- Esto dejará en el ordenador el fichero `~/dat/TULOGIN/teoria.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (2.5 puntos)

Este fragmento de HTML 4 no es válido en HTML 5. Explica por qué.

```
<p><font size="4" color="red" face="monospace">Hola, mundo</font>
```

Si sabes escribir una versión equivalente correcta en HTML 5, hazlo. Pero lo más importante es que describas por qué esto ya no es correcto y qué técnica se usa para escribir algo equivalente en HTML 5.

# Ejercicio 2 (2.5 puntos)

De los siguientes selectores css, indica si son correctos o no. Si son correctos, indica su significado. Si no son correctos, indica por qué.

1) destacado informacion
2) .destacado.informacion
3) destacado.informacion
4) #destacado#informacion

# Ejercicio 3 (2.5 puntos)

En la asignatura hemos dicho que *Los datos de OpenStreetMap son libres y gratuitos. Pero el servidor de tiles no lo es.*

Explica con tus propias palabras el significado de esta frase. Incluye una breve descripción de qué es OpenStreetMap, cuáles son sus datos, que significa *libre y gratuito*, que es un *tile* y qué es un *servidor de tiles*.

# Ejercicio 4 (2.5 puntos)

Explica brevemente en qué consiste el *problema del código yo-yo*.

**Desarrollo de Aplicaciones telemáticas**
**Examen parcial, prueba de teoría. 14 de abril de 2021**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

## Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/dat.abril.21/parcial.TULOGIN.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicios 1, 2 y 3 (3 puntos)

A continuación se presentan tres afirmaciones. Para cada una de ellas, indica claramente si es verdadera o falsa, y además:

- Si es verdadera, explícala un poco más.
- Si es falsa, indica **todo** lo que sea erróneo.

1. CSS se diseña principalmente como respuesta a la aparición de los *smartphones*, para que el mismo contenido se pueda visualizar tanto en un ordenador tradicional como en un teléfono móvil. Anteriormente no era necesario, las pantallas de escritorio también tenían tamaños variables, pero con la barra de desplazamiento vertical este problema se solucionaba razonablemente bien.

2. `a, b`

   es un selector CSS que se refiere a los elementos de clase a y de clase b, esto es, los que pertenezcan a ambas clases simultáneamente. También a sus descendiente directos (*hijos*) pero no a los descendientes de sus descendientes (*nietos, bisnietos...*)

3. `img #personal`

   es un selector CSS que se refiere a los elementos cuyo id sea *personal* que además sean imágenes

## Respuesta

1. Falso. CSS sirve especificar los aspectos gráficos del interface de usuario, de forma que en el HTML solo estén los aspetos semánticos del interface [1]. Para ver contenidos en pantallas pequeñas se usa inicialmente el *viewport* virtual y posteriormente los sistemas *responsivos* basados en rejillas, como Bootstrap y similares.

2. Falso. Se referiría a los elementos *a* y a los elementos *b*. Que no existen en HTML. Para que fueran clases, deberían llevar un punto. También se referiría a sus descendientes.

3. Falso. Se referiría al elemento cuyo *id* es *personal* (solo puede haber uno), si está dentro una imagen. Aunque esto es imposible porue las imágenes son de tipo *void* y no tienen contenido. También se referiría a sus descendientes.

[1] No confundir con la lógica de negocio.

**Aplicaciones telemáticas**
**Examen final, prueba de teoría. 4 de junio de 2021**
Grado en ingeniería telemática.
Universidad Rey Juan Carlos

---

### Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/at.junio.21/examen.TULOGIN.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

¿Qué significa *NaN* en JavaScript? ¿Para qué sirve? ¿Cómo se usa? ¿Qué problemas suele dar a los programadores principantes en JavaScript? ¿Cuál es su equivalente en la mayoría de lenguajes de programación?

### Respuesta

*NaN* (*Not a Number*) es un valor de error devuelto por una operación. El programador esperaba un número, pero no es el caso. Bien por tratarse de un operación matemática que no devuelve un número real (como $\sqrt{-1}$)), bien porque no todos los operandos son numéricos cuando deberían serlo.

Suele resultar problemático para programadores no experimentados porque, cuando aparece, probablemente el flujo normal del programa debería interrumpirse para ser tratado como un error, no como un número más (paradójicamente, *NaN* es de tipo numérico). Además, no es posible detectarlo como una expresión como `x===NaN`, es necesario emplear la función *isNaN()*.

Es una peculiariad de JavaScript, en la mayoria de lenguajes lo equivalente sería elevar una excepción. Observaciones:

- *NaN* es muy distinto de *undefined* (variable o parámetro no inicializado) y muy distinto de *null* (objeto sin valor).
- *NaN* no es una función para comprobar si un valor es numérico o no.

# Ejercicio 2 (3 puntos)

Como sabes, hay un tipo programación orientada a objetos basada en clases y otro basado en prototipos. Las clases son fuertemente acopladas, los prototipos presentan un acoplamiento más débil. Explica esto brevemente.

### Respuesta

En POO basada en clases, un objeto hereda atributos y métodos de su clase y de todas las clases de nivel superior en la jerarquía, normalmente todos los atributos y todos los métodos, sean necesarios o no. Esto crea una dependencia muy fuerte de mucho código, potencialmente muy variado. Un cambio en cualquier clase influirá en cualquier objeto que dependa de ella o de una de sus clases descendientes.

En la POO basada en prototipos, no hay clases. Los objetos se crean a partir de otros objetos mediante factorías. Los atributos y métodos que se necesiten se copian, los que no, no. Ningún cambio en el objeto original afecta al objeto copiado.

## Ejercicio 2 (3 puntos)

¿Para qué sirven los *web workers*?

### Respuesta

Se trata de un API de JavaScript que permite ejecutar un hilo de código en segundo plano. Son especialmente útiles para tareas que consuman mucho tiempo, normalmente por ser intensivas en uso de CPU o de comunicaciones, porque así no interfieren a la tarea en primer plano donde está el interfaz de usuario.

**Desarrollo de Aplicaciones telemáticas**
**Examen final, prueba de teoría. 2 de junio de 2021**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

### Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/dat.junio.21/examen.TULOGIN.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

¿Cuál es la diferencia entre `=` , `==` y `===` en JavaScript? ¿Para qué se usa cada uno?

### Respuesta

- El igual simple sirve para asignar un valor a una variable.
- El igual doble es el comparador de igualdad *permisivo* que implementaba inicialmente JavaScript. Su propósito es considerar iguales dos valores aunque su tipo de datos sea distinto. Demostró ser muy problemático, es preferible no usarlo nunca.
- El igual triple, que JavaScript llama *estricto* es el que deberíamos usar siempre, es como el de cualquier otro lenguaje de programación: para que dos valores sean iguales, en condición necesaria (pero no suficiente) que sean del mismo tipo.

# Ejercicio 2 (3 puntos)

Explica brevemente la siguiente afirmación: En JavaScript 6 la programación orientada a objetos basada en herencia es *azúcar sintáctico*.

### Respuesta

La programación orientada a objetos basada en prototipos es el tipo de programación nativo en JavaScript. Pero muchos programadores prefieren la programación orientada a objetos basada en clases, porque está mucho más extendida y porque funciona bien para problemas de complejidad baja o media.

Así que en JavaScript 6 aparece, como opción adicional, una sintaxis similar a la de cualquier POO basada en clases. Es solo un recubrimento externo, una apariencia. Internamente sigue siendo POO basada en prototipos, pero *endulzada* para que sea *más fácil de digerir* por quienes prefieren la programación tradicional.

# Ejercicio 2 (3 puntos)

¿Para qué sirven los *web workers*?

## Respuesta

Se trata de un API de JavaScript que permite ejecutar un hilo de código en segundo plano. Son especialmente útiles para tareas que consuman mucho tiempo, normalmente por ser intensivas en uso de CPU o de comunicaciones, porque así no interfieren a la tarea en primer plano donde está el interfaz de usuario.

**Aplicaciones telemáticas**
**Examen final, prueba de teoría. 20 de mayo de 2022**
Grado en ingeniería telemática.
Universidad Rey Juan Carlos

---

## Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/at.mayo.22/teoria.TULOGIN.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué. (Tal vez te ayude el considerar que esto podría aplicarse en un documento HTML donde tenemos elementos que representan personas y las vacunas que se han puesto)

1) pfizer moderna
2) .pfizer.moderna
3) #moderna#pfizer
4) moderna #pfizer

## Respuesta

1. Incorrecto. Seleccionaría los elementos HTML *moderna* descendientes de elementos *pfizer*, pero esto no tiene sentido, en HTML no existen estos elementos.
2. Correcto. Selecciona los elementos con las clases *pfizer* y *moderna* simultáneamente.
3. Incorrecto. Sería un elementos con los identificadores *pfizer* y *moderna* pero un elemento no puede tener dos identificadores distintos De la misma forma que, por ejemplo, no es legal que un español tenga dos DNI distintos.
4. Incorrecto. Sería un elemento con identificador *pfizer* descendiente del elemento HTML *moderna*. Pero como hemos dicho, *moderna* no es un elemento HTML.

# Ejercicio 2 (2 puntos)

¿En qué se diferencia el *directorio raiz* del *directorio padre*?

## Respuesta

El directorio raiz es el directorio de mayor nivel jerárquico de un sistema de ficheros, el directorio del cual cuelgan todos los demás, el directorio que contiene a todos los demás directorios. Solo hay un directorio raiz.

El directorio padre de un directorio es el directorio que tiene por encima, el directorio del que cuelga cada directorio. Todo directorio tiene un directorio padre, excepto el raíz.

# Ejercicio 3 (2 puntos)

¿Cuál es la diferencia entre *sistema fuertemente acoplado* y *arquitectura frágil*?

## Respuesta

Son conceptos relacionados, pero no exactamente iguales.

- Un sistema fuertemente acoplado es un sistema formado por módulos, donde los módulos tiene una relación muy fuerte unos con otros, muy estrecha. Se comunican muchas cosas entre ellos y no es fácil reemplazar uno por otro.
- Una arquitectura frágil es aquella en la que un cambio en alguno de sus componentes puede provocar que otro componente deje de funcionar.

Los sistemas fuertemente acoplados normalmente provocan arquitecturas frágiles. Aunque no necesariamente: con un sistema con un acoplamiento fuerte, pero bien definido, tal vez se pueda conseguir una arquitectura robusta (lo contrario de frágil). Aunque será complicado, mucho más que si la arquitectura fuera débilmente acoplada.

# Ejercicio 4 (2 puntos)

Como sabes, mediante el API de geolocalización de HTML5, podemos obtenemos los parámetros *latitude, longitude* y *accuracy*. Los dos primeros son obvios, no hace falta decir nada sobre ellos. Respecto al tercer, *accuracy* ¿Qué representa? ¿De qué depende su valor, cuándo será mayor y cuándo menor?

## Respuesta

Representa la precisión de las coordenadas, medida en metros. Si el navegador está en un dispositivo con GPS, la precisión será muy alta. Si el dispositivo no tiene GPS pero tiene conexión via WiFi, la precisión será también bastante buena aunque seguramente no tanto. Si la conexión es por cable, la precisión será la peor de todas.

Suponiendo siempre que el usuario permite el acceso a la geolocalización.

**Desarrollo de Aplicaciones telemáticas**
**Examen final, prueba de teoría. 26 de mayo de 2022**
Grado en ingeniería telemática.
Universidad Rey Juan Carlos

---

### Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/dat.mayo.22/teoria.TULOGIN.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué. (Tal vez te ayude el considerar que esto podría aplicarse en un documento HTML donde tenemos elementos que representan personas y las vacunas que se han puesto)

1) .pfizer.azeneca
2) pfizer azeneca
3) #azeneca#pfizer
4) #pfizer azeneca

### Respuesta

1. Correcto. Selecciona los elementos con las clases *pfizer* y *azeneca* simultáneamente.
2. Incorrecto. Seleccionaría los elementos HTML *azeneca* descendientes de elementos *pfizer*, pero esto no tiene sentido, en HTML no existen estos elementos.
3. Incorrecto. Sería un elementos con los identificadores *pfizer* y *azeneca* pero un elemento no puede tener dos identificadores distintos De la misma forma que, por ejemplo, no es legal que un español tenga dos DNI distintos.
4. Incorrecto. Sería un elemento HTML *azeneca* descendiente del elemento con id *pfizer*. Pero como hemos dicho, *azeneca* no es un elemento HTML.

# Ejercicio 2 (2 puntos)

En *Bootstrap* las columnas (celdas) en ocasiones se muestran en disposición *normal* y en ocasiones en disposición *apilada*. Explica esto.

### Respuesta

En *Bootstrap* y en todas las tecnologías web similares basadas en un *grid*, el contenido se coloca dentro de unos bloques llamados *columnas*, que nosotros preferimos llamar *celdas*. Las celdas se organizan en filas y columnas, las celdas de la misma fila está unas al lado de otras. Esta es la *disposición normal*, cuando la pantalla es lo bastante grande. Si la pantalla es menor, las celdas de una fila ya *no caben* unas al lado de otras, así que se colocan unas encima de otras, en *disposición apilada*.

### Errores frecuentes

- Es un error hablar de *viewport virtual*. También es una técnica para adaptar el contenido a pantallas de tamaño variable, pero anticuada, de segunda generación. El *grid* es una técnica diferente, de tercera generación, que no utilizar *viewport virtual*.
- Es un error decir que las celdas se redimensionan o adaptan su tamaño, o usan más o menos columnas. No lo hacen. Lo que hacer es colocarse en disposición normal (horizontal) o apilada (vertical).

## Ejercicio 3 (2 puntos)

Explica brevemente en qué consiste el *problema del código yo-yo*.

### Respuesta

Es un inconvenient que se da en programación orientada a objetos basada en clases cuando hay jerarquías complejas, esto es: *clases padre*, *clases abuelo*, *clases bisabuelo*, *clases tatarabuelo*, etc.

Cuando un programador trabaja en una clase *nieto*, le afecta el código de todos sus *antepasados*, así que tiene que estar contiuamente subiendo y bajando por la jerarquía de clases, buscando las propiedades o los métodos relevantes en cada caso.

Observación importante: este es un problema de la programación orientada a objetos **solamente cuando las jerarquías son complejas**, lo cual no es especialmente habitual.

## Ejercicio 4 (2 puntos)

En JavaScript ¿Qué es una *promesa*?

### Respuesta

Es una técnica de programación concurrente. Es un objeto que inicialmente no tiene valor, pero que transcurrido cierto tiempo, si todo va bien, acabará teniendo valor. O indicará que se ha producido un error, si ha habido problemas.

Este objeto es devuelto por una función asíncrona, el código que llama a esta función se queda bloqueado esperando a que la promesa se resuelva.

**Desarrollo de Aplicaciones telemáticas**
**Examen final, prueba de teoría. 21 de junio de 2022**
Grado en ingeniería telemática.
Universidad Rey Juan Carlos

---

## Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/dat.junio.22/teoria.TULOGIN.txt`, donde debes escribir tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué. (Tal vez te ayude el considerar que esto podría aplicarse en un documento HTML donde tenemos elementos que representan personas y las vacunas que se han puesto)

1) janssen moderna
2) .janssen.moderna
3) #moderna#janssen
4) moderna #janssen

## Respuesta

1. Incorrecto. Seleccionaría los elementos HTML *moderna* descendientes de elementos *janssen*, pero esto no tiene sentido, en HTML no existen estos elementos.
2. Correcto. Selecciona los elementos con las clases *janssen* y emphmoderna simultáneamente.
3. Incorrecto. Sería un elementos con los identificadores *janssen* y *moderna* pero un elemento no puede tener dos identificadores distintos De la misma forma que, por ejemplo, no es legal que un español tenga dos DNI distintos.
4. Incorrecto. Sería un elemento con identificador *janssen* descendiente del elemento HTML *moderna*. Pero como hemos dicho, *moderna* no es un elemento HTML.

# Ejercicio 2 (2 puntos)

Explica brevemente el concepto *viewport virtual*.

## Respuesta

Es una técnica de segunda generación para adaptar el contenido de una página web a pantallas de distinto tamaño. Desarrollada inicialmente por *Apple* para el *iPhone*, pasó a ser estándar. Consiste en componer la página sobre una pantalla virtual de tamaño convencional, y luego mostrar en la pantalla física del teléfono una parte cada vez, la parte que selecciona el usuario *arrastrando* con el dedo en la pantalla táctil.

En la actualidad se utilizan técnicas de tercera generación, basadas en un *grid*.

# Ejercicio 3 (2 puntos)

Explica brevemente qué es en JavaScript una *función flecha*. Basta con un explicación conceptual, no es necesario ni que describas su sintaxis ni que pongas ejemplos.

## Respuesta

Es una sintaxis de JavaScript y otros lenguajes para declarar funciones de forma compacta, sin necesidad de ponerles nombre ni usar las palabras reservadas *function* o *return*. Se usa para pasar funciones como argumento a otras funciones, especialmente en los manejadores. En JavaScript son muy idiomáticas, pero los programadores de otros lenguajes suelen encontrar esta sintaxis incómoda y algo confusa.

# Ejercicio 4 (2 puntos)

Describe brevemente qué es

- *Web Storage*
- *sessionStorage*
- *localStorage*

Basta con un explicación conceptual, no es necesario ni que describas su sintaxis ni que pongas ejemplos.

## Respuesta

*Web Storage* es un API estándar de HTML5 que permite almacenar información en el navegador web sin usar *cookies*. Usa una estructura de datos de tipo *diccionario*, con pares clave/valor. Se puede usar con dos objetos distintos:

- *sessionStorage*, que guarda información de la sesión, la información se pierde al cerrar el navegador
- *localStorage*, que es persistente. La información se mantiene en el sistema de ficheros del cliente.

**Aplicaciones telemáticas**
**Examen final, prueba de teoría. 15 de mayo de 2023**
Grado en ingeniería telemática.
Universidad Rey Juan Carlos

---

### Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/at.mayo.23/teoria.TULOGIN.txt`, escribe ahí tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué.

1) `tr img`
2) `minimo .p`
3) `p  .minimo`
4) `.minimo, #oscuro`

### Respuesta

1) Correcto. Selecciona los elementos *img* descendentes de filas de una tabla [1].
2) Incorrecto. Seleccionaría los elementos de clase *párrafo* dentro del elemento html *minimo*.
   - Lo primero, usar un nombre de clase que coincida con un elemento de HTML, es legal. Seguramente es mala idea, porque induce a confusión, pero la norma lo permite.
   - Lo segundo es incorrecto: *minimo* no es un elemento HTML.
3) Correcto. Selecciona los elementos de clase *minimo* descendentes de un párrafo.
4) Correcto. Se refiere a los elementos de clase *minimo* y al elemento con identificador *oscuro*. En otras palabras: a los elementos que tengan la clase *minimo* o el identificador *oscuro*[2]

# Ejercicio 2 (3 puntos)

En bootstrap ¿Cuándo y por qué se presentan las columnas en disposición normal? ¿Cuándo y por qué se presentan las columnas en disposición apilada?

[1] Si estuviera al revés, img tr, sería incorrecto porque las imágenes son de tipo void, no pueden tener contenido
[2] Sería un error interpretar esto como 'elementos con clase minimo e identificador oscuro, simultáneamente'

### Respuesta

- Las columnas, que en la asignatura preferimos llamar *celdas*, se presentan en disposición normal, esto es, unas al lado de otras, cuando la pantalla tiene el tamaño *suficiente*. Este es el diseño previsto por el autor de la página cuando no hay restricciones de tamaño.
- Se presentan apiladas, unas encima de otras, cuando la pantalla es *pequeña*. De esta forma cada una de las celdas se puede ver completa, aunque la composición global no resulte tan vistosa.

El tamaño de la pantalla se considera *suficiente* o *pequeña* según lo indicado por el autor de la página con las clases *.col-N, .col-sm-N, .col-md-N, etc.*

## Ejercicio 3 (3 puntos)

¿Por qué tantos programadores se quejan de la programación orientada a objetos de JavaScript? ¿Tienen fundamento esas quejas?

### Respuesta

La programación orientada a objetos (POO) nativa de JavaScript es de un tipo poco habitual, la POO basada en prototipos. Es distinta a la POO basada en clases, que es con diferencia la predominante entre los lenguajes de programación. La POO basada en prototipos se considera generalmente más potente y libre de muchos de los problemas habituales de la POO basada en clases. Pero también tiene sus inconvenientes: está poco extendida y tiene una curva de aprendizaje muy pronunciada.

Cuando JavaScript solo ofrecía POO basada en prototipos, sí resultaba razonable criticar al lenguaje por esto. Actualmente esta crítica es, por lo menos, muy discutible: desde ECMAScript 6 (año 2015), JavaScript ya tiene, además, una sintaxis que permite trabajar de manera muy similar a la de cualquier lenguaje orientado a objetos tradicional.

**Desarrollo de Aplicaciones telemáticas**
**Examen final, prueba de teoría. 11 de mayo de 2023**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

### Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/dat.mayo.23/teoria.TULOGIN.txt`, escribe ahí tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué.

1) p .clasico
2) clasico .p
3) .clasico.reducido
4) td img

### Respuesta

1) Correcto. Selecciona los elementos de clase *clasico* descendentes de un párrafo.
2) Incorrecto. Seleccionaría los elementos de clase párrafo dentro del elemento html *clasico*.
   - Lo primero, usar un nombre de clase que coincida con un elemento de HTML, es legal. Seguramente es mala idea, porque induce a confusión, pero la norma lo permite.
   - Lo segundo es incorrecto: *clasico* no es un elemento HTML.
3) Correcto. Se refiere a los elementos que tengan simultáneamente las clases *clasico* y *reducido*.
4) Correcto. Selecciona los elementos *img* descendentes de celdas de una tabla [1].

# Ejercicio 2 (3 puntos)

Explica la diferencia entre *viewport virtual* y *viewport físico*.

[1] Si estuviera al revés, img td, sería incorrecto porque las imágenes son de tipo void, no pueden tener contenido

### Respuesta

Es normal que una página HTML tenga mucho contenido y que no quepa todo en pantalla. El *viewport* es la parte de la página que ve el usuario en cierto momento, el fragmento de HTML que el navegador construye para la pantalla.

El *viewport virtual* es una técnica web de segunda generación desarrollada para los teléfonos móviles, que tienen una pantalla muy pequeña. En vez de componer la página para la pantalla real, se hace un *render* de la página para un *viewport* virtual, imaginario, de tamaño *normal*. Luego el usuario usa la pantalla táctil del teléfono móvil para desplazar la zona de la página que ve en cada momento, sin que este desplazamiento obligue a recomponer la página.

En este contexto podemos referirnos al *viewport* normal como *viewport* físico, el que coincide con la pantalla real tradicional.

## Ejercicio 3 (3 puntos)

Explica brevemente los conceptos *débilmente acoplado* y *fuertemente acoplado*.

### Respuesta

Son conceptos que se aplican al diseño del software y también a cualquier sistema modular. Si los módulos necesitan mucha información unos de otros para poder ensamblarse, decimos que son *fuertemente acoplados.* Si necesitan poca información unos de otros, son *débilmente acoplados.* Y por supuesto, hay todo tipo de valores posibles intermedios. Así podríamos decir que los componentes de un ordenador tipo torre tienen un acoplamiento mucho más débil que los de un portátil: en los primeros es fácil por ejemplo quitar una tarjeta de vídeo y poner otra, en los segundos, no.

**Desarrollo de Aplicaciones telemáticas**
**Examen final, prueba de teoría. 30 de junio de 2023**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

## Instrucciones:

- Ejecuta en un terminal `~mortuno/prepara`
- Esto dejará en el ordenador el fichero `~/dat.junio.23/teoria.TULOGIN.txt`, escribe ahí tus respuestas. TULOGIN será tu nombre de usuario en el laboratorio.

# Ejercicio 1 (4 puntos)

Indica si los siguientes selectores CSS son correctos o no. Si son correctos, indica su significado. Si no son correctos, explica brevemente por qué.

1.1) castellano h3

1.2) .castellano .h3

1.3) .castellano.simplificado

1.4) img td

Un apartado erróneo o en blanco tendrá una penalización del 50 %. Dos o más apartados erróneos o en blanco, tendrán puntuación nula.

## Respuesta

1) Incorrecto. Serían los elementos h3 descendente de los elementos *castellano*, pero en HTML no existen elementos *castellano*.

2) No es erróneo pero resulta confuso. Se refiere a los elementos de clase *h3*, descendentes de elementos de clase *castellano*. Está permitido un nombre de clase como *h3*, a pesar de que también es un elemento HTML. Es desaconsejable.

3) Correcto. Elementos que pertenezcan a la clase *castellano* y a la clase *simplificado*, ambas simultáneamente.

4) No tiene sentido, se refiere a los elementos *td*, celdas de tabla, contenidos dentro de elementos *img*. Pero los *img* son de tipo *void*, no pueden tener contenido.

# Ejercicio 2 (3 puntos)

2.1) Desde el punto de vista de la comodidad en la lectura de una página web ¿cúal es la diferencia entre la barra de desplazamiento vertical y la barra de desplazamiento horizontal? ¿Por qué?

2.2) Como sabes, en el web hay tres generaciones de técnicas en lo relativo a adaptar una página al tamaño variable de las pantallas. ¿Cómo cambia el uso de la barra de desplazamiento vertical y la horizontal en estas tres generaciones?

### Respuesta

2.1) La barra horizontal es muy incómoda porque hay que moverla continuamente en cada línea. La vertical no es un gran problema porque hay que usarla después de haber leído varias líneas.

2.2) Lás páginas de primera generación usaban la barra horizontal y la vertical. Las de segunda no tenían barras, pero el *viewport virtual* había que arrastralo tanto en horizontal como en vertical. Las de tercera mantienen el desplazamiento vertical, pero no el horizontal, porque el contenido se escribe en columnas adaptadas al tamaño de la pantalla.

## Ejercicio 3 (3 puntos)

En el contexto de JavaScript y la P.O.O. ¿qué es una *factoría de objetos*?

### Respuesta

En la POO convencional, los objetos se crean como instancias de clases. En la POO basada en prototipos no hay clases, los objetos los crean las *factorías de objetos*, que son funciones que crean objetos a partir de otros objetos.

**Aplicationes Telemáticas**
**Examen de prácticas. 17 de Mayo de 2018**
Grado en Ingeniería Telemática, Universidad Rey Juan Carlos

---

# Ejercicio único

- Ejecuta el script `prepara_at`

  Esto creará en tu cuenta el directorio `~/at.mayo.18`

- Copia en el directorio `~/at.mayo.18` los ficheros de la calculadora que hiciste en práctica, esto es, `~/at/practica03/calculadora.js` y `~/at/practica04/calculadora.html`

  Asegúrate de copiar los ficheros originales, no enlaces simbólicos.

- Si lo deseas, puedes usar la versión basada en clases, `~/at/practica04/ccalculadora.js`

  En este caso tendrás mejor nota.

Modifica tu práctica para que haga lo siguiente:

- Queremos que no solo se vea el resultado final del cálculo, sino también el histórico de operaciones, en una pantalla distinta. No es necesario que esta segunda pantalla se actualice continuamente, es suficiente con que se haga al pulsar el botón *igual*.
  Supongamos que el usuario pulsa la siguiente secuencia:

  ```
  1 + 2 * 3 = C 2 + 2 =
  ```

  En el momento en que se pulsa el primer igual, en la segunda pantalla se verá

  ```
  1 + 2 * 3 = 7
  ```

  En el momento en que se pulsa el segundo igual, en la segunda pantalla se verá

  ```
  1 + 2 * 3 = 7
  2 + 2 = 4
  ```

- Esta segunda pantalla estará implementada como una tabla HTML, cada nueva operacion será una nueva fila en la tabla.
- Además de escribir cada operación en la tabla HTML, deberás almacenarla en un array de JavaScript, de forma que cada línea sea un elemento de este array.
- Cada vez que añadas un elemento a este array (una línea de texto con una operación), haz una traza con `console.log()` de forma que se vea el array. Debe verse el estado del array completo en ese momento, no solo la línea que acabas de añadir.

El tiempo que tardes en hacer el ejercicio será tenido en cuenta, así que cuando lo tengas funcionando, enséñaselo al profesor.

**Desarrollo de Aplicationes Telemáticas**
**Examen de prácticas. 8 de Mayo de 2018**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Universidad Rey Juan Carlos

---

# Ejercicio único

- Ejecuta el script `prepara_dat`

  Esto creará en tu cuenta el directorio `~/dat.mayo.18`

- Copia en el directorio

  `~/dat.mayo.18` los ficheros de tu práctica 5.3, esto es,

  `~/dat/practica05/sesion2.html` y `~/dat/practica05/sesion2.js`

- Modifícalos para que

1. El HTML use Bootstrap.
2. El botón de borrar tenga un *tooltip* describiendo su uso.
3. El programa no almacene solo la fecha de la última visita, sino un array con todas las fechas de las visitas.
4. Todos los elementos del array de fechas se muestren en una tabla en el HTML.
5. Esta tabla sea invisible en la primera visita, cuando el array no esté definido.

El tiempo que tardes en hacer el ejercicio será tenido en cuenta, así que cuando lo tengas funcionando, enséñaselo al profesor.

**Aplicaciones Telemáticas**
**Examen de prácticas. 15 de Junio de 2018**
Grado en Ingeniería Telemática, Universidad Rey Juan Carlos

---

# Ejercicio único

- Ejecuta el script `prepara_at`

  Esto creará en tu cuenta el directorio `~/at.junio.18`

- Copia en el directorio

  `~/at.junio.18` los ficheros de tu práctica 5.3, esto es,

  `~/at/practica05/sesion2.html` y `~/at/practica05/sesion2.js`

- Modifícalos para que

1. El HTML use Bootstrap.
2. El programa no almacene solo la fecha de la última visita, sino un array con todas las fechas de las visitas.
3. Todos los elementos del array de fechas se muestren en una tabla en el HTML.
4. Esta tabla sea invisible en la primera visita, cuando el array no esté definido.

El tiempo que tardes en hacer el ejercicio será tenido en cuenta, así que cuando lo tengas funcionando, enséñaselo al profesor.

**Desarrollo de Aplicationes Telemáticas**
**Examen de prácticas. 13 de Junio de 2018**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Universidad Rey Juan Carlos

---

# Ejercicio único

- Ejecuta el script `prepara_dat`

  Esto creará en tu cuenta el directorio `~/dat.junio.18`

- Copia en el directorio

  `~/dat.junio.18` los ficheros de la calculadora que hiciste en práctica, esto es,

  `~/dat/practica03/calculadora.js` y `~/dat/practica04/calculadora.html`

  Asegúrate de copiar los ficheros originales, no enlaces simbólicos.

- Si lo deseas, puedes usar la versión basada en clases, `~/dat/practica04/ccalculadora.js`

  En este caso tendrás mejor nota.

Modifica tu práctica para que haga lo siguiente:

- Queremos que no solo se vea el resultado final del cálculo, sino también el histórico de operaciones, en una pantalla distinta. No es necesario que esta segunda pantalla se actualice continuamente, es suficiente con que se haga al pulsar el igual
  Supongamos que el usuario pulsa la siguiente secuencia:

  ```
  1 + 2 * 3 = C 2 + 2 =
  ```

  En el momento en que se pulsa el primer igual, en la segunda pantalla se verá

  ```
  1 + 2 * 3 = 7
  ```

  En el momento en que se pulsa el segundo igual, en la segunda pantalla se verá

  ```
  1 + 2 * 3 = 7
  2 + 2 = 4
  ```

- Esta segunda pantalla estará implementada como una tabla HTML, cada nueva operacion será una nueva fila en la tabla.
- Además de escribir cada operación en la tabla HTML, deberás almacenarla en un array de JavaScript, de forma que cada línea sea un elemento de este array.
- Cada vez que añadas un elemento a este array (una línea de texto con una operación), haz una traza con `console.log()` de forma que se vea el array. El estado del array completo en ese momento, no solo la línea que acabas de añadir

El tiempo que tardes en hacer el ejercicio será tenido en cuenta, así que cuando lo tengas funcionando, enséñaselo al profesor.

**Aplicationes telemáticas**
**Examen de prácticas. 9 de mayo de 2019**
Grado en ingeniería telemática, Universidad Rey Juan Carlos

---

# Ejercicio 1. (4 puntos)

En tu práctica 5.1 dibujaste un *canvas* con patrones de camuflaje en franjas horizontales. Posiblemente las franjas se solapaban. Por ejemplo, si dibujabas un círculo con el centro cerca del borde, parte del círculo saldría de su franja, para ocupar la franja continua. Este es un pequeño defecto que el enunciado indicaba que era admisible. En este ejercicio, lo corregirás.

1. Copia tus ficheros de la práctica 5.1, cámbiales el nombre para que se llamen
   `~/at/mayo19/camuflaje2.html` y `~/at/mayo19/camuflaje2.js`
2. Modifícalos para que se ajusten a la siguiente especificación:

- En algún lugar del código debe haber un parámetro llamado *solapamiento_permitido*.
  - Cuando este parámetro valga *true*, el programa se comportará como hasta ahora.
  - Cuando este parámetro no esté definido o cuando valga *false*, la franjas no se solaparán porque habrá entre ellas una pequeña banda blanca, que *borrará* completamente la zona con solapamientos. El tamaño de la banda blanca será el que te parezca adecuado, pero procura que sea pequeño. Observa que para conseguir esa banda basta con que dibujes un rectángulo blanco.

El tiempo que tardes en hacer el ejercicio será tenido en cuenta, así que cuando lo acabes, enséñaselo al profesor. Ten preparado en tu navegador:

- Una pestaña con una ejecución de tu ejercicio con valor *true* en el parámetro *solapamiento_permitido*.
- Otra pestaña donde se vea la ejecución cuando este parámetro sea *false*.

# Ejercicio 2. (6 puntos)

En este ejercicio modificarás tu práctica del cronómetro, para que en vez de tres botones, tenga solo dos, con un nombre que cambiará según la función del botón en cada momento. En todo lo demás, el programa será igual.

- Copia los ficheros de las práctica 3.6 y 4.6, cámbiales el nombre para que se llamen `~/at/mayo19/crono2.html` y `~/at/mayo19/crono2.js`
- Modifícalos para que se ajusten a la siguiente especificación:

Queremos que el cronómetro sea similar a http://cronometro-online.chronme.com
Esto es, que en vez de tres botones, tenga dos.

- Un botón que cuando el cronómetro esté parado, sirva para iniciar. Y cuando esté en marcha, sirva para parar.

  En el ejemplo, el botón *iniciar/parar* siempre mantiene su nombre. Pero en tu ejercicio, se llamará *iniciar* o *parar* según su función actual.

- Otro botón que:

  - Cuando el cronómetro está parado, tenga la función *reset*. Igual que en tu práctica original.
  - Cuando el cronómetro está en marcha, tenga la función *parcial*. Igual que en tu práctica original.

Implementación

- El programa debe estar basado en un autómata finito. Dibújalo en papel y entrégalo. Recomendación: primero dibuja el autómata, luego haz el programa.

Observaciones:

- Los botones pueden tener el nombre que quieras, con tal de que resulte claro. Por ejemplo, da igual *start/stop* que *iniciar/parar*.

El tiempo que tardes en hacer el ejercicio será tenido en cuenta, así que cuando lo acabes, enséñaselo al profesor.

**Desarrollo de Aplicationes Telemáticas**
**Examen de prácticas. 16 de mayo de 2019**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Universidad Rey Juan Carlos

---

## Ejercicio 1. (4 puntos)

En tu práctica 5.1 dibujaste un canvas con patrones de camuflaje en bandas horizontales. Haz una copia de estos ficheros, cambiándoles el nombre para que se llamen
`~/dat/mayo19/camuflaje2.html` y `~/dat/mayo19/camuflaje2.js` y modifícalos para que se ajusten a la siguiente especificación:

1. Los patrones se dibujarán por columnas verticales, no por bandas horizontales.
2. Dentro de cada columna, un rótulo indicará el nombre de cada patrón. El texto debe estar dentro de un *caja* blanca. Esto es, dibuja un rectángulo blanco, y dentro, escribe el texto.

Instrucciones adicionales:

- Las columnas pueden tener el ancho y el alto que quieras, con tal de que sean todas iguales. El texto puede tener la posición y el tamaño que quieras, con tal de que resulte claro y legible. La caja tiene que tener un tamaño que, según tu criterio, sea *adecuado* al texto. Todas las cajas pueden ser iguales (aunque en general la longitud del texto sea variable).
- Esta no es una asignatura de diseño gráfico, no hace falta que busques unas proporciones especialmente elegantes. Basta con que todo tenga un aspecto *razonable*.
- El tiempo que tardes en hacer el ejercicio será tenido en cuenta, así que cuando lo acabes, enséñaselo al profesor.

## Ejercicio 2. (6 puntos)

Copia los ficheros de las práctica 3.6 y 4.6, cámbiales el nombre para que se llamen
`~/dat/mayo19/crono2.html` y `~/dat/mayo19/crono2.js`. Modifícalos para que se ajusten a la siguiente especificación:

1. El cronómetro ya no tendrá botón *parcial* ni la funcionalidad asociada. Borra todo el código relativo a esto.
2. El cronómetro mostrará el tiempo cronometrado, exactamente igual que en tu práctica original. Pero además, mostrará en una tabla HTML
    - La hora inicio del intervalo cronometrado, en formato hh:mm:ss, hora local
    - La hora final del intervalo cronometrado.
    - El tiempo del último intervalo cronometrado.
    - El tiempo total cronometrado.

Ejemplo. Pulsamos start, mostrará:

```
12:52:15
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
```

Esperamos 10 segundos y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4
```

Pulsamos reset. Esto provocará que se genere una fila en blanco. Esperamos 1 segundo y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 -
```

Esperamos 5 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
```

Esperamos 5 segundos y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
12:52:40 -
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
12:52:40 - 12:52:42     2    7
```

Instrucciones adicionales:

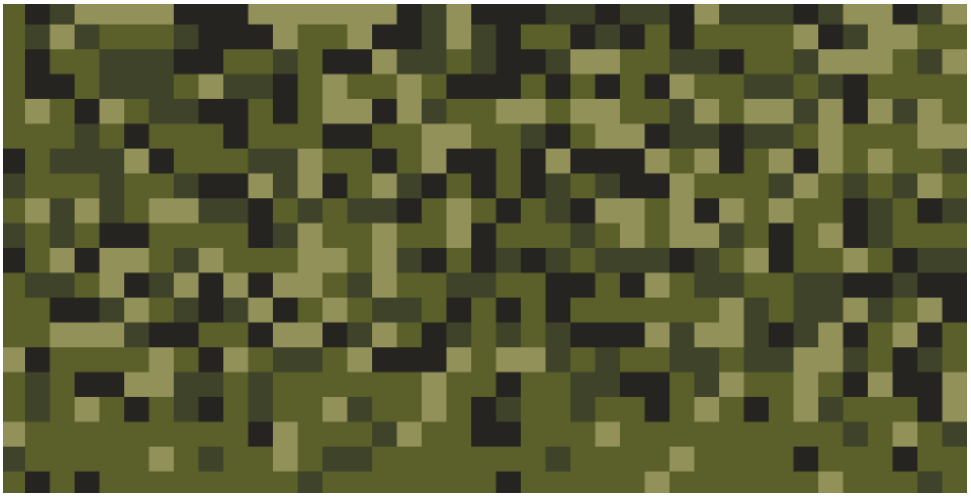
- Dibuja el autómata en papel y entrégalo al final del examen.
- Empieza escribiendo el programa en node.js, no en el navegador. Tienes libertad para usar las funciones y estructuras que quieras, aunque procura que sea similar a la práctica original.
- Cuando parezca funcionar, añade el interfaz HTML para que funcione en el navegador.
- El tiempo que tardes en hacer el ejercicio será tenido en cuenta. Cuando acabes, enséñaselo al profesor.

**Aplicaciones telemáticas**
**Examen de prácticas. 20 de junio de 2019**
Grado en ingeniería telemática, Universidad Rey Juan Carlos

---

# Ejercicio 1. (4 puntos)

Copia los ficheros de tu práctica 5.1 en `~/at.junio.19/camuflaje.html` y `~/at.junio.19/camuflaje.js` y modifícalos para que generen un único patrón, similar a este:



Observa que está formado por cuadrados de 4 colores, sin bordes. Son tonos de verde, llámalos verde01, verde02, verde03 y verde04. Van de más claro a más oscuro, el verde04 es prácticamente negro. Están repartidos aleatoriamente, con predominio de verde02. No es necesario que obtengas un patrón idéntico pero procura aproximarte.

# Ejercicio 2. (6 puntos)

Copia los ficheros de las práctica 3.6 y 4.6, cámbiales el nombre para que se llamen `~/at.junio.19/crono2.html` y `~/at.junio.19/crono2.js`. Modifícalos para que se ajusten a la siguiente especificación:

1. El cronómetro ya no tendrá botón *parcial* ni la funcionalidad asociada. Borra todo el código relativo a esto.
2. El cronómetro mostrará el tiempo cronometrado, exactamente igual que en tu práctica original. Pero además, mostrará en una tabla HTML
   - La hora inicio del intervalo cronometrado, en formato hh:mm:ss, hora local
   - La hora final del intervalo cronometrado.
   - El tiempo del último intervalo cronometrado.
   - El tiempo total cronometrado.

Ejemplo. Pulsamos start, mostrará:

```
12:52:15
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
```

Esperamos 10 segundos y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4
```

Pulsamos reset. Esto provocará que se genere una fila en blanco. Esperamos 1 segundo y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 -
```

Esperamos 5 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
```

Esperamos 5 segundos y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
12:52:40 -
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
12:52:40 - 12:52:42     2    7
```

Instrucciones adicionales:

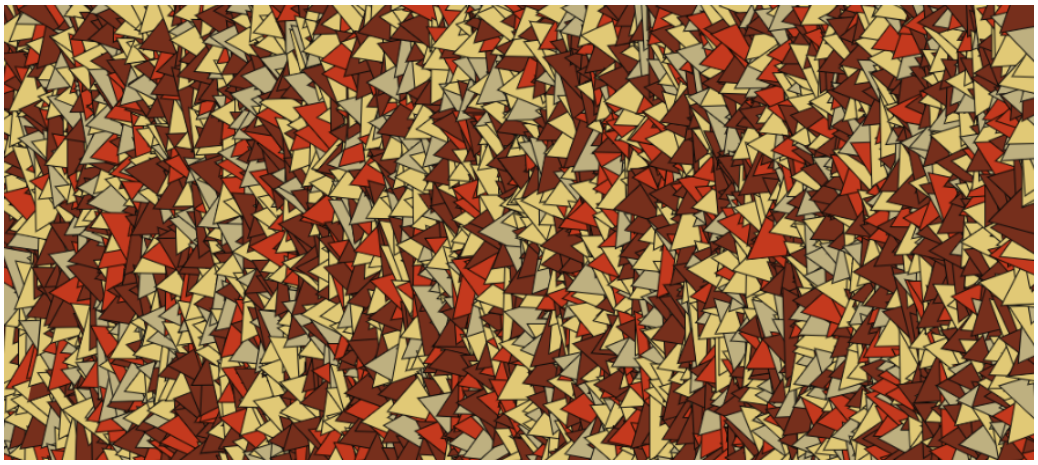
- Dibuja el autómata en papel y entrégalo al final del examen.
- Empieza escribiendo el programa en node.js, no en el navegador. Tienes libertad para usar las funciones y estructuras que quieras, aunque procura que sea similar a la práctica original.
- Cuando parezca funcionar, añade el interfaz HTML para que funcione en el navegador.
- El tiempo que tardes en hacer el ejercicio será tenido en cuenta. Cuando acabes, enséñaselo al profesor.

**Desarrollo de Aplicationes Telemáticas**
**Examen de prácticas. 18 de junio de 2019**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Universidad Rey Juan Carlos

---

## Ejercicio 1. (4 puntos)

Copia los ficheros de tu práctica 5.1 en `~/dat.junio.19/camuflaje.html` y `~/dat.junio.19/camuflaje.js` y modifícalos para que generen un único patrón, similar a este. Observa que está formado por triángulos de 4 colores, con posición, tamaño y rotación aleatoria. No es necesario que obtengas un patrón idéntico pero procura aproximarte.



## Ejercicio 2. (6 puntos)

Copia los ficheros de las práctica 3.6 y 4.6, cámbiales el nombre para que se llamen `~/dat.junio.19/crono2.html` y `~/dat.junio.19/crono2.js`. Modifícalos para que se ajusten a la siguiente especificación:

1. El cronómetro ya no tendrá botón *parcial* ni la funcionalidad asociada. Borra todo el código relativo a esto.

2. En tu práctica original había un botón *start-stop* que ponía en marcha el cronómetro (si estaba parado) o lo paraba (si estaba en marcha). Modifícalo para que el texto en el botón sea *start* si el cronómetro está parado, y sea *stop* si el cronómetro está en marcha.

3. El cronómetro mostrará el tiempo cronometrado, exactamente igual que en tu práctica original. Pero además, mostrará en una tabla HTML

   - La hora inicio del intervalo cronometrado, en formato hh:mm:ss, hora local
   - La hora final del intervalo cronometrado.
   - El tiempo del último intervalo cronometrado.
   - El tiempo total cronometrado.

Ejemplo. Pulsamos start, mostrará:

```
12:52:15
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
```

Esperamos 10 segundos y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4
```

Pulsamos reset. Esto provocará que se genere una fila en blanco. Esperamos 1 segundo y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 -
```

Esperamos 5 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
```

Esperamos 5 segundos y pulsamos start.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
12:52:40 -
```

Esperamos 2 segundos y pulsamos stop.

```
12:52:15 - 12:52:17     2    2
12:52:27 - 12:52:29     2    4

12:52:30 - 12:52:35     5    5
12:52:40 - 12:52:42     2    7
```

Instrucciones adicionales:

- Dibuja el autómata en papel y entrégalo al final del examen.
- Empieza escribiendo el programa en node.js, no en el navegador. Tienes libertad para usar las funciones y estructuras que quieras, aunque procura que sea similar a la práctica original.
- Cuando parezca funcionar, añade el interfaz HTML para que funcione en el navegador.
- El tiempo que tardes en hacer el ejercicio será tenido en cuenta. Cuando acabes, enséñaselo al profesor.

**Desarrollo de Aplicationes Telemáticas**
**Examen parcial, prueba de prácticas. 14 de abril de 2021**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

### Instrucciones

1. Ejecuta en un terminal `~mortuno/prepara`
2. Comprueba que esto ha dejado en tu cuenta los ficheros
   - `~/dat.abril.21/mitos.html`
   - `~/dat.abril.21/textos.txt`

   así como el directorio
   - `~/dat.abril.21/images`
     que contiene los ficheros
     `mitos.jpg`, `foto00.jpg`, `foto01.jpg`, `foto02.jpg` y `foto03.jpg`

## Ejercicio 4. (7 puntos)

Edita el fichero `~/dat.abril.21/mitos.html` para que se parezca a `mitos.jpg`. Observaciones:

- El uso del W3C HTML Validator está permitido y es recomendable.
- No es necesario que teclees los textos de la página, están en el fichero `textos.txt`.
- Haz que tu documento html lea las imágenes desde el directorio donde están actualmente, esto es, el subdirectorio *images*.
- Usa colores aproximados, no es necesario que sean idénticos.
- Las fotos tienen un *padding* equivalente al ancho de una letra *normal*.
- El título del documento que aparece en la primera línea tiene un tamaño equivalente a 4 letras *normales*.
- Los párrafos tienen un margen equivalente a una letra normal. Los elementos de lista, la mitad de una letra.
- Todos los atributos no descritos aqui elígelos *a ojo* para que se parezcan al ejemplo. No te preocupes por hacerlo idéntico.
- No lo hemos visto en clase, pero puedes conseguir que las fotos estén alineadas horizontalmente añadiendoles la clase *center-block*.
- Aunque no sea visible, recuerda que en HTML es necesario incluir una breve descripción de cada imagen en el código fuente. Pon el texto que creas adecuado.
- Añade la atribución de autoría de cada foto como comentarios dentro del código fuente. Está en el documento *textos.txt*.

# COVID 19: Mitos sobre las vacunas


## Mito 1. Las vacunas no son seguras porque están hechas a todo prisa y sin probar bien.

Esto es falso. Todas las vacunas, incluyendo las del coronavirus, siguen las mismas fases de desarrollo

- Fase preclínica: Se realiza en animales. Se evalúa el mecanismo de acción, toxicidad e inmunología.
- Fase clínica 1: Participan menos de 100 voluntarios. Se evalúa la seguridad y la respuesta inmunológica en adultos jóvenes sanos.
- Fase clínica 2: De 100 a 1000 participantes. Se evalúa la dosis, vías de administración, eficacia y seguridad en distintos grupos de población.
- Fase clínica 3: Participan miles de voluntarios. Se prueba la eficacia y seguridad. Si los resultados son positivos, se evalúa para aprobación.
- Fase clínica 4. Denominada de farmacovigilancia, se concentra en el seguimiento de efectos secundarios que pueden aparecer cuando la vacuna se aplica a mayor escala. No concluye nunca desde que la vacuna es autorizada.

Las vacunas contra la COVID-19 se desarrollaron rápidamente en comparación con vacunas anteriores porque científicos, médicos y agencias gubernamentales de todo el mundo invirtieron dinero y otros recursos en cantidades sin precedente para concentrar el trabajo, reducir plazos, trámites y publicar los resultados tan pronto como estaban disponibles. Las autoridades sanitarias están haciendo un seguimiento minucioso de todas las reacciones adversas y estas han sido extraordinariamente inusuales. En general, los problemas se presentaron en personas que ya tenían antecedentes de alergias graves. El riesgo general de sufrir una reacción alérgica grave es prácticamente el mismo que el de otras vacunas comunes.

Incluso en la vacuna más controvertida, la de Astra-Zeneca, la tasa de problemas severos es del orden de 0.000001, esto es, 1 entre 1 millón. Sería deseable disponer de vacunas o de medicamentos con un tasa de efectos secundarios severos de 0.0, pero esto es imposible. Naturalmente hay que identificar, minimizar y tratar estos casos, pero las vacunas salvan la vida y la salud de millones de personas. Pero rechazarlas porque causa problemas en unos pocos individuos es un comportamiento que no es racional.

## Mito 2. Soy joven, estoy sano. Esto no me afecta, no necesito la vacuna.

Esto es falso. La COVID-19 puede infectar a personas de todas las edades, si bien se ha observado que las personas mayores y las que padecen algunas enfermedades (como el asma, la diabetes o las cardiopatías) tienen más probabilidades de enfermarse gravemente cuando adquieren la infección.

Estar protegido para evitar enfermarse es importante porque, aunque muchas personas con COVID-19 solo presentan manifestaciones leves de la enfermedad, otras pueden enfermarse gravemente, experimentar efectos en la salud a largo plazo o incluso morir. Es imposible saber cómo le afectará el COVID-19, muchas personas jóvenes y sanas enferman gravemente o mueren, aunque no sean tantas como las personas mayores o enfermas.

Sin olvidar que cada persona infectada contagia en promedio a otras 3, cada una de ellas a 3 más, sucesivamente. Las vacunas son imprescindibles para poder detener este crecimiento exponencial.

## Mito 3. Tomo muchos suplementos de vitaminas y minerales, incluyendo hidroxicloroquina. No necesito la vacuna.

Esto es falso. Los micronutrientes, como las vitaminas D y C o el zinc, son fundamentales para el buen funcionamiento del sistema inmunológico y desempeñan un papel vital para la salud y el bienestar nutricional. En la actualidad, no hay ninguna indicación sobre el uso de suplementos de micronutrientes como tratamiento de la COVID-19.

Se han estudiado los efectos de la hidroxicloroquina y la cloroquina —fármacos empleados para tratar el paludismo, el lupus eritematoso y la artritis reumatoide— como posibles terapias contra la COVID-19. Algunos resultados apuntan que estos fármacos podrían resultar útiles para determinadas formas leves de la enfermedad, o como profilaxis previa o posterior a la exposición a la COVID-19.

No obstante, la OMS afirma que los datos actuales indican que no reducen la mortalidad de los pacientes de COVID-19 hospitalizados ni son de ayuda para las personas con síntomas moderados de esta enfermedad. En general, se considera que es seguro tomar hidroxicloroquina o cloroquina para tratar el paludismo y las enfermedades autoinmunes, pero que si se toman sin estar indicadas y sin supervisión médica pueden ocasionar efectos secundarios graves, por lo que deben evitarse.

Modelo de la página

**Aplicationes Telemáticas**
**Examen de junio, prueba de prácticas. 4 de junio de 2021**
Grado en ingeniería telemática
Universidad Rey Juan Carlos

---

# Ejercicio 1. (5 puntos)

Si repasas las fórmulas de las funciones lineales, recordarás que:

- La ecuación de una recta es $y = mx + b$. Donde $m$ es la pendiente de la recta y $b$ es la intersección con el eje $y$.
- Dos puntos en un plano definen una recta.
- A partir de dos puntos $(x_0, y_0), (x_1, y_1)$, la pendiente se puede calcular con la fórmula.

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

- El valor $b$ se puede calcular con la fórmula $b = y_0 - mx_0$ o bien con $b = y_1 - mx_1$ (ambos valores son iguales).

1. Escribe un programa `~/at.junio.21/recta.js` que contenga las funciones necesarias para calcular la ecuación de una recta a partir de las coordenadas de dos puntos.
2. Escribe un fichero `~/at.junio.21/recta.html` que:
   - Contenga un formulario que solicite al usuario las coordenadas de dos puntos. También el número de decimales deseados para la precisión del resultado.
   - Invoque al programa anterior para mostrar la ecuación (dentro de la misma página web, no como ventana emergente).
   - Si las coordenadas $x$ de ambos puntos fuera la misma, la pendiente sería infinita. En este caso, el programa lo advertirá con el mensaje adecuado.

   Ejemplo:

   - Si el usuario introduce los puntos (3.5,-2) (8,4) y una precisión de 2, el programa mostrará un mensaje similar a este:

     ```
     La recta resultante tiene como ecuación y = 1.33 x + -6.66  (redondeo = 2 decimales)
     ```

     O mejor aún

     ```
     La recta resultante tiene como ecuación y = 1.33 x - 6.66  (redondeo = 2 decimales)
     ```

Recuerda que para redondear un número puedes usar el método `toFixed()`. Ejemplo:

```
> pi=3.14159265358979
3.14159265358979
> pi.toFixed(3)
'3.142'
```

# Ejercicio 2. (5 puntos)

Escribe un programa en JavaScript atendiendo a la siguiente especificación:

- Copia tus ficheros `~/at/practica05/paletas.html` y `~/at/practica05/paletas.js` en los ficheros `~/at.junio.21/vertical.html` y `~/at.junio.21/vertical.js`.
- Modifícalos para que el resultado sea similar al original, pero con franjas verticales.

Observa que esto se puede hace de diversas formas: rotando las figuras o sin rotarlas, redimensionándolas o no. Hazlo como prefieras, tienes libertad. Basta con que sea *semejante*. Pero se valorará que la figura siga siendo *vistosa*.

**Desarrollo de Aplicaciones Telemáticas**
**Examen de junio, prueba de prácticas. 2 de junio de 2021**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

# Ejercicio 1. (5 puntos)

Si repasas la fórmula para resolver ecuaciones de segundo grado ($ax^2+bx+c=0$), recordarás que

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}$$
$$x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

Si $a$ es nulo no es una ecuación de segundo grado (el usuario estaría escribiendo un valor erróneo). Si el discriminante ($b^2-4ac$) es negativo, la ecuación no tendrá soluciones reales. Si el discriminante es nulo, tendrá una solución. En otro caso, tendrá dos soluciones.

1. Escribe un programa `~/dat.junio.21/ecuacion.js` que contenga las funciones necesarias para calcular los valores de ecuaciones de segundo grado: esto es, cuántas son sus soluciones reales y cuáles son sus valores (si es que tiene alguno).

2. Escribe un fichero `~/dat.junio.21/ecuacion.html` que:

   - Contenga un formulario que solicite al usuario los coeficientes $a$, $b$ y $c$ de una ecuación de segundo grado. También el número de decimales deseados para la precisión del resultado.
   - Invoque al programa anterior para mostrar el resultado, incluyendo los mensajes de error que corresponda (dentro de la misma página web, no como ventana emergente).

Recuerda que para redondear un número puedes usar el método `toFixed()`. Ejemplo:

```
> pi=3.14159265358979
3.14159265358979
> pi.toFixed(3)
'3.142'
```

# Ejercicio 2. (5 puntos)

Escribe un programa en JavaScript atendiendo a la siguiente especificación:

- Copia tus ficheros `~/dat/practica05/paletas.html` y `~/dat/practica05/paletas.js` en los ficheros `~/dat.junio.21/franjas.html` y `~/dat.junio.21/franjas.js`.
- Modifícalos para que en cada una de las franjas lo que aparezca sea una figura similar al ejemplo *rotacion_degradado_01* de las transparencias

  https://gsyc.urjc.es/~mortuno/rotacion_degradado_01.html

  Pero en cada franja, los triángulos deberán tener un color (*hue*) distinto.

**Aplicationes telemáticas**
**Examen de mayo, prueba de prácticas. 20 de mayo de 2022**
Grado en ingeniería telemática
Universidad Rey Juan Carlos

---

# Ejercicio único. (10 puntos)

Escribe una aplicación cliente-servidor en JavaScript contemporáneo atendiendo a la siguiente especificación:

## Cliente

1. El código del cliente estará en los ficheros

   `~/at.mayo.22/cliente.html`

   `~/at.mayo.22/js/cliente.js`

2. La página web del cliente mostrará todas las *figuras* de tu práctica 4.4. Esto es, una *figura* será una carta, una pieza de dominó, una figura de la tragaperras, etc

3. Cuando el usuario haga click sobre una figura, se verá un efecto *dar la vuelta*. En vez de verse la figura ordinaria, el anverso, se verá el *reverso*: una imagen *neutra*, siempre la misma. Por ejemplo, si es una carta, la cara posterior de una carta. Si es un valor de una tragaperras, puede ser por ejemplo la imagen de un interrogante.

4. Si es necesario, el profesor te ayudará a preparar esta *imagen neutra*. (pero no te ayudará a nada más)

5. Al hacer click de nuevo sobre un figura en estado *reverso*, pasará al estado normal, en que se volverá a ver el anverso.

6. Cada vez que el usuario haga click sobre una carta, el cliente informará al servidor de qué carta ha sido, y a qué estado pasa una vez que ha recibido el click. El protocolo será REST/ROA.

7. Habrá un botón con el texto *dime mi estado*. Al pulsarlo, el cliente recibirá del servidor una descripción en modo texto del estado de todas las figuras. Esto es: qué figuras son y en qué estado (anverso o reverso) está cada una. Esta información se mostrará en el HTML, en modo texto.

## Servidor

1. El código del servidor se ejecutará en Express y estará en el fichero

   `~/at.mayo.22/servidor.js`

2. El servidor recibirá la información del cliente y la mostrará por consola, a modo de traza.

3. El servidor guardará la información que le remite el cliente, en la estructura de datos que creas más adecuada.

4. Cuando el servidor reciba la petición del cliente, le devolverá el informe en modo texto con el estado de cada figura. No es necesario que esté en un formato muy cuidado, considéralo una especie de traza, pero mostrada en el HTML.

**Desarrollo de Aplicaciones Telemáticas**
**Examen de mayo, prueba de prácticas. 26 de mayo de 2022**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

# Ejercicio único. (10 puntos)

Escribe una aplicación cliente-servidor en JavaScript contemporáneo atendiendo a la siguiente especificación:

## Cliente

1. El código del cliente estará en los ficheros

   `~/dat.mayo.22/cliente.html`

   `~/dat.mayo.22/js/cliente.js`

2. El cliente hará lo mismo que tu práctica 4.7, añadiendo (o modificando) todo lo indicado a continuación. (Por tanto empieza copiando tu práctica 4.7 en los ficheros indicados)

3. Llamemos *figura* a una carta, una pieza de dominó, una figura de la tragaperras, etc. Cada vez que el usuario pulse sobre una figura, en cierta zona del HTML se verá un texto con el nombre de esa figura. Texto que se irá actualizando. En pantalla se verá algo como *Última carta pulsada: 3 de copas*.

4. Recuerda lo que aprendiste en un práctica 3.11 y haz que cada vez que el usuario pulse sobre una figura, se inserte su nombre (o más bien su clave) en un array.

5. Usando la información de este array, haz que cuando el usuario haga click pulsa sobre una figura, en el HTML se indique cuántas veces lo ha hecho.

6. Cada vez que el usuario haga click sobre una figura, el cliente informará al servidor de qué figura ha sido.

7. El cliente no generará ningún valor aleatorio: se lo pedirá al servidor y este se lo proporcionará.

## Servidor

1. El código del servidor se ejecutará en Express y estará en el fichero

   `~/dat.mayo.22/servidor.js`

2. El servidor recibirá la información del cliente y la mostrará por consola, a modo de traza.

3. El servidor devolverá al cliente el código de una figura aleatoria cuando este se lo pida.

**Desarrollo de Aplicaciones Telemáticas**
**Examen práctico. 21 de junio de 2022**
Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

# Ejercicio único. (10 puntos)

Escribe una aplicación cliente-servidor en JavaScript contemporáneo atendiendo a la siguiente especificación:

## Cliente

1. El código del cliente estará en los ficheros

   `~/dat.junio.22/cliente.html`

   `~/dat.junio.22/js/cliente.js`

2. La página web del cliente mostrará todas las *figuras* de tu práctica 4.4. Esto es, una *figura* será una carta, una pieza de dominó, una figura de la tragaperras, etc

3. Cuando el usuario haga click sobre una figura, se verá un efecto *dar la vuelta*. En vez de verse la figura ordinaria, el anverso, se verá el *reverso*: una imagen *neutra*, siempre la misma. Por ejemplo, si es una carta, la cara posterior de una carta. Si es un valor de una tragaperras, puede ser por ejemplo la imagen de un interrogante.

4. Si es necesario, el profesor te ayudará a preparar esta *imagen neutra*. (pero no te ayudará a nada más)

5. Al hacer click de nuevo sobre un figura en estado *reverso*, pasará al estado normal, en que se volverá a ver el anverso.

6. Cada vez que el usuario haga click sobre una figura, el cliente informará al servidor de qué figura ha recibido el evento. El protocolo será REST/ROA.

7. Habrá un botón con el texto *dime mi estado*. Al pulsarlo, el cliente recibirá del servidor una descripción en modo texto del estado de todas las figuras. Esto es: qué figuras son y en qué estado (anverso o reverso) está cada una. Esta información se mostrará en el HTML, en modo texto.

## Servidor

1. El código del servidor se ejecutará en Express y estará en el fichero

   `~/dat.junio.22/servidor.js`

2. El servidor recibirá la información del cliente y la mostrará por consola, a modo de traza.

3. El servidor guardará la información que le remite el cliente, en la estructura de datos que creas más adecuada.

4. Cuando el servidor reciba la petición del cliente, le devolverá el informe en modo texto con el estado de cada figura. No es necesario que esté en un formato muy cuidado, considéralo una especie de traza, pero mostrada en el HTML.

**Aplicaciones Telemáticas, examen práctico, 15/05/2023**

Grado en Ingeniería Telemática

Universidad Rey Juan Carlos

---

### Instrucciones

1. Ejecuta en un terminal `~mortuno/prepara` y comprueba que esto ha creado los ficheros

   - `~/at.mayo.23/distancias.TULOGIN.js` donde resolverás el ejercicio 1.
   - `~/at.mayo.23/figuras.TULOGIN.html` y `~/at.mayo.23/figuras.TULOGIN.js` que contienen una copia de tus ficheros de la práctica 4.7. Deberás modificarlos para hacer el ejercicio 2.

2. Emplea lo visto en clase, y solo lo visto en clase (que puedes consultar en las transparencias). No será válido ni JavaScript más antiguo ni más avanzado.

## Ejercicio 1. (3 puntos)

Para conocer la distancia entre dos puntos en el plano, basta aplicar el teorema de Pitágoras. Pero como la Tierra es esférica, para conocer la distancia entre dos puntos del planeta a partir de sus coordenadas es necesario aplicar la fórmula del semiverseno.

En el fichero `~/at.mayo.23/distancias.TULOGIN.js` encontrarás una implementación en JavaScript de la fórmula del semiverseno. Úsala para escribir un programa de acuerdo con la siguiente especificación.

1. El programa tendrá una función llamada *informe()* que recibirá un array de puntos.
2. Cada elemento de este array será a su vez otro array, formado por una cadena con el nombre del punto, su latitud y su longitud.
   Ejemplo:

```
p1 = ["madrid",  40.416, -3.703]
p2 = ["oviedo", 43.362, -5.848]
p3 = ["fuenlabrada", 40.284, -3.799]
p4 = ["sevilla",  37.385, -5.994]
puntos = [p1,p2,p3,p4]
```

3. La función *informe()* devolverá una lista con la distancia de cada punto con todos los demás (excluido él mismo).
4. Cada elemento de esta lista será a su vez otra lista

   - Su primer elemento será una cadena con el nombre del primer punto.
   - Su segundo elemento será una cadena con el nombre del segundo punto.
   - Su tercer elemento será un número entero con la distancia en kilómetros entre ambos puntos.

5. Ejemplo de valor devuelvo

```
[
  [ 'madrid', 'oviedo', 373 ],
  [ 'madrid', 'fuenlabrada', 17 ],
  [ 'madrid', 'sevilla', 391 ],
  [ 'oviedo', 'madrid', 373 ],
  [ 'oviedo', 'fuenlabrada', 382 ],
  [ 'oviedo', 'sevilla', 665 ],
  [ 'fuenlabrada', 'madrid', 17 ],
  [ 'fuenlabrada', 'oviedo', 382 ],
  [ 'fuenlabrada', 'sevilla', 374 ],
  [ 'sevilla', 'madrid', 391 ],
  [ 'sevilla', 'oviedo', 665 ],
  [ 'sevilla', 'fuenlabrada', 374 ]
]
```

6. La función *informe()* devolverá la lista requerida, pero no tendrá efectos laterales, incluyendo el no escribir nada en la salide estándar. En otras palabras: aunque es conveniente que uses trazas, bórralas o coméntalas cuando acabes.

7. El programa puede tener todas las funciones adicionales que te parezca conveniente.

### Solución)

```
'use strict'
function semiverseno(coords1, coords2) {
    // Recibe dos parámetros, cada uno es una lista [latitud, longitud]
    // Devuelve la distancia en metros entre ambos puntos.
    // Extraido de https://github.com/dcousens/haversine-distance

    function toRad(x) {
      return x * Math.PI / 180;
    }

    var lat1 = coords1[0];
    var lon1 = coords1[1];

    var lat2 = coords2[0];
    var lon2 = coords2[1];

    var R = 6371000; // metros

    var x1 = lat2 - lat1;
    var dLat = toRad(x1);
    var x2 = lon2 - lon1;
    var dLon = toRad(x2)
    var a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
      Math.cos(toRad(lat1)) * Math.cos(toRad(lat2)) *
      Math.sin(dLon / 2) * Math.sin(dLon / 2);
    var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    var d = R * c;

    return d;
}

function distancia(a,b){
    return semiverseno( [a[1],a[2]], [b[1],b[2]]) / 1000;
}

let p1 = ["madrid",  40.416, -3.703]
let p2 = ["oviedo", 43.362, -5.848]
let p3 = ["fuenlabrada", 40.284, -3.799]
let p4 = ["sevilla",  37.385, -5.994]

let puntos = [p1,p2,p3,p4]

function informe(puntos){
    let rval = []
    let elemento;
    for  (let a of puntos){
        for (let b of puntos){
            if (a !== b){
                elemento = [];
                elemento.push(a[0]);
                elemento.push(b[0]);
                elemento.push(Math.round(distancia(a,b)));
                rval.push(elemento);
            }
        }
    }
    return rval;
}

console.log(informe(puntos))
```

## Ejercicio 2. (7 puntos)

Modifica los ficheros `~/at.mayo.23/figuras.TULOGIN.html` y `~/at.mayo.23/figuras.TULOGIN.js` según esta especificación:

1. Hasta ahora, tu programa o bien añadía una figura en modo texto o bien añadía una figura en modo gráfico. Una figura en modo texto siempre se quedaba en modo texto, una figura en modo gráfico siempre se quedaba en modo gráfico.

   Ahora, tu programa debe tener un único botón para añadir figuras. Y un segundo botón para elegir *modo gráfico* o *modo texto*.

   - En modo gráfico, todas las figuras se verán en modo gráfico. Incluyendo las que previamente fueron lanzadas en modo texto.
   - En modo texto, todas las figuras se verán en modo texto, incluyendo las que previamente fueron lanzadas en modo gráfico.

2. Tu programa siempre trabajará en modo *sin repetición*. Borra todo lo relativo al modo *con repetición*.

3. En tu programa se tiene que ir viendo el histórico de todas las figuras aparecidas.

   - Cuando sale una figura nueva no puede borrar la anterior.
   - Al cambiar el modo gráfico/texto, no pueden borrar las figuras.
   - Al pulsar *nuevo mazo*, las figuras sí se pueden borrar. O no. Como prefieras.

4. Cuando tu programa esté en *modo gráfico*, el botón de cambio de modo debe contener el texto *modo texto*, porque al pulsarlo este será el nuevo modo.

5. Cuando tu programa esté en *modo texto*, el botón de cambio de modo debe contener el texto *modo gráfico*, porque al pulsarlo este será el nuevo modo.

Sugerencia

- Cuando añadas una figura, añade simultáneamente el texto y la imagen. Pero con clases distintas. Luego, al cambiar de modo, haz que unas u otras sean visibles o invisibles, según corresponda.

**Desarrollo de Aplicaciones Telemáticas, examen práctico, 11/05/2023**

Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

### Instrucciones

1. Ejecuta en un terminal `~mortuno/prepara` y comprueba que esto ha creado los ficheros
   - `~/dat.mayo.23/listas.TULOGIN.js` donde resolverás el ejercicio 1.
   - `~/dat.mayo.23/cuentaImagen.TULOGIN.html` y `~/dat.mayo.23/cuentaImagen.TULOGIN.js` que contienen una copia de tus ficheros de la práctica 4.7. Deberás modificarlos para hacer el ejercicio 2.
2. Emplea lo visto en clase, y solo lo visto en clase (que puedes consultar en las transparencias). No será válido ni JavaScript más antiguo ni más avanzado.

## Ejercicio 1. 4 puntos

Escribe un programa en JavaScript en el fichero `~/dat.mayo.23/listas.TULOGIN.js` de acuerdo con la siguiente especificación.

1. El programa tendrá una función llamada *extraer_elemento* que:
   *a*) Recibirá dos parámetros: una lista y un elemento.
   *b*) Si el elemento pertenece a la lista, lo extraerá. De forma destructiva, esto es, la lista quedará modificada, dejará de contener el elemento.
   *c*) Si el elemento está incluido más de una vez, extraerá la primera aparición y solo la primera aparición.
   *d*) Si el elemento no pertenece a la lista, la función no hará nada.
   *e*) No es necesario que compruebes si el programador invoca correctamente a esta función.
   *f*) La función devolverá *true* si ha modificado la lista, *false* en otro caso.
   *g*) Ejemplo de traza:

```
[ 'SOTA', 'sota', 'sota', 'caballo' ]
extraemos   sota
[ 'SOTA', 'sota', 'caballo' ]
extraemos   Comodín
[ 'SOTA', 'sota', 'caballo' ]
extraemos   sota
[ 'SOTA', 'caballo' ]
```

2. El programa tendrá una función llamada *extraer_elementos* que:
   *a*) Recibirá dos parámetros: una lista y un elemento
   *b*) Llamando a la función anterior, eliminará de forma destructiva todas las apariciones del elemento en la lista. Si no hay ninguna, no hará nada.
   *c*) La lista quedará modificada, si procede, pero la función no devolverá ningún valor.
   *d*) Ejemplo de traza:

```
[ 'SOTA', 'sota', 'sota', 'caballo' ]
extraemos todas las apariciones de  sota
[ 'SOTA', 'caballo' ]
```

3. El programa tendrá una función llamada *contar_elemento* que:

*a*) Recibirá dos parámetros: una lista y un elemento

*b*) Devolverá el número de veces que el elemento aparece en la lista.

*c*) Ejemplo de traza:

```
[ 'sota', 'caballo', 'sota', 'Sota' ]
apariciones de rey  : 0
apariciones de sota  : 2
```

4. El programa contendrá llamadas a estas funciones para probar su funcionamiento. Podrá tener funciones adicionales si lo crees conveniente.

## Solución

```
'use strict'
function extraer_elemento(lista, elemento){
    let rval = false;
    let indice = lista.indexOf(elemento);
    if (indice!==-1){
        lista.splice(indice,1);
        rval = true;
    }
    return rval;
}

function extraer_elementos(lista, elemento){
    while( extraer_elemento(lista, elemento) ){
    }
}

function contar_elemento(lista, elemento){
    let rval = 0;
    for (let l of lista){
        if (l === elemento) {
        rval = rval+1;
        }
    }
    return rval;
}

let listado = ['SOTA', 'caballo', 'sota', 'sota', 'sota', 'sota']
let elemento;

console.log('listado:', listado)
elemento = "rey";
console.log("apariciones de",elemento,":",contar_elemento(listado,elemento))

elemento = "sota";
console.log("apariciones de",elemento,":",contar_elemento(listado,elemento))
elemento="sota"

extraer_elemento(listado,elemento)
console.log("extraemos",elemento)
console.log(listado)

elemento="caballo"
extraer_elementos(listado,elemento)
console.log("extraemos todos los ",elemento)
console.log(listado)

elemento="sota";
console.log("extraemos todos los ",elemento)
extraer_elementos(listado,elemento)
console.log(listado)

elemento="Comodín";
```

```
console.log("extraemos  ",elemento)
extraer_elemento(listado,elemento)
console.log(listado)
```

## Ejercicio 1. 6 puntos

Modifica los ficheros `~/dat.mayo.23/cuentaImagen.TULOGIN.html`
y `~/dat.mayo.23/cuentaImagen.TULOGIN.js` de forma que sigan haciendo todo lo que hacían hasta ahora, y además, lo siguiente.

1. Cada vez que crees una de tus figuras en modo *sin repetición*, junto a ella aparecerá un botón al que llamaremos *sumar*, cuyo texto será el signo *más*. El *id* de este botón será el prefijo `sum_`, concatenado con el *id* de la imagen. Ejemplo, si la imagen es `reyPicas`, el botón será `sum_reyPicas`.

2. Cada vez que crees una de tus figuras en modo *sin repetición*, junto a ella aparecerá, además, un botón al que llamaremos *restar* cuyo texto será el signo *menos*. El *id* de este botón será el prefijo `res_`, concatenado con el *id* de la imagen. Ejemplo, si la imagen es `reyPicas`, el botón será `res_ReyPicas`.

3. Cuando tu programa trabaje en modo texto, no habrá novedades, no aparecerán estos botones. Tampoco cuando tu programa trabaje en modo imagen con repetición.

4. No te preocupes por el aspecto y disposición de estos dos botones. A menos que hayas resuelto el examen completo y te sobre tiempo.

5. Cuando el usuario haga *click* en alguna imagen o en los botones *sumar* o en los botones *restar*, en algún lugar de la pantalla se verá el *id* de la imagen o el botón.

6. Tu programa contendrá un array llamado *listado*, que inicialmente estará vacío.

7. Cada vez que el usuario haga *click* sobre un botón *sumar*, el programa añadirá a *listado* el *id* de la imagen correspondiente al botón a *listado*. Si ese *id* ya estaba en *listado*, aparecerá una vez más.
   Ejemplo de traza:

   ```
   Listado: [ 'reinaCorazones', 'reyPicas']
   El usuario hace click en sum_ReyPicas
   Listado: [ 'reinaCorazones', 'reyPicas', 'reyPicas']
   ```

8. Cada vez que el usuario haga *click* sobre un botón *restar*, el programa eliminará de *listado* una aparición del *id* de la imagen correspondiente. Si el *id* no estaba en la lista, no hará nada. Emplea para esto la función *extraer_elemento* del ejercicio anterior (si te funciona)
   Ejemplo de traza:

   ```
   Listado: [ 'reinaCorazones', 'reyPicas', 'reyPicas']
   El usuario hace click en res_reyPicas]
   Listado: [ 'reinaCorazones', 'reyPicas']
   ```

9. Observa que una vez que conozcas el *id* de un botón, podrás obtener el *id* de la imagen asociada quedándote con una subcadena. Ejemplo: la imagen asociada al botón *sum_reyPicas* es *reyPicas*.

10. En alguna parte de la pantalla se verá el *listado* y se actualizará cada vez que se modifique.

11. Recuerda que es muy conveniente hacer trazas detalladas. Si quieres puedes dejar todas las trazas, pero es preferible dejar sólo las que consideres más importantes.

### Solución

Una solución completa real tendría que estar basada en una praćtica 4.7. A continuación se muestra un programa similar, que contiene código como el que habría que añadir a una práctica.
`https://gsyc.urjc.es//~mortuno/cuentaImagen.html`

**Desarrollo de Aplicaciones Telemáticas, examen práctico, 30/06/2023**

Grado en Ingeniería en Tecnologías de la Telecomunicación
Grado en Ingeniería en Sistemas de la Telecomunicación
Universidad Rey Juan Carlos

---

### Instrucciones

1. Ejecuta en un terminal `~mortuno/prepara` y comprueba que esto ha creado los ficheros

   - `~/dat.junio.23/matrices.TULOGIN.js` donde resolverás el ejercicio 1.
   - `~/dat.junio.23/figuras.TULOGIN.html` y `~/dat.junio.23/figuras.TULOGIN.js` que contienen una copia de tus ficheros de la práctica 4.7. Deberás modificarlos para hacer el ejercicio 2.

2. Emplea lo visto en clase, y solo lo visto en clase (que puedes consultar en las transparencias). No será válido ni JavaScript más antiguo ni más avanzado.

## Ejercicio 1. (3.5 puntos)

En este ejercicio escribirás un par de funciones para generar y procesar matrices. Una matriz no es más que una lista cuyos elementos son filas, y estas filas, son a su vez filas. Completa el fichero `~/dat.junio.23/matrices.TULOGIN.js` para que sea un programa en JavaScript de acuerdo con la siguiente especificación.

1. Los elementos de la matriz serán cadenas formadas por una letra minúscula, un único dígito para indicar la fila ($i$) y un único dígito para indicar la columna ($j$). P.e. la cadena `a23` sería el elemento de la segunda fila, tercera columna.

   Un ejemplo de una de estas matrices sería

   ```
   [
     [ 'a10', 'a11', 'a12' ],
     [ 'a20', 'a21', 'a22' ],
   ]
   ```

2. Escribe una función llamada *crea_fila* que devuelva una fila de esta estructura. Recibirá:

   - Un prefijo, que será una letra minúscula.
   - Un número (entero), formado por la concatenación de los índices $i$, $j$ del primer elemento de la fila
   - Un número (entero), formado por la concatenación de los índices $i$, $j$ del último elemento de la fila

   Así expresado parece un poco complicado, pero con un ejemplo verás que es muy fácil: dando como entrada por ejemplo la cadena $a$, el número 20 y el número 24, la función tendría que devolver la lista de cadenas `[ 'a20', 'a21', 'a22', 'a23', 'a24' ]`. Insertando varias de estas listas en otra lista, tendremos una matriz.

3. Escribe una función llamada *a_cadena* que reciba una matriz con este formato y devuelva una cadena de texto que represente esta matriz, sin corchetes, comas ni comillas. Los elementos de una fila estarán separados por un espacio. Entre una fila y otra habrá un salto de línea (`'\n'`). Ejemplo: si recibe como entrada esta matriz

   ```
   [
     [ 'a10', 'a11', 'a12', 'a13', 'a14' ],
     [ 'a20', 'a21', 'a22', 'a23', 'a24' ],
     [ 'a30', 'a31', 'a32', 'a33', 'a34' ]
   ]
   ```

la función devolverá la cadena

```
a10 a11 a12 a13 a14
a20 a21 a22 a23 a24
a30 a31 a32 a33 a34
```

4. Cada una de estas funciones puede llamar a otras funciones, si lo crees conveniente.

5. En cualquier programa es muy importante comprobar que los parámetros que reciben las funciones son los correctos: número, tipo, rango, etc. Por razones de tiempo, aquí no es necesario que lo hagas. Pero indica en cada función en un comentario qué habría que comprobar. No *cómo*, solamente *qué*.

## Solución

```
'use strict'
function crea_fila(prefijo, i0, i_n){
// Habría que comprobar:
// Número de parámetros = 3
// Primer parámetro es una letra minúsculas
// Segundo parámetro: número entero entre 10 y 99
// Tercer parámetro: número entero entre 10 y 99
// Tercer parámetro >= segundo parámetro
// Segundo parámetro y tercer parámetro son de la misma fila
//     (misma decena, misma i)

    let rval = [];
    for (let i=i0; i <= i_n; ++i){
        let elemento = prefijo + String(i);
        rval.push(elemento)
    }
    return rval;
}

function a_cadena(m){
// Habría que comprobar:
// m es una lista
// Todos los elementos de m son filas
// Todas las filas tienen el mismo número de elementos
// Cada fila es una lista
// Cada elemento de la fila es una cadena
// Cada cadena tiene el formato aij, donde 'a' es
//    minúscula, ij son dígitos
    let cadena = ""
    for (let fila of m){
        for (let elemento of fila){
            cadena = cadena + elemento + " ";
        }
        cadena = cadena + "\n"
    }
    return cadena;
}


let f1 = crea_fila('a', 10, 14);
let f2 = crea_fila('a', 20, 24);
let f3 = crea_fila('a', 30, 34);
let m = [f1,f2,f3]
console.log(m)

// Resultado:
// [
//   [ 'a10', 'a11', 'a12', 'a13', 'a14' ],
//   [ 'a20', 'a21', 'a22', 'a23', 'a24' ],
//   [ 'a30', 'a31', 'a32', 'a33', 'a34' ]
// ]
```

```
console.log(a_cadena(m))

// Resultado:

//a10 a11 a12 a13 a14
//a20 a21 a22 a23 a24
//a30 a31 a32 a33 a34
```

## Ejercicio 2. (6.5 puntos)

Modifica los ficheros `~/dat.junio.23/figuras.TULOGIN.html` y `~/dat.junio.23/figuras.TULOGIN.js` según esta especificación:

- En modo gráfico, el ejercicio tiene que seguir funcionando como hasta ahora. Para ello, copia el directorio donde tuvieras las imágenes de esta práctica al directorio del examen (`~/dat.junio.23`)
- Hasta ahora, el usuario podía marcar o desmarcar una figura haciendo *clic* sobre ella. Esto provocaba que o bien se mostrase la figura normalmente, o bien una imagen de reverso o similar. Cuando el usuario marcaba una carta o similar en modo texto, no sucedía nada.
- Modifica tu práctica para que las figuras en modo texto tengan un comportamiento similar al de las figuras en modo gráfico: al hacer clic sobre una, su texto será reemplazado por una serie de asteriscos, tantos como letras hubiera originalmente. Al volver a hacer clic, el texto volverá a su estado original. Las figuras en modo texto seleccionadas aparecerán en la misma estructura de datos que las figuras en modo gráfico. En esta estructura de datos no se distinguirán las figuras en modo texto o en modo gráfico.

Ejemplo: al hacer clic sobre *3 de corazones* el texto cambiará a ********** y el 3 de corazones aparecerá en la estructura que almacena las figuras seleccionadas. Al hacer clic sobre **********, el texto volverá a ser *3 de corazones* y esta figura desaparecerá de la estructura de figuras seleccionadas.