**INTRODUCCIÓN A LA TERMINAL**

**Uso de comando básicos git bash**

1. **git init**: Inicia un nuevo repositorio Git en el directorio actual.
2. **git clone [url]**: Clona un repositorio existente desde una URL remota a tu máquina local.
3. **git add [archivo(s)]**: Agrega cambios al área de preparación para ser incluidos en el próximo commit.
4. **git commit -m "[mensaje]"**: Crea un nuevo commit con los cambios que están en el área de preparación, junto con un mensaje que describe los cambios realizados.
5. **git status**: Muestra el estado actual del repositorio, incluyendo los archivos modificados, los archivos en el área de preparación y los archivos sin seguimiento.
6. **git pull**: Obtiene los cambios desde un repositorio remoto y los fusiona con tu rama local.
7. **git push**: Envía tus commits locales a un repositorio remoto.
8. **git branch**: Lista todas las ramas locales en el repositorio.
9. **git checkout [rama]**: Cambia a una rama específica.
10. **git merge [rama]**: Fusiona una rama específica con la rama actual.
11. **git remote -v**: Muestra las URL de los repositorios remotos asociados con el repositorio local.
12. **git log**: Muestra un registro de commits, incluyendo detalles como el autor, la fecha y el mensaje del commit.
13. **git reset [archivo(s)]**: Deshace los cambios en el área de preparación, pero conserva los cambios en el directorio de trabajo.
14. **git revert [commit]**: Deshace un commit específico creando un nuevo commit que revierte los cambios realizados en el commit especificado.
15. **git stash**: Guarda temporalmente los cambios en un área de trabajo sin comprometerlos en un commit.
16. **git remote add [nombre] [url]**: Agrega un nuevo repositorio remoto con un nombre específico y una URL.
17. **git remote remove [nombre]**: Elimina un repositorio remoto del repositorio local.
18. **git diff**: Muestra las diferencias entre los cambios realizados en el directorio de trabajo y el área de preparación.
19. **git log --graph**: Muestra un registro de commits en forma de gráfico, mostrando las relaciones entre las ramas.
20. **git fetch**: Descarga cambios desde un repositorio remoto sin fusionarlos en tu rama local.
21. **git checkout -b [nombre]**: Crea una nueva rama y cambia a ella en un solo paso.
22. **git branch -d [nombre]**: Elimina una rama local después de fusionar sus cambios.
23. **git branch -D [nombre]**: Elimina una rama local incluso si no se han fusionado sus cambios.
24. **git tag**: Lista todas las etiquetas disponibles en el repositorio.
25. **git tag [nombre]**: Crea una nueva etiqueta en el commit actual.
26. **git tag -d [nombre]**: Elimina una etiqueta específica.
27. **git blame [archivo]**: Muestra quién modificó cada línea de un archivo y en qué commit se realizó la modificación.
28. **git config**: Permite configurar opciones de Git, como el nombre de usuario y el correo electrónico.
29. **git checkout -- [archivo(s)]**: Deshace los cambios locales en el archivo especificado y lo restaura a su estado en el último commit.
30. **git cherry-pick [commit]**: Aplica los cambios introducidos por un commit específico a la rama actual.
31. **git rebase [rama]**: Reorganiza la historia de los commits para que la rama actual esté basada en la última confirmación de la rama especificada.
32. **git rebase -i [commit]**: Realiza una reorganización interactiva de la historia de los commits, lo que te permite fusionar, eliminar o reorganizar commits antes de aplicarlos.
33. **git submodule**: Gestiona submódulos dentro de un repositorio Git principal.
34. **git clean**: Elimina archivos no rastreados del directorio de trabajo.
35. **git grep [patrón]**: Busca archivos en el árbol de trabajo que coincidan con un patrón específico.
36. **git reflog**: Muestra un registro de referencia de los cambios realizados en el repositorio, útil para recuperar commits eliminados o deshacer acciones.
37. **git config --global alias.[nombre] "[comando]"**: Crea un alias para un comando Git, lo que te permite ejecutar comandos más complejos con un nombre corto.
38. **git bisect**: Ayuda a encontrar el commit que introdujo un error, realizando una búsqueda binaria entre dos commits conocidos.
39. **git archive**: Crea un archivo comprimido que contiene el contenido de un árbol de trabajo o un commit específico.
40. **git revert --abort**: Cancela una operación de revertir en curso.
41. **git rm [archivo(s)]**: Elimina archivos del directorio de trabajo y del área de preparación. También puede usar la opción **-r** para eliminar directorios de forma recursiva.

**Comandos básicos de la terminal**

1. **ls**: Lista los archivos y directorios en el directorio actual.
2. **cd**: Cambia el directorio.
3. **pwd**: Muestra la ruta del directorio actual.
4. **mkdir**: Crea un nuevo directorio.
5. **rm**: Elimina archivos o directorios.
6. **cp**: Copia archivos o directorios.
7. **mv**: Mueve archivos o directorios.
8. **touch**: Crea un nuevo archivo vacío.
9. **cat**: Muestra el contenido de un archivo.
10. **echo**: Muestra texto en la terminal o lo redirige a un archivo.
11. **grep**: Busca patrones en archivos.
12. **chmod**: Cambia los permisos de archivos o directorios.
13. **chown**: Cambia el propietario de un archivo o directorio.
14. **man**: Muestra el manual de un comando.
15. **history**: Muestra el historial de comandos utilizados.

**REPOSITORIO LOCAL (GITBASH) A GITHUB**

1. **Crear un nuevo directorio para tu proyecto**: Crea un nuevo directorio en tu computadora donde desees almacenar tu proyecto.
2. **Inicializar un repositorio Git en el directorio**: Navega al directorio del proyecto en tu terminal y ejecuta: **git init**
3. **Git status**
4. **Agregar archivos al repositorio**: Coloca tus archivos de proyecto en el directorio del repositorio y ejecuta: **git add** (agregar el archivo NO la carpeta!)
5. **Hacer el primer commit**: Haz tu primer commit con los archivos agregados: **git commit -m "Primer commit"**
6. **Crear un repositorio en GitHub**: Abre tu navegador web, inicia sesión en GitHub (o regístrate) y crea un nuevo repositorio.
7. **Conectar repositorio local con GitHub**: En tu terminal, ejecuta: **git remote add origin https://github.com/nombre\_usuario/nombre\_repositorio.git**
8. **Subir cambios a GitHub**: Finalmente, sube tus cambios al repositorio en GitHub: **git push -u origin master** (Si estás utilizando una rama diferente a **master**, reemplaza **master** con el nombre de tu rama).

**ARCHIVO README (GITHUB) A REPOSITORIO LOCAL (GITBASH)**

1. Abrir Git Bash.
2. Navegar a la ubicación deseada utilizando **cd**.
3. Clonar el repositorio de GitHub con el comando **git clone URL\_DEL\_REPOSITORIO**, donde **URL\_DEL\_REPOSITORIO** es la URL del repositorio que deseas clonar.
4. Una vez completado, el repositorio, incluyendo el archivo README, estará clonado en tu directorio actual.
5. Puedes abrir el archivo README en un editor de texto para ver su contenido y modificarlo.
6. Para que luego los cambios se suban a GitHub usamos **git add (nombre del archivo)**.
7. Agregamos el commit **git commit –m (ponemos el mensaje)**.
8. **Git push** y se sube a la nube.

**ARCHIVO NO README (GITHUB) A REPOSITORIO LOCAL (GITBASH)**

1. Abrir Git Bash.
2. Navegar a la ubicación deseada utilizando el comando **cd**.
3. Luego **git remote add origin (código del archivo)**.
4. **Git brunch –m main**.
5. **Git push –u origin main**.
6. Actualizamos para ver los cambios.

**HTML**

HTML, siglas de **HyperText Markup Language**, es el lenguaje de marcado estándar utilizado para crear y diseñar páginas web. Proporciona la estructura básica y el contenido de una página web, permitiendo definir diferentes elementos y su disposición en la pantalla. En resumen, HTML es el lenguaje fundamental utilizado en el desarrollo web para crear la estructura y el contenido de las páginas web. Junto con CSS (Cascading Style Sheets) y JavaScript, forma la base de la mayoría de los sitios web modernos, permitiendo crear experiencias web interactivas, atractivas y funcionales.

**Estructura inicial**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Título de la página</title>

</head>

<body>

<!-- Contenido de la página -->

</body>

</html>

1. **<!DOCTYPE html>**: Define el tipo de documento y la versión de HTML utilizada (generalmente HTML5).
2. **<html>**: Engloba todo el contenido HTML del documento.
3. **<head>**: Contiene metadatos y enlaces a recursos externos, como hojas de estilo CSS o scripts JavaScript.
4. **<meta charset="UTF-8">**: Define el juego de caracteres utilizado en el documento (generalmente UTF-8 para admitir caracteres internacionales).
5. **<title>**: Define el título del documento que se mostrará en la pestaña del navegador.
6. **</head>**: Cierra la sección de encabezado del documento.
7. **<body>**: Contiene todo el contenido visible de la página, como texto, imágenes, enlaces, etc.
8. **</body>**: Cierra la sección de cuerpo del documento.
9. **</html>**: Cierra el elemento raíz **<html>** que engloba todo el contenido del documento.

**Etiquetas de formato, bloque y contenido**

1. **<html>**: Define el comienzo y el final de un documento HTML.
2. **<head>**: Contiene información meta sobre el documento, como títulos, metadatos y enlaces a estilos y scripts.
3. **<title>**: Define el título de la página que se muestra en la barra de título del navegador.
4. **<body>**: Contiene el contenido visible de la página, como texto, imágenes, enlaces, etc.
5. **<h1>** a **<h6>**: Define encabezados de diferentes niveles, donde **<h1>** es el más importante y **<h6>** el menos importante.
6. **<p>**: Define un párrafo de texto.
7. **<a>**: Define un enlace (hipervínculo) a otra página web o recurso.
8. **<img>**: Inserta una imagen en la página.
9. **<div>**: Define una división o sección genérica de contenido, útil para aplicar estilos o estructurar la página.
10. **<span>**: Define un contenedor de texto en línea que se puede usar para aplicar estilos o manipular con scripts.
11. **<ul>**, **<ol>**, **<li>**: Define listas no ordenadas (viñetas), listas ordenadas (numeradas) y elementos de lista, respectivamente.
12. **<table>**, **<tr>**, **<td>**: Define una tabla, una fila en la tabla y una celda en la tabla, respectivamente.
13. **<form>**, **<input>**, **<button>**: Define un formulario, un campo de entrada en el formulario y un botón en el formulario, respectivamente.
14. **<br>**: Inserta un salto de línea (line break) dentro del texto.
15. **<hr>**: Inserta una línea horizontal para separar contenido.
16. **<meta>**: Define metadatos que proporcionan información sobre la página, como el juego de caracteres, la descripción y las palabras clave.
17. **<style>**: Define estilos CSS en línea dentro del documento HTML.
18. **<script>**: Define scripts JavaScript dentro del documento HTML.
19. **<header>**: Define el encabezado de una sección o página.
20. **<footer>**: Define el pie de una sección o página.
21. **<nav>**: Define una sección de navegación.
22. **<main>**: Define el contenido principal de una página.
23. **<article>**: Define un artículo independiente en una página.
24. **<section>**: Define una sección genérica en una página.
25. **<aside>**: Define contenido relacionado, como una barra lateral, que está separado del contenido principal pero relacionado con él.
26. **<figure>**: Define contenido multimedia, como imágenes, videos, gráficos, etc., junto con una leyenda **<figcaption>**.
27. **<figcaption>**: Define la leyenda de una figura, que describe el contenido multimedia.
28. **<blockquote>**: Define una cita larga o un bloque de texto citado.
29. **<cite>**: Define el título de la obra citada en un bloque de cita.
30. **<abbr>**: Define una abreviatura o un acrónimo y proporciona una explicación para el término abreviado.
31. **<code>**: Define texto de código fuente.
32. **<pre>**: Define texto preformateado, donde el formato y los espacios en blanco se conservan.
33. **<kbd>**: Define la entrada del teclado, como una combinación de teclas o un atajo de teclado.
34. **<samp>**: Define una salida de muestra, como la salida de un programa o un script.
35. **<mark>**: Define texto resaltado o marcado.
36. **<time>**: Define una fecha o hora, que puede ser formateada para su visualización.
37. **<audio>**: Inserta contenido de audio en la página.
38. **<video>**: Inserta contenido de video en la página.
39. **<iframe>**: Inserta un marco en línea para mostrar contenido de otra página web.
40. **<canvas>**: Define un área gráfica para dibujar gráficos, gráficos y otros gráficos dinámicos mediante JavaScript.
41. **<datalist>**: Define una lista de opciones predefinidas para los controles de entrada, como **<input type="text">**.
42. **<details>**: Define detalles adicionales que el usuario puede mostrar u ocultar.
43. **<summary>**: Define el resumen o título de los detalles desplegables.
44. **<fieldset>**: Agrupa elementos de formulario relacionados con un borde y un título opcional.
45. **<legend>**: Define el título del campo de conjunto de campos **<fieldset>**.
46. **<meter>**: Define una medida o un rango escalar dentro de un rango conocido, como una barra de progreso.
47. **<progress>**: Define una barra de progreso para mostrar el progreso de una tarea.
48. **<ruby>**, **<rt>**, **<rp>**: Utilizados para mostrar texto ruby, que es texto adicional que se muestra junto con el texto base, comúnmente utilizado en texto de idiomas asiáticos.
49. **<strong>:** Se utiliza para definir texto que tiene un énfasis fuerte o una importancia especial. Indica su importancia semántica.
50. **<b>:** indica que el texto debe mostrarse en negrita sin implicar necesariamente un énfasis importante. Le da forma visual al texto.
51. **<label>:** se utiliza para asociar texto descriptivo con elementos de formulario, como **<input>**, **<textarea>**, **<select>**, etc. Proporciona una etiqueta legible para humanos que describe la función del campo de entrada.
52. **Placeholder:** El atributo **placeholder** se utiliza en elementos de formulario, como **<input>** y **<textarea>**, para proporcionar un texto de marcador de posición que se muestra en el campo antes de que el usuario ingrese datos. Ayuda a los usuarios a comprender qué tipo de información se espera en el campo.
53. **<select>:** La etiqueta **<select>** se utiliza para crear un menú desplegable que permite al usuario seleccionar una opción de una lista predefinida. Dentro de la etiqueta **<select>**, se suelen incluir una o más etiquetas **<option>** que representan las opciones disponibles en el menú desplegable.
54. **<opción>:** La etiqueta **<option>** se utiliza dentro de una etiqueta **<select>** para definir una opción en un menú desplegable. Cada **<option>** puede tener un valor asociado utilizando el atributo **value**, que se enviará al enviar el formulario si se selecciona esa opción.
55. **<link>:** se utiliza para establecer una conexión entre el documento actual y un recurso externo, como una hoja de estilos CSS, un archivo de icono, un archivo de fuente o una relación de URL prefijada
56. **<i>:** Letra cursiva.
57. **<textarea>:** se utiliza para crear un área de texto multi línea en un formulario, donde los usuarios pueden ingresar y editar texto libremente. Es útil cuando se necesita que los usuarios proporcionen comentarios, escriban mensajes largos u otras formas de texto extenso.

**CSS**

Significa Cascading Style Sheets (Hojas de Estilo en Cascada), es un lenguaje de diseño utilizado para controlar el aspecto visual y la presentación de documentos HTML y XML. Permite definir cómo se muestran los elementos HTML en un navegador web, especificando propiedades como el color, la tipografía, el tamaño, el espaciado y el diseño de la página. La principal función de CSS es separar el contenido de la presentación, lo que significa que los estilos se definen en un archivo separado (archivo CSS) y se aplican a los elementos HTML a través de selectores. Esto permite una mayor flexibilidad y control sobre el diseño y la apariencia de un sitio web, facilitando la creación de diseños consistentes y atractivos.

**Selectores de clases**

**Definición:** Las clases en CSS se definen anteponiendo un punto (**.**) al nombre de la clase. Por ejemplo, **.mi-clase { ... }**.

**Uso:** Puedes asignar una clase a uno o varios elementos HTML utilizando el atributo **class**. Por ejemplo, **<div class="mi-clase"></div>**.

**Selección:** Para seleccionar elementos por clase en CSS, simplemente usa el nombre de la clase precedido por un punto (**.**). Por ejemplo, **.mi-clase { color: red; }**.

**Aplicación:** Los estilos definidos para una clase se aplicarán a todos los elementos HTML que tengan esa clase asignada. Por ejemplo, todos los elementos con la clase **mi-clase** tendrán texto rojo si se define **.mi-clase { color: red; }**.

**Reutilización:** Las clases pueden reutilizarse en múltiples elementos HTML, lo que permite aplicar los mismos estilos a diferentes partes de un sitio web.

Los selectores de clases son una forma eficiente y flexible de aplicar estilos a elementos HTML, lo que permite una separación clara entre la estructura y la presentación de un sitio web. Esto facilita la creación de estilos coherentes y la gestión de la apariencia de un sitio web a medida que cambian los requisitos de diseño.

**Selectores de etiquetas**

**Significado:** Los selectores de etiqueta apuntan a todos los elementos HTML que coinciden con el nombre de la etiqueta especificada y les aplican los estilos definidos.

**Definición:** Un selector de etiqueta se define utilizando simplemente el nombre de la etiqueta HTML. Por ejemplo, si deseas aplicar un estilo a todos los elementos de párrafo **<p>**, usarías **p { ... }**.

**Uso:** Los selectores de etiqueta son útiles cuando deseas aplicar un estilo a un tipo específico de elemento HTML en toda tu página o en una sección específica de tu documento.

**Selección:** Cuando defines un estilo con un selector de etiqueta, como **p { color: blue; }**, todos los elementos **<p>** en tu página tendrán el color de texto azul.

**Aplicación:** Los estilos definidos con selectores de etiqueta se aplican a todos los elementos correspondientes en tu documento HTML. Esto significa que si tienes múltiples elementos **<p>**, todos ellos heredarán los mismos estilos.

Los selectores de etiqueta son una herramienta poderosa en CSS para aplicar estilos a elementos HTML específicos en un documento, lo que te permite controlar la apariencia y el diseño de tu página web de manera efectiva y eficiente.

**Selectores ID**

**Definición:** Un selector de ID se define utilizando el símbolo de numeral (**#**) seguido del nombre del ID del elemento HTML. Por ejemplo, si deseas aplicar un estilo a un elemento con el ID "encabezado", usarías **#encabezado { ... }**.

**Uso:** Los selectores de ID son útiles cuando deseas aplicar un estilo a un elemento HTML específico en tu página, como un encabezado, un pie de página, un menú, etc.

**Selección:** Cuando defines un estilo con un selector de ID, como **#encabezado { color: red; }**, solo el elemento con el ID "encabezado" en tu página tendrá el color de texto rojo.

**Aplicación:** Los estilos definidos con selectores de ID se aplican solo al elemento específico que tiene ese ID en tu documento HTML. No se pueden reutilizar en otros elementos.

Es importante tener en cuenta que los selectores de ID deben ser únicos en un documento HTML. Aunque es posible aplicar estilos a múltiples elementos con la misma clase utilizando selectores de clase, los selectores de ID están diseñados para ser utilizados en elementos únicos. Esto los hace especialmente útiles cuando necesitas aplicar estilos a elementos específicos y distintivos en tu página web.

**Términos y sus funciones**

1. Selector de elemento: Selecciona elementos HTML específicos para aplicar estilos.
2. Selector de clase: Selecciona elementos con una clase específica para aplicar estilos.
3. Selector de ID: Selecciona un elemento con un ID específico para aplicar estilos.
4. Selector universal (\*): Selecciona todos los elementos en un documento para aplicar estilos.
5. Pseudo-clases: Selecciona elementos en ciertos estados, como **:hover**, **:active**, **:focus**, etc.
6. Pseudo-elementos: Selecciona partes específicas de un elemento, como **::before** y **::after**.
7. **color**: Establece el color del texto.
8. **background-color**: Establece el color de fondo de un elemento.
9. **font-size**: Establece el tamaño de la fuente.
10. **font-family**: Establece la familia de fuentes para el texto.
11. **font-weight**: Establece el grosor de la fuente, como normal o negrita.
12. **text-align**: Alinea el texto en el elemento, como izquierda, derecha, centrado o justificado.
13. **text-decoration**: Establece la decoración del texto, como subrayado, tachado, etc.
14. **line-height**: Establece la altura de línea del texto.
15. **letter-spacing**: Establece el espaciado entre letras.
16. **word-spacing**: Establece el espaciado entre palabras.
17. **margin**: Establece el margen exterior de un elemento.
18. **padding**: Establece el relleno interior de un elemento.
19. **border**: Establece el borde de un elemento.
20. **width**: Establece el ancho de un elemento.
21. **height**: Establece la altura de un elemento.
22. **display**: Controla cómo se muestra un elemento, como bloque, en línea o en línea-bloque.
23. **position**: Controla la posición de un elemento en la página.
24. **top**, **right**, **bottom**, **left**: Establece la posición de un elemento en relación con su contenedor.
25. **float**: Controla la ubicación de un elemento en relación con otros elementos.
26. **clear**: Controla cómo se deben manejar los elementos después de un elemento flotante.
27. **overflow**: Controla cómo se muestra el contenido que desborda un elemento.
28. **visibility**: Controla la visibilidad de un elemento.
29. **z-index**: Controla el orden de apilamiento de los elementos.
30. **opacity**: Controla la opacidad de un elemento.
31. **transition**: Agrega transiciones suaves entre los estados de un elemento.
32. **transform**: Aplica transformaciones a un elemento, como rotación, escala, etc.
33. **box-shadow**: Agrega una sombra alrededor de un elemento.
34. **text-shadow**: Agrega una sombra al texto de un elemento.
35. **background-image**: Establece una imagen de fondo para un elemento.
36. **background-position**: Controla la posición de una imagen de fondo.
37. **background-size**: Controla el tamaño de una imagen de fondo.
38. **background-repeat**: Controla la repetición de una imagen de fondo.
39. **background-attachment**: Controla si la imagen de fondo se desplaza con el contenido.
40. **cursor**: Controla el aspecto del cursor cuando se sitúa sobre un elemento.
41. **list-style**: Establece el estilo de viñetas o numeración para listas.
42. **outline**: Establece un contorno alrededor de un elemento sin afectar al diseño de la caja.
43. **box-sizing**: Controla cómo se calcula el tamaño total de un elemento.
44. **border-radius**: Establece el radio de las esquinas de un elemento.
45. **text-transform**: Controla la transformación del texto, como mayúsculas o minúsculas.
46. **white-space**: Controla cómo se maneja el espacio en blanco dentro de un elemento.
47. **cursor**: Controla el aspecto del cursor cuando se coloca sobre un elemento.
48. **pointer-events**: Controla la respuesta de eventos del puntero para un elemento.
49. **user-select**: Controla si el texto se puede seleccionar en un elemento.

**overflow-scrolling**: Controla el desplazamiento suave en dispositivos táctiles.

Principio del formulario

**Pseudoclases**

Las pseudoclases en CSS son selectores que permiten aplicar estilos a un elemento HTML en función de su estado o posición dentro del documento, sin necesidad de modificar el HTML. Estas pseudoclases permiten seleccionar elementos basados en interacciones del usuario o en información estructural que no está presente en el código HTML original.

Algunos ejemplos comunes de pseudoclases en CSS incluyen **:hover**, que se aplica cuando el cursor del mouse se sitúa sobre un elemento, **:active**, que se aplica cuando un elemento está siendo activado por el usuario, y **:focus**, que se aplica cuando un elemento está siendo seleccionado por el usuario, como en el caso de un campo de entrada de formulario.

Las pseudoclases son una herramienta poderosa para crear estilos interactivos y mejorar la experiencia del usuario en una página web, permitiendo cambios dinámicos en la apariencia de los elementos en respuesta a acciones específicas del usuario.

Se escriben luego de un selector y antecedidos por dobles puntos.

**Flex Box**

Flexbox, también conocido como Flexible Box Layout, es un modelo de diseño en CSS que permite crear diseños más flexibles y eficientes, especialmente útiles para organizar elementos en una sola dimensión (ya sea horizontal o verticalmente) y para crear diseños responsivos.

1. **display**: Define el elemento como un contenedor flex. Se aplica al contenedor padre y su valor debe ser **flex** para activar el modelo de diseño flex.
2. **flex-direction**: Establece la dirección en la que se colocan los elementos flexibles dentro del contenedor. Los valores pueden ser **row** (misma dirección del texto), **row-reverse**(dirección opuesta al texto), **column** (de arriba hacia abajo) o **column-reverse** (de abajo hacia arriba).
3. **flex-wrap**: Determina si los elementos flexibles deben envolverse o no cuando no hay suficiente espacio en el contenedor. Los valores pueden ser **nowrap** ( se ajusta a una sola línea. No envolver), **wrap** (Se envuelve alrededor de las líneas adicionales) o **wrap-reverse** (Se envuelven alrededor de las líneas adiciones en reversa).
4. **justify-content**: Alinea los elementos flexibles a lo largo del eje principal del contenedor (horizontal). Los valores pueden ser **flex-start** (Alinea lado izquierdo), **flex-end** (Alinea lado derecho), **center**, **space-between** (Distribuye el mismo espacio entre ellas), **space-around** (distribuye el espacio alrededor de los elementos, hay mucho espacio alrededor).
5. **align-items**: Alinea los elementos flexibles a lo largo del eje transversal del contenedor (Vertical). Determina los elementos en su conjunto, están alineados dentro de su contenedor. Cuando hay una sola línea align-content NO tiene efecto. Los valores pueden ser **flex-start** (Alinea a la parte superior del contenedor), **flex-end** (Alinea a la parte inferior del contenedor), **center**, **baseline** (alinea los elementos en la línea base) o **stretch** (estira los elementos para que coincidan con la altura del contenedor).
6. **align-content**: Alinea las líneas de elementos flexibles dentro del contenedor cuando hay espacio adicional en el eje transversal (Vertical). Determina el espacio entre líneas. Los valores pueden ser **flex-start** (Parte superior del contenedor), **flex-end** (Parte inferior del contenedor), **center** (Centro vertical), **space-between** (misma distancia), **space-around** (Las líneas se muestran con la misma separación alrededor de ellas) o **stretch** (Las líneas se estiran para ajustarse al contenedor).
7. **Flex-flow:** propiedad abreviada que combina las propiedades **flex-direction** y **flex-wrap**. Permite definir tanto la dirección de los elementos flexibles dentro del contenedor (fila o columna) como si los elementos deben envolverse o no cuando no hay suficiente espacio en el contenedor.
8. **flex-grow**: Determina cuánto espacio adicional puede ocupar un elemento flexible en relación con los demás elementos flexibles dentro del mismo contenedor.
9. **flex-shrink**: Determina cuánto puede encogerse un elemento flexible en relación con los demás elementos flexibles dentro del mismo contenedor si no hay suficiente espacio.
10. **flex-basis**: Establece el tamaño base de un elemento flexible antes de que se apliquen **flex-grow** y **flex-shrink**.
11. **order**: Establece el orden de visualización de un elemento flexible en relación con los demás elementos flexibles dentro del mismo contenedor. Elementos individuales.
12. **Align-self:** se utiliza para anular la alineación predeterminada de un elemento flexible dentro del contenedor flex en el eje transversal. Mientras que la propiedad **align-items** se aplica al contenedor flex y afecta a todos los elementos hijos, **align-self** se aplica a un elemento flexible específico y le permite controlar su propia alineación vertical dentro del contenedor.

**JAVASCRIPT**

Es un lenguaje de programación interpretado, de tipado dinámico y débilmente tipado. Esto significa que no necesitas declarar el tipo de una variable al definirla y que el tipo de una variable puede cambiar durante la ejecución del programa.

**Caracteristicas**

1. Velocidad 2. Popularidad 3. Server load 4. Compatibilidad 5. Simplicidad 6. Versatilidad

**Variables y datos**

* Variables: Pueden ser declaradas con las palabras clave **var**, **let** o **const**.
* Tipos de datos: **number** (representar y manipular valores númericos), **string** (datos textuales-palabras, van entre comillas). **boolean** (valor lógico. True o False. Palabras reservadas), **null** (ausencia intencional de cualquier valor. Valor nulo o vacío), **undefined** (no se le ha asignado ninguna variable/valor**)**, **object (**permite almacenar una colección de pares clave-valor. Las claves (también llamadas propiedades) son cadenas o símbolos, y los valores pueden ser de cualquier tipo de datos, incluidos otros objetos. Los objetos pueden tener métodos, que son funciones asociadas al objeto) y **symbol** (dato primitivo,cada valor es único y no se puede cambiar. Identificadores únicos para propiedades de objetos).

**Orden de Precedencia**

Determina el orden en el que se evalúan las expresiones en JavaScript. Se puede modificar utilizando paréntesis.

**Strings y Adición**

Las cadenas de texto se definen con comillas simples o dobles. La concatenación se realiza con el operador + y no solo se aplican a números.

**Operadores de comparación**

1. **Operadores de Comparación**:

**Igualdad (== y ===)**:

* + - **==**: Comprueba la igualdad de valor después de realizar la conversión de tipos, lo que significa que puede hacer conversiones automáticas entre tipos de datos.
    - **===**: Comprueba la igualdad de valor y tipo sin realizar conversiones de tipo. Es más estricto que **==**.

**Desigualdad (!= y !==)**:

* + - **!=**: Comprueba la desigualdad de valor después de realizar la conversión de tipos.
    - **!==**: Comprueba la desigualdad de valor y tipo sin realizar conversiones de tipo.

**Mayor y Menor que (> y <)**:

* + - **>**: Comprueba si el operando izquierdo es mayor que el operando derecho.
    - **<**: Comprueba si el operando izquierdo es menor que el operando derecho.

**Mayor o Igual y Menor o Igual (>= y <=)**:

* + - **>=**: Comprueba si el operando izquierdo es mayor o igual que el operando derecho.
    - **<=**: Comprueba si el operando izquierdo es menor o igual que el operando derecho.

1. **Operadores de Igualdad y Asignación**:
   * **Asignación (=)**: Asigna un valor a una variable.
   * **Igualdad (== y ===)**: Compara dos valores para determinar si son iguales.
     + **==**: Igualdad de valor (conversión de tipo).
     + **===**: Igualdad de valor y tipo (sin conversión de tipo). Igualdad **estricta**.
2. **Asociatividad**:
   * La asociatividad de un operador determina el orden en que se evalúan las operaciones que tienen la misma precedencia.
   * Los operadores de asignación (**=**) y los operadores de igualdad (**==** y **===**) son operadores no asociativos, lo que significa que no se pueden encadenar.
   * Los operadores de comparación (**>**, **<**, **>=**, **<=**, **!=**, **!==**) son asociativos de izquierda a derecha, lo que significa que se evalúan de izquierda a derecha. Por ejemplo, **a > b > c** se evaluará como **(a > b) > c**.

**Funciones**

En JavaScript, las funciones son bloques de código reutilizables que pueden ser llamados y ejecutados en cualquier momento durante la ejecución del programa. Las funciones permiten encapsular un conjunto de instrucciones y ejecutarlas varias veces sin necesidad de repetir el código. Puedes declarar una función utilizando la palabra clave **function**, seguida del nombre de la función y sus parámetros (si los tiene), y luego el bloque de código que define lo que hace la función. También puedes definir una función como una expresión, asignándola a una variable. En este caso, la función se define sin un nombre (función anónima) o se asigna a una variable con un nombre específico. A partir de ECMAScript 6 (ES6), se introdujeron las arrow functions, que ofrecen una sintaxis más concisa y clara para definir funciones. Estas funciones son ideales para funciones simples y anónimas. Una vez que has definido una función, puedes llamarla en cualquier parte de tu código proporcionando los valores necesarios para los parámetros, si los tiene. Las funciones son una parte fundamental del lenguaje y se utilizan ampliamente en el desarrollo de aplicaciones web y otros tipos de programas.

**Ejemplo:** function saludar(nombre) {

console.log('¡Hola, ' + nombre + '!');

}

var saludar = function(nombre) {

console.log('¡Hola, ' + nombre + '!');

};

var saludar = (nombre) => {

console.log('¡Hola, ' + nombre + '!');

};

**Diferencias entre return, console log y break**

La diferencia entre **return** , **console.log(), break** radica en su propósito y cómo afectan el flujo de ejecución de una función en JavaScript:

**1.console.log()**

Propósito: **console.log()** se utiliza para imprimir mensajes de depuración en la consola del navegador o en la consola de desarrollo del entorno de ejecución (como Node.js). Se usa principalmente para mostrar información durante el desarrollo y la depuración de un programa.

Efecto en el flujo de ejecución: **console.log()** no afecta el flujo de ejecución de la función en la que se encuentra. Simplemente imprime el valor especificado en la consola y la ejecución continúa con la siguiente instrucción.

En resumen imprime en la consola una variable. Sirve solo como ayuda para programar, luego se debe eliminar del código.

**2.return**

Propósito: **return** se utiliza para devolver un valor de una función al lugar donde se llamó. Permite que una función calcule un valor y luego lo pase de vuelta al código que la invocó.

Efecto en el flujo de ejecución: Cuando se ejecuta **return**, la función se detiene y devuelve el valor especificado. Después de que se alcanza una declaración **return**, la ejecución de la función se detiene y el control vuelve al punto de llamada.

En resumen es fundamental para toda función. Es la instrucción que indica que valor nos debe devolver, el return devuelve ese valor y corta, no sigue ejecutando el código que hay debajo.

**3.Break**

Se utiliza para alterar el flujo de control del programa, saliendo de la estructura de control en la que se encuentra. No devuelve ningún valor, corta los bucles y luego sigue evaluando. Solo se utilizan en bucles.

En resumen, **console.log()** se utiliza para imprimir valores en la consola, mientras que **return** se utiliza para devolver valores de una función al código que la llamó. **console.log()** es útil para propósitos de depuración, mientras que **return** es fundamental para el flujo de datos y el control de una aplicación JavaScript. **Break** Se utiliza para salir de una estructura de control como un bucle o un switch. Cada uno tiene un propósito diferente y afecta al flujo de ejecución del programa de manera diferente.

**Nomenclatura**

1. **Camel Case**:
   * Escribir palabras compuestas o frases sin espacios ni guiones.
   * La primera palabra comienza con minúscula, mientras que las palabras subsiguientes comienzan con mayúscula.
   * Ejemplo: **miVariable**, **miFuncion**, **miClase**.
2. **Pascal Case**:
   * Similar al Camel Case, pero la primera letra de cada palabra se escribe en mayúscula, incluso la primera palabra.
   * Se utiliza comúnmente para nombrar clases o constructores en JavaScript.
   * Ejemplo: **MiVariable**, **MiFuncion**, **MiClase**.
3. **Snake Case**:
   * Las palabras están separadas por guiones bajos (**\_**).
   * Puede ser útil cuando se trabaja con lenguajes que no permiten espacios en los nombres de variables o funciones.
   * Ejemplo: **mi\_variable**, **mi\_funcion**, **mi\_clase**.
4. **Kebab Case**:
   * Similar al Snake Case, pero las palabras están separadas por guiones (**-**) en lugar de guiones bajos.
   * Se utiliza comúnmente en CSS y en la definición de atributos en HTML.
   * Ejemplo: **mi-variable**, **mi-funcion**, **mi-clase**.
5. **Upper Case**:
   * Todas las letras se escriben en mayúsculas.
   * Puede ser utilizado para nombrar constantes o valores inmutables.
   * Ejemplo: **MI\_CONSTANTE**, **VALOR\_INMUTABLE**.

Estos son algunos de los tipos de nomenclatura más utilizados en JavaScript. La elección de uno u otro tipo de nomenclatura puede depender de las convenciones del proyecto, del estilo de codificación personal o de las normas establecidas por el equipo de desarrollo. Es importante mantener la consistencia en el estilo de nomenclatura dentro de un proyecto para mejorar la legibilidad y mantenibilidad del código

**Operadores condicionales**

Permite que se cumplan unas u otras instrucciones dependiendo de alguna condición.

**IF:**

* La declaración **if** se utiliza para ejecutar un bloque de código si una condición especificada es verdadera.
* Si la condición dentro del **if** se evalúa como verdadera (es decir, el resultado de la expresión es true), el bloque de código dentro del **if** se ejecutará.
* Si la condición es falsa, el bloque de código dentro del **if** se omitirá y la ejecución continuará con la siguiente línea de código después del bloque **if.**

**ELSE**

* La declaración **else** se utiliza junto con **if** para ejecutar un bloque de código diferente si la condición especificada en el **if** es falsa.
* El bloque de código dentro de **else** se ejecutará solo si la condición del **if** es falsa.
* **else** no tiene una condición propia; simplemente captura cualquier cosa que no se haya capturado por el **if** previo.

**ELSE IF**

* Puedes anidar múltiples declaraciones **if-else** dentro de otras para realizar pruebas adicionales.
* Esto permite controlar diferentes escenarios con múltiples condiciones.

**Objetos globales**

En JavaScript, los objetos globales son aquellos que están disponibles en todo el entorno de ejecución del código, ya sea en el navegador o en Node.js. Estos objetos proporcionan funcionalidades y características esenciales que están disponibles en todo momento durante la ejecución del programa. Algunos de los objetos globales más comunes en JavaScript incluyen:

1. **Object**: Es el objeto raíz de JavaScript, y todos los demás objetos heredan de él. Proporciona métodos y propiedades para trabajar con objetos.
2. **Array**: Proporciona métodos para crear y manipular matrices (listas de elementos).
3. **Function**: Es el constructor para crear funciones en JavaScript.
4. **String**: Proporciona métodos para trabajar con cadenas de texto.
5. **Number**: Proporciona métodos y propiedades para trabajar con números.
6. **Boolean**: Proporciona métodos y propiedades para trabajar con valores booleanos (verdadero o falso).
7. **Date**: Proporciona métodos para trabajar con fechas y horas.
8. **Math**: Proporciona métodos y propiedades para realizar operaciones matemáticas.

**Math pow**: potencia un número

**Math 1. round** (n° decimal al más entero próximo) **2.** Floor (n° decimal al entero de menor valor) **3.** Ceil ( n° al siguiente entero).

**Math max o min:** permite conocer el valor máximo o mínimo.

**Math random:** genera n° aleatorios entre 1 y 0.

1. **JSON**: Proporciona métodos para trabajar con el formato JSON (JavaScript Object Notation), que es comúnmente utilizado para el intercambio de datos.
2. **Console**: Proporciona métodos para enviar mensajes de depuración y registro a la consola del navegador o del entorno de ejecución.
3. **Window (en el navegador)**: Es el objeto global que representa la ventana del navegador. Proporciona acceso a métodos y propiedