Unidad 8
Errores, módulos y API

CESUR
Tu Centro Oficial de FP

1. Errores

ERRORES – comportamientos anómalos de la aplicación que no estaban previstos.

Tipos de errores:

1) **Errores en tiempo de desarrollo** – se cometen al escribir mal una instrucción. Los suele detectar el editor de código.

2) **Errores en tiempo de ejecución** – se producen una vez que se ejecuta la aplicación. Podrían ser: identificadores no definidos previamente, operaciones matemáticas incorrectas, conexiones no disponibles, interbloqueos, etc.

En una aplicación web se pueden dar tres tipos de errores:

I. **Error** – errores que provocan que la aplicación deje de funcionar. Es un error no controlado que genera una serie de fallos que se van acumulando hasta que la aplicación se detiene.

II. **Exception** – una excepción es un error para el que se ha preparado un gestor que lo administra. Se debe que se puede dar en ciertas situaciones y en vez de evitarlo, se prepara una gestión del error para que esté controlado y no afecte significativamente a la ejecución de la aplicación.

III. **Warning** – los avisos son los errores menos importantes. No provocan que la aplicación deje de funcionar, pero notifican que se están produciendo situaciones que podrían terminar generando un error fatal. Deben tenerse en cuenta e intentar averiguar por qué se están produciendo para intervenir y que la situación empeore.

1. Errores

CREACIÓN Y LANZAMIENTO DE ERRORES

En JS los errores en tiempo de ejecución provocan la creación y lanzamiento automático de objetos Error. Aunque también se puede utilizar esta infraestructura de objetos para crear y lanzar errores propios.

```
let error = new Error("Formato de fecha incorrecto");
throw error;
```

La primera instrucción crea el objeto **Error** donde se almacenará toda la información sobre dicho error.

Con **throw** se lanza el error.

El resultado que se muestra por consola es:

1. Errores

Para errores específicos podemos utilizar los siguientes constructores:

Constructor	Utilidad
EvalError	Error lanzado al utilizar la función **eval()**, que evalúa una expresión y retorna su valor.
InternalError*	Error interno en el motor de JavaScript, por ejemplo: demasiada recursividad, demasiados paréntesis en una expresión regular, demasiadas opciones en un **switch**…
RangeError	Error lanzado cuando una variable numérica o parámetro toma un valor que está fuera de sus valores establecidos.
ReferenceError	Error que se produce cuando se realiza una referencia a objeto no válida.
SyntaxError	Error producido cuando JavaScript intenta interpretar código que no es válido sintácticamente.
TypeError	Error lanzado cuando el tipo de una variable o parámetro no es válido.
URIError	Error que se produce al codificar **encodeURI()** o al decodificar **decodeURI()** un URI.

* Es un constructor no estándar. No se debe usar en sitios web en producción porque no funcionará a todos los usuarios.

1. Errores

Existen dos propiedades estándar para acceder a la información concreta de un error:

- **message** – mensaje del error
- **name** – nombre del error

```
let error = new Error("Formato de fecha incorrecto");
console.log(`Mensaje del error: ${error.message}`);
console.log(`Nombre del error: ${error.name}`);
```

NOTA: navegadores como Mozilla o Microsoft proporcionan otras propiedades adicionales para la gestión de errores, pero no son generales a todos.

1. Errores

GESTIÓN DE EXCEPCIONES

En JS la gestión de excepciones se realiza con la declaración **try…catch.**

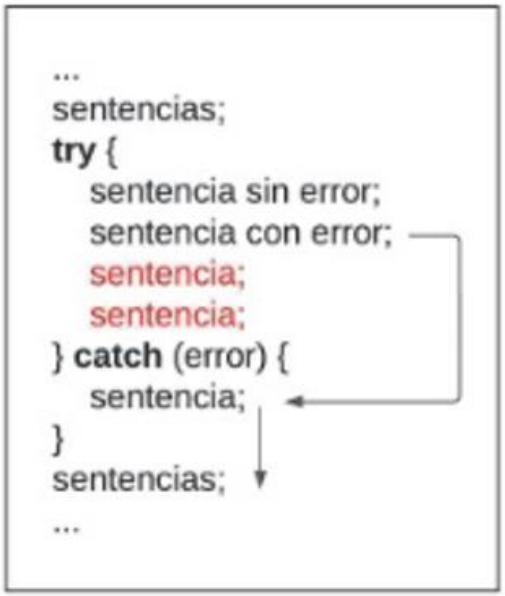
En el bloque **try** se colocan las instrucciones que podrían generar un error.

En el bloque **catch** se colocan las declaraciones que especifican qué hacer si se llega a producir un error.

Si alguna instrucción dentro del bloque **try** lanza una excepción, el control del programa salta automáticamente al bloque **catch**.

Cuando termine la ejecución del bloque **catch**, no se seguirán ejecutando las instrucciones que quedaron pendientes del bloque **try**.

```
try {
        throw "Error inducido";
} catch (error) {
        console.log(error);
}
```



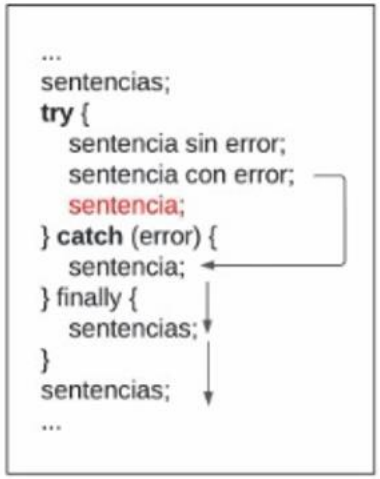
1. Errores

Existe otra variante que es la estructura **try…catch…finally**.

El bloque finally contiene instrucciones que se ejecutarán después de que se ejecuten los bloques try y catch.

El bloque finally se ejecutará siempre, tanto si se ha lanzado el error o no.

Ejecución del bloque try…catch…finally con error en el bloque try



Ejecución del bloque try…catch…finally sin error en el bloque try

La utilidad del bloque finally por ejemplo es: liberar un recurso que el programa haya bloqueado. Si se produce un error finally lo cierra y así lo libera, esto evita que el programa se detenga.

1. Errores

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <script>
        do{
            var n = prompt('Introduce un número del 0 al 10');
            function compruebaNota(n){
                if (n<0 || n>10){
                    throw new RangeError("Valor incorrecto para la nota.");
                }else {
                    document.write('Valor correcto, la nota es: '+ n);
                }
            }
            try{
                compruebaNota(n);
            } catch (error){
                if (error instanceof RangeError){
                    alert('ERROR: '+`${error.name}:${error.message}`
                    +" Por favor, vuelve a introducir un valor para la nota.");
                };
            };
        }while (n<0 || n>10);
    </script>
  </body>
</html>
```

Ejemplo 1: FUERA DE RANGO

Escribe un programa que comprueba el valor de una variable numérica entera llamada nota y lance un error si su valor no se encuentra entre 0 y 10.

2. Módulos

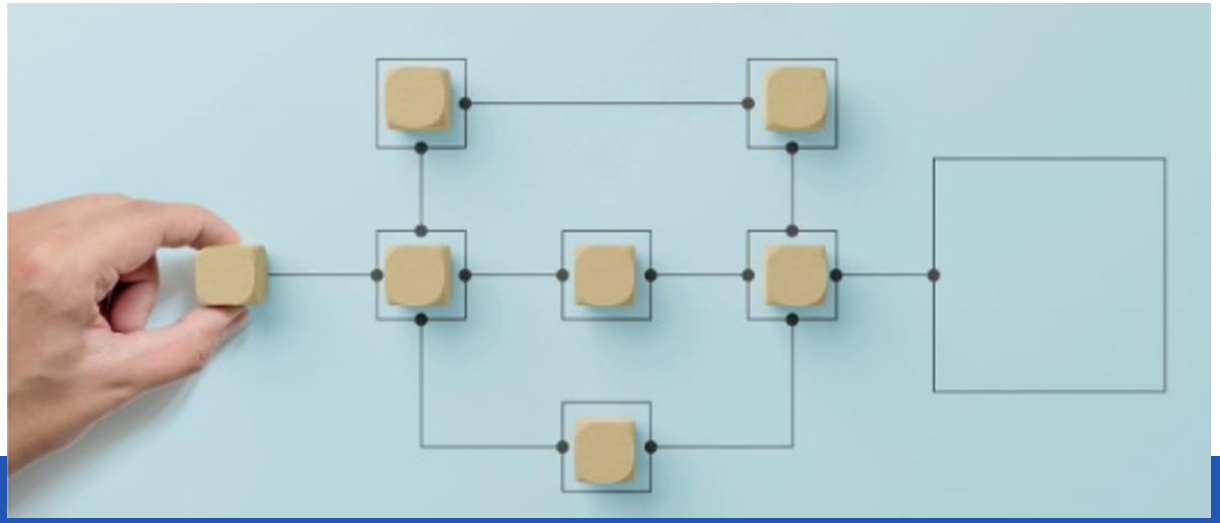
MÓDULO – conjunto de objetos, funciones, constantes, etc. que pueden usarse como una librería y que pueden reutilizarse para agilizar el desarrollo de aplicaciones, es decir, es un archivo de JS que agrupa funciones, clases, variables, etc. Que luego pueden ser exportadas y utilizadas en otras partes de nuestra aplicación.

Un módulo permite ocultar funcionalidad del mismo y sólo exportar aquello para lo que ha sido implementado.

El uso de módulos mejora la reutilización del código y permite tener aplicaciones más rápidas, más eficientes y mucho más ordenadas.

Con **import** y **export**, se pueden crear y cargar módulos propios.

Al importar módulos, si se escribe el código JS en el contenido de la etiqueta script de HTML, debe indicarse con **type="module"**, en lugar de **type="text/javascript"**



2. Módulos

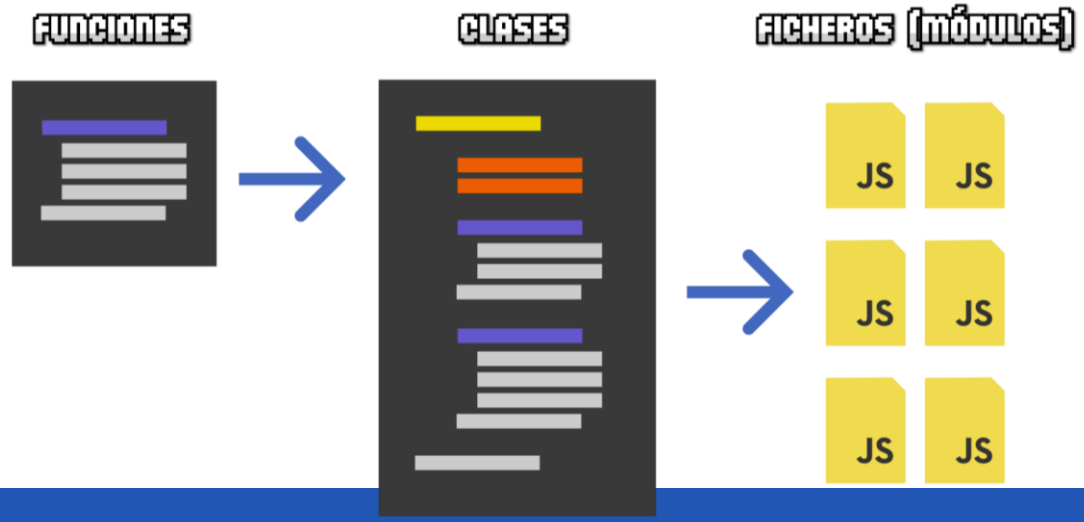
Vamos a crear el módulo modulo1.js:

```
export function sumar(x, y) {
    return x + y;
}

export function restar(x, y) {
    return x - y;
}
```

Este módulo consta de 2 funciones exportadas (sumar y restar).

El módulo podría estar formado por otras funciones, clases, variables, etc. pero sin la palabra **export**, a esos elementos solo se pueden acceder desde dentro del módulo.



2. Módulos

Para poder utilizar un módulo desde otro módulo o desde una página HTML, debemos utilizar la palabra clave **import**:

Después de import indicamos entre llaves los nombres de las funciones, clases, variables, etc. que importamos, también debemos indicar entre comillas el **nombre del módulo** y el path o dirección donde se almacena.

NOTA: utilizamos ./ para indicar que está en la misma carpeta.

Una vez importados los recursos del paquete los podemos **acceder** por su nombre.

```
<!DOCTYPE html>
<html>

<head>
    <title>Ejemplo de JavaScript</title>
    <meta charset="UTF-8">
</head>

<body>

    <script type="module">

        import {sumar, restar} from './modulo1.js';
        alert(sumar(3,8));
        alert(restar(10,3));

    </script>
</body>

</html>
```

Para indicarle al navegador que estamos utilizando módulos, lo hacemos asignando a la propiedad **type** el valor 'module'.

2. Módulos

Para exportar recursos de un módulo podemos utilizar otra sintaxis:

```
function sumar(x,y) {
    return x+y;
}

function restar(x,y) {
    return x-y;
}

export {sumar, restar};
```

Define dos funciones privadas al módulo y luego mediante la palabra clave **export** y entre llaves indicamos los nombres de las funciones a exportar.

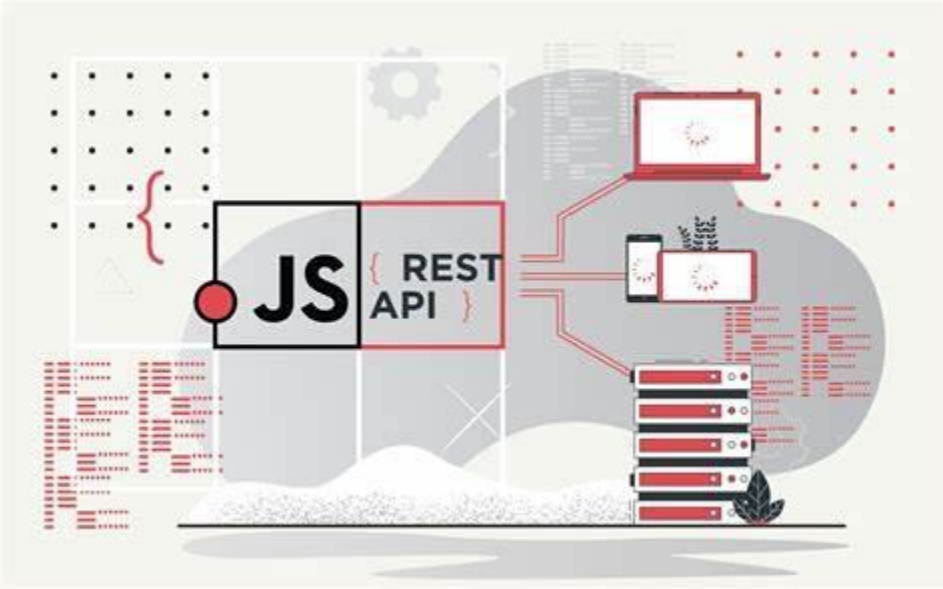
Export e Import
https://es.javascript.info/import-export

3. API

API – interfaz de programación de aplicaciones. Se trata de un conjunto de objetos, propiedades, métodos, funciones y otros componentes que se utilizan para desarrollar aplicaciones web permitiendo la comunicación entre dos aplicaciones siguiendo un conjunto de reglas. Es como un módulo externo de software que se comunica o interactúa con otro para conseguir ciertos objetivos.

Por ejemplo: al registrarse o iniciar sesión en muchos sitios web, se utilizan las credenciales de Google, Facebook, Instagram, etc. Lo que hace la aplicación web en la que se quiere iniciar sesión es, conectarse en segundo plano con las API de esos servicios para identificar al usuario, sin necesidad de registrar a dicho usuario.

El DOM, es una API que ofrece objetos como window o document y propiedades y métodos que permiten gestionar los elementos del DOM.



3. API

Algunas de las APIs que proporciona HTML5 son:

- **API Canvas:** ofrece funcionalidad para dibujar en 2D en un lienzo de HTML5.
- **Application Cache:** muy útil para que las aplicaciones web puedan trabajar *offline*.
- **Battery Status:** información detallada de la batería del dispositivo.
- **Drag & Drop:** amplia funcionalidad para trabajar con elementos arrastrables.
- **FileSystem API:** trabajo con archivos locales usando JavaScript.
- **FullScreen:** utilidades para poder gestionar la aplicación cuando se visualiza a pantalla completa.
- **Geolocation:** gestión de las coordenadas de localización del usuario.
- **Media API:** funcionalidad extendida para gestionar contenido de audio y vídeo.
- **Text Track API:** maneja los subtítulos de componentes de audio y vídeo.
- **Web GL:** amplía la funcionalidad del *canvas* para trabajar con 3D incorporando elementos de Open GL.
- **Web Sockets:** gestiona los *sockets* de una red para facilitar la comunicación cliente-servidor.
- **Web Storage:** permite esquivar las *cookies* almacenando datos en el navegador.
- **Web Workers:** ofrece la posibilidad de realizar procesamiento en segundo plano para evitar problemas de rendimiento en el primer plano.

APIs
https://developer.mozilla.org/es/docs/Learn/JavaScript/Client-side_web_APIs/Introduction

4. API Canvas

<canvas> es un elemento HTML que puede ser usado para dibujar gráficos usando JS. Permite dibujar gráficos, realizar composición de fotos e incluso crear animaciones.

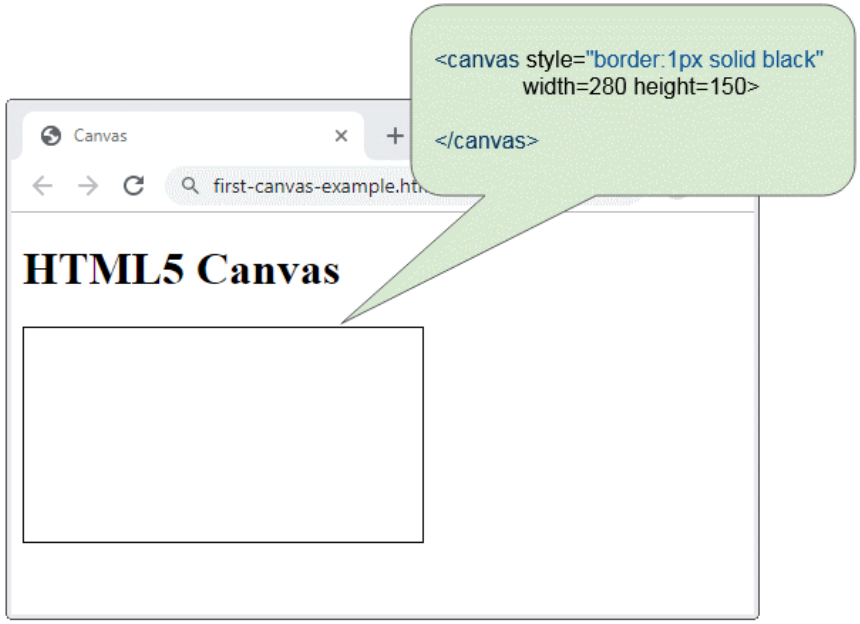
NOTA: canvas sólo está soportado por las versiones más recientes de los navegadores.

El elemento canvas sólo tiene dos atributos opcionales: **width** y **height**. Si no se especifica nada, el lienzo se inicializará con 300 px de ancho y 150 px de alto.

El canvas está inicialmente en blanco, para mostrar algo, el script necesita primero acceder al contexto a renderizar y dibujar sobre este. Para ello canvas tiene el método **getContext()** que se usa para obtener el contexto a renderizar y sus funciones de dibujo.

NOTA: para gráficos 2D, getContext es “2d”.

Al dibujar en canvas se parte de la esquina superior izquierda, en las coordenadas (0,0)



Tutorial de Canvas
https://developer.mozilla.org/es/docs/Web/API/Canvas_API/Tutorial

4. API Canvas

Canvas sólo admite dos formas primitivas: **rectángulos** y **trazados** (listas de puntos conectados por líneas). Todas las formas deben crearse combinando uno o más caminos.

Existen tres funciones para **dibujar rectángulos**:

`fillRect(x, y, width, height)`

Dibuja un rectángulo relleno.

`strokeRect(x, y, width, height)`

Dibuja un contorno rectangular.

`clearRect(x, y, width, height)`

Borra el área rectangular especificada, haciéndola completamente transparente.

Cada una de estas funciones toma los mismos parámetros x, y que especifican la posición en el lienzo relativa a la esquina superior izquierda.

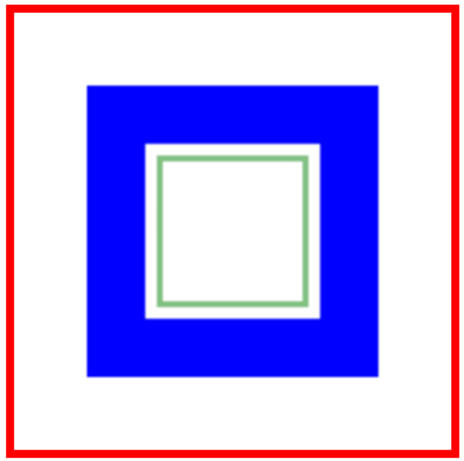
Width y height proporcionan el tamaño del rectángulo.

4. API Canvas

Ejemplo 2: RECTÁNGULOS

Crea un programa, usando la API Canvas, que dibuje lo siguiente:

1) Delimite la zona canvas en rojo y 2 px
2) Dibuje un rectángulo azul de 100 px de lado
3) Borre un rectángulo de 60 px de lado dentro del anterior
4) Dibuje un contorno rectangular verde de 50 px dentro del anterior.



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <style>
      canvas {border: 3px solid red;}
    </style>
    <script>
      function draw() {
        const canvas = document.getElementById("canvas");
        if (canvas.getContext) {
          const ctx = canvas.getContext("2d");
          /*dibuja un cuadrado de 100px de lado
          de color azul*/
          ctx.fillStyle = "blue";
          ctx.fillRect(25, 25, 100, 100);
          //borra un cuadrado de 60px del centro
          ctx.clearRect(45, 45, 60, 60);
          /*crea un contorno rectangunar de 50px dentro
          del cuadrado borrado y de color verde*/
          ctx.strokeStyle = "green";
          ctx.strokeRect(50, 50, 50, 50);
        }
      }
    </script>
  </head>
  <body onload="draw()">
    <canvas id="canvas" width="150" height="150"></canvas>
  </body>
</html>
```

4. API Canvas

Para **dibujar trazados** en canvas:

1) **CREAR EL CAMINO** con **beginPath() –** Crea un nuevo trazado y una vez creado los futuros comandos de dibujo se dirigen al trazado y se utilizan para construirlo.

   Internamente los trazados se almacenan como una lista de sub-trazados (líneas, arcos, etc.) que juntos, forman una forma.

   Cada vez que se llama a **beginPath()** la lista se restablece y podemos empezar a dibujar nuevas formas.

2) Utilizar los comandos de dibujo para **DIBUJAR EL TRAZADO path**

   - **closePath() –** añade una línea recta, que va al inicio del sub-trazado actual.
   - **stroke() –** dibuja la forma trazando su contorno.
   - **fill()** – dibuja una forma sólida rellenando el área de contenido del trazado.

3) Opcionalmente se puede llamar a **closePath()**. Este método intenta **CERRAR LA FORMA** dibujando una línea recta desde el punto actual hasta el inicio. Si la forma ya ha sido cerrada o sólo hay un punto en la lista, esta función no hace nada.

   NOTA: cuando se llama a **fill()** cualquier forma abierta se cierra automáticamente, por tanto, no es necesario llamar a **closePath()**

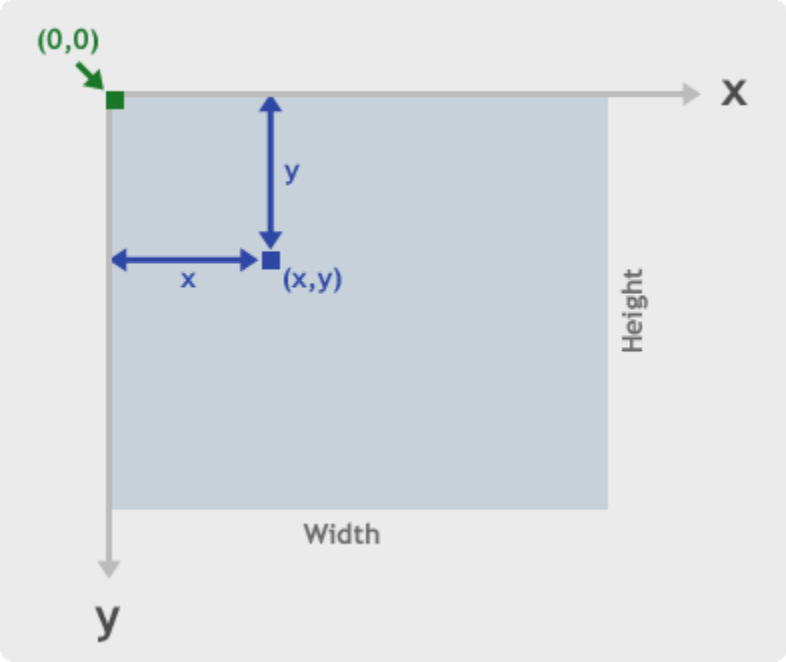
   moveTo(x, y) - mueve la pluma a las coordenadas x e y. En canvas se utiliza para colocar el punto de partida en otro lugar. También se utiliza para dibujar trazados no conectados.

DIBUJAR LÍNEAS

lineTo(x, y) – dibuja una línea recta desde la posición actual de dibujo hasta la posición especificada por x e y.

El punto final nos lo dan las coordenadas x e y.

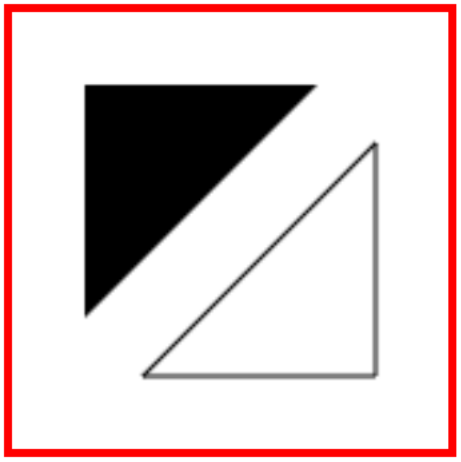
El punto de partida depende de los trazados anteriores o puede cambiarse utilizando el método moveTo().



CESUR
Tu Centro Oficial de FP

Ejemplo 3: FLECHAS

Crea un programa, usando la API Canvas, que dibuje dos triángulos, uno contorneado y otro relleno como los de la figura:

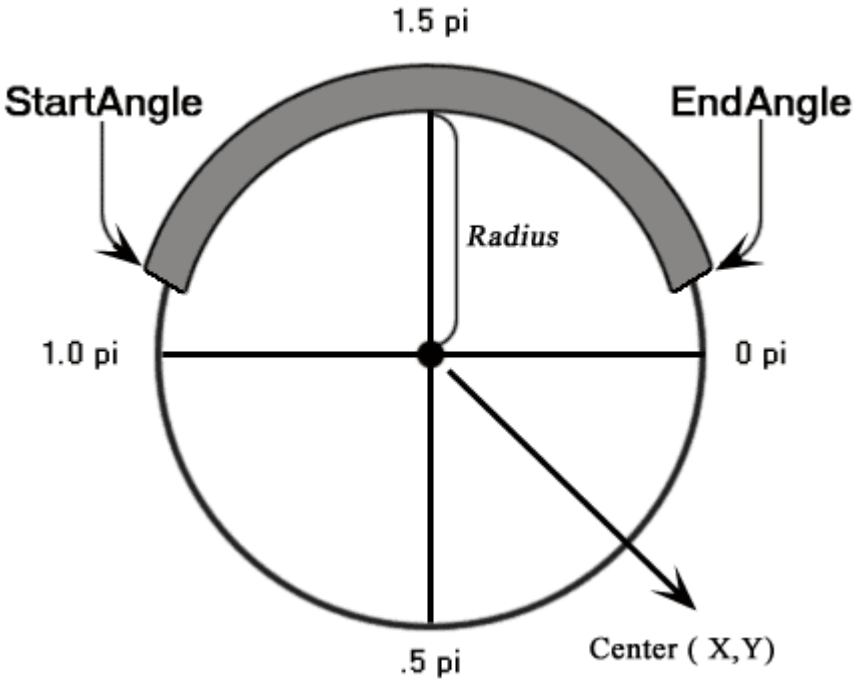


```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <style>
      canvas {border: 3px solid red;}
    </style>
    <script>
      function draw() {
        const canvas = document.getElementById("canvas");
        if (canvas.getContext) {
          const ctx = canvas.getContext("2d");
          // Triángulo relleno
          ctx.beginPath();
          ctx.moveTo(25, 25);
          ctx.lineTo(105, 25);
          ctx.lineTo(25, 105);
          ctx.fill();
          // Triángulo contorneado
          ctx.beginPath();
          ctx.moveTo(125, 125);
          ctx.lineTo(125, 45);
          ctx.lineTo(45, 125);
          ctx.closePath();
          ctx.stroke();
        }
      }
    </script>
  </head>
  <body onload="draw()">
    <canvas id="canvas" width="150" height="150"></canvas>
  </body>
</html>
```

DIBUJAR ARCOS

Para dibujar arcos o círculos utilizamos dos métodos:

- **arc(x, y, radius, startAngle, endAngle, counterclockwise**) – dibuja un arco centrado en la posición **(x,y)** con el radio **r** que comienza en **startAngle** y termina en **endAngle** yendo en la dirección indicada en **counterclockwise** (parámetro booleano, por defecto en el sentido de las agujas del reloj [false]).



NOTA: Los ángulos en la función **arc** se miden en radianes, no en grados.

Para convertir los grados en radianes puedes utilizar la siguiente expresión de JS:

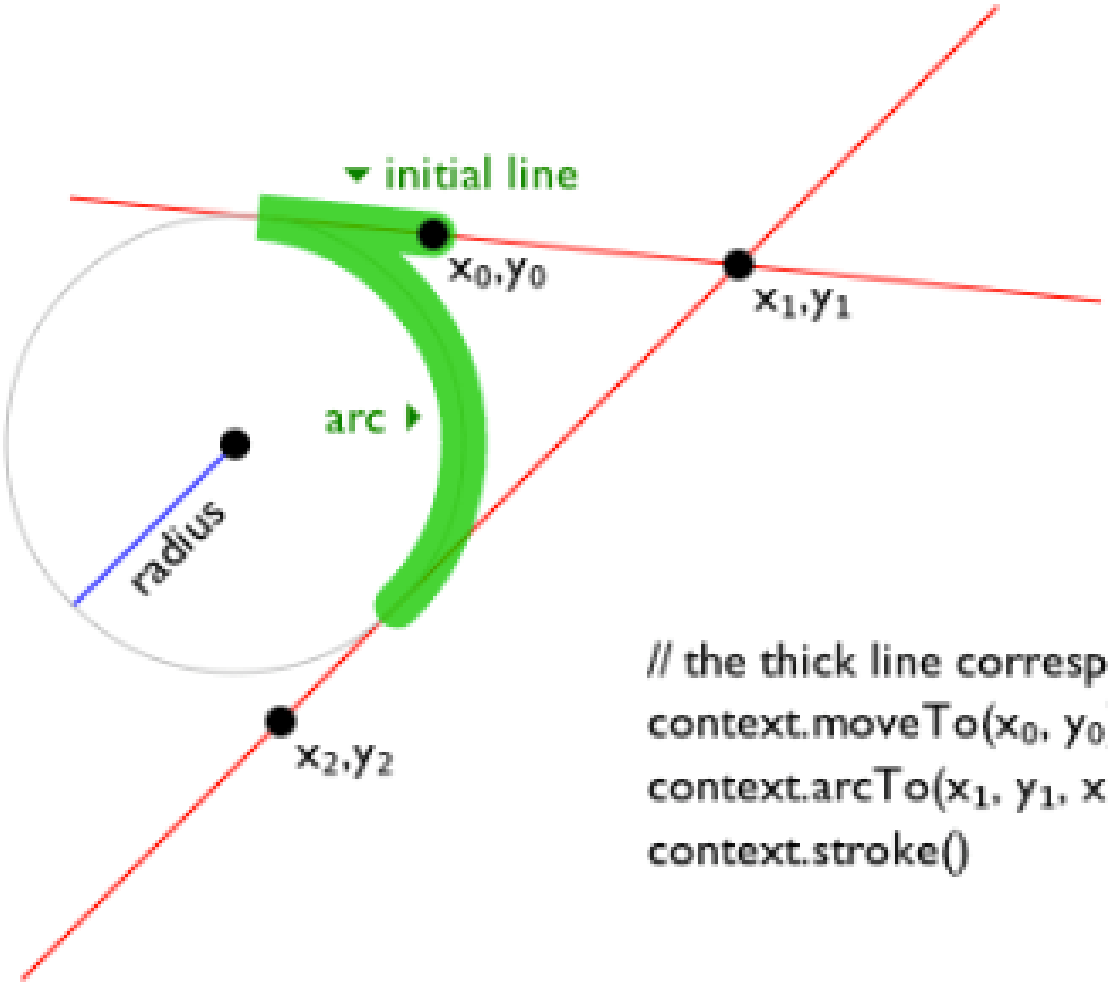
radianes = (Math.PI/180)*grados

4. API Canvas

- **arcTo(x1, y1, x2, y2, radius)** – dibuja un arco con los puntos de control y el radio dados, conectado al punto anterior por una línea recta.

Dibujar formas complejas en canvas

https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_shapes#bezier_and_quadratic_curves




```
// the thick line corresponds to:
context.moveTo(x0, y0)
context.arcTo(x1, y1, x2, y2, radius)
context.stroke()
```

CESUR
Tu Centro Oficial de FP

Ejemplo 4: SMILE

Crea un programa, usando la API Canvas, que dibuje una carita sonriente.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <style>
      canvas {border: 3px solid red;}
    </style>
    <script>
        function draw() {
            const canvas = document.getElementById("canvas");
            if (canvas.getContext) {
                const ctx = canvas.getContext("2d");
                ctx.beginPath();
                // Círculo externo
                ctx.arc(75, 75, 50, 0, Math.PI * 2, true);
                // Boca (en el sentido de las agujas del reloj)
                ctx.moveTo(110, 75);
                ctx.arc(75, 75, 35, 0, Math.PI, false);
                // Ojo izquierdo
                ctx.moveTo(65, 65);
                ctx.arc(60, 65, 5, 0, Math.PI * 2, true);
                // Ojo derecho
                ctx.moveTo(95, 65);
                ctx.arc(90, 65, 5, 0, Math.PI * 2, true);
                ctx.stroke();
            }
        }
    </script>
  </head>
  <body onload="draw()">
    <canvas id="canvas" width="150" height="150"></canvas>
  </body>
</html>
```

ESTILOS Y COLORES

Para aplicar colores utilizamos los métodos:

fillStyle = color – establece el estilo utilizado al rellenar las formas

stokeStyle = color – establece el estilo de los contornos de las formas

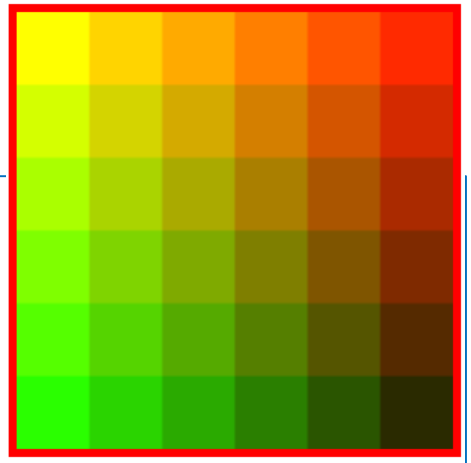
Por ejemplo, ada uno de los siguientes ejemplos describe el mismo color:

```
ctx.fillStyle = "orange";
ctx.fillStyle = "#FFA500";
ctx.fillStyle = "rgb(255 165 0)";
ctx.fillStyle = "rgb(255 165 0 / 100%)";
```

4. API Canvas

Ejemplo 5: COLORES

Crea un programa, usando la API Canvas, que dibuje una cuadrícula de rectángulos, cada uno de un color diferente.



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <style>
      canvas {border: 3px solid red;}
    </style>
    <script>
        function draw() {
            const ctx = document.getElementById("canvas").getContext("2d");
            for (let i = 0; i < 6; i++) {
                for (let j = 0; j < 6; j++) {
                ctx.fillStyle = `rgb(${Math.floor(255 - 42.5 * i)} ${Math.floor(255 - 42.5 * j,)} 0)`;
                ctx.fillRect(j * 25, i * 25, 25, 25);
                }
            }
        }
    </script>
  </head>
  <body onload="draw()">
    <canvas id="canvas" width="150" height="150"></canvas>
  </body>
</html>
```

Se utilizan dos bucles for para dibujar una cuadrícula de rectángulos, cada uno de un color diferente. Usamos dos variables i y j para generar un color RGB único para cada cuadrado y sólo modificamos el rojo y el verde, el canal azul tiene un valor fijo.

5. API Web Storage

Esta API permite almacenar datos en la máquina del usuario. Las ventajas que tiene respecto a las cookies son:

- Menos restricciones
- Los datos no se envían al servidor
- No se incluyen en ninguna petición/respuesta HTTP
- Permiten almacenar varios Gb

El objeto window ofrece dos nuevas propiedades:

1. **localStorage** – para almacenar los datos
2. **sessionStorage** – almacena datos con gestión de sesión. Estos datos se eliminan tan pronto como se acaba la sesión

API Almacenamiento web
https://developer.mozilla.org/es/docs/Web/API/Web_Storage_API

6. API Geolocation

Esta API permite conocer cierta información geográfica del dispositivo del usuario. Por privacidad, se pide al usuario que confirme el permiso para proporcionar información de ubicación.

getCurrentPosition – nos permite conocer la posición del usuario y facilita la longitud y latitud del dispositivo. Recibe un objeto **Position**.

watchPositon – se ejecuta constantemente para conocer variaciones en la posición y poder hacer un seguimiento del dispositivo.

Otras propiedades del objeto Position son:

Propiedad	Utilidad
accuracy	Precisión del cálculo de la longitud y la latitud (en metros).
altitude	Altura sobre el nivel del mar.
altitudeAccuracy	Precisión del cálculo de la altitud (en metros).
heading	Ángulo de movimiento del dispositivo del usuario.
speed	Velocidad de movimiento del dispositivo del usuario (en m/s).

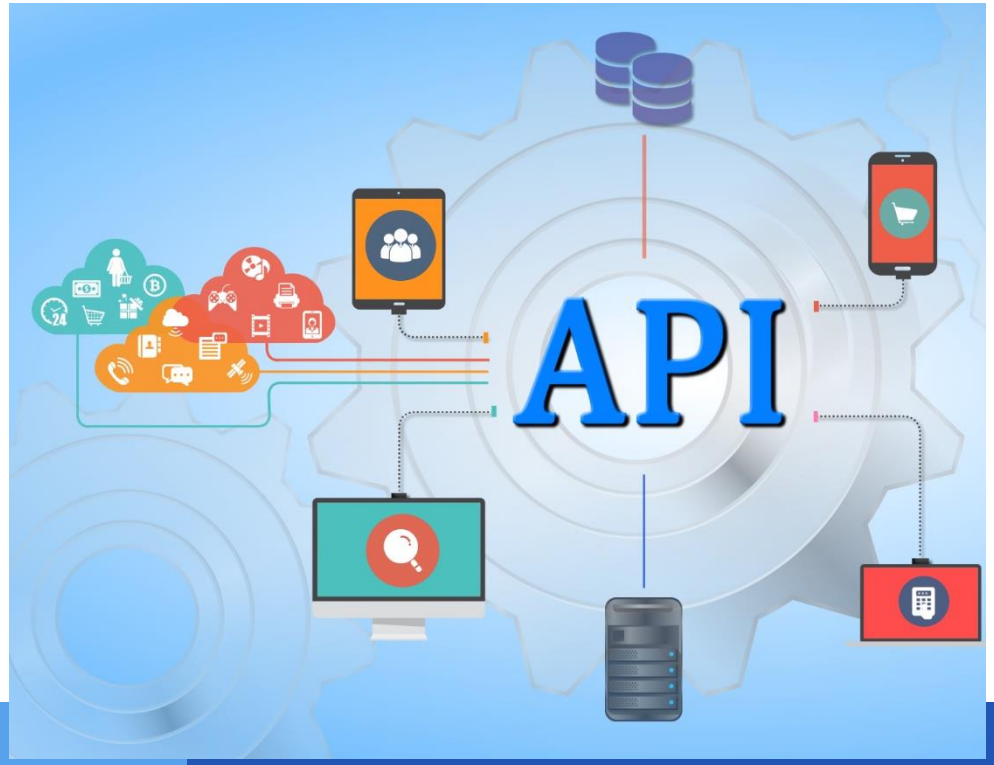
API de geolocalización
https://developer.mozilla.org/es/docs/Web/API/Geolocation_API

7. API Notification

Objeto que gestiona las notificaciones.

requestPermission() – método estático que se utiliza para solicitar permiso al usuario. La respuesta será granted (aceptado) o denied (rechazado).

```
Notification.requestPermission().then(respuesta=>{
        if (respuesta == "granted")
                new Notification("¡Gracias!");
});
```



API de Notificación
https://developer.mozilla.org/es/docs/Web/API/notification

Listado APIs Web de JS
https://developer.mozilla.org/es/docs/Web/API

Gracias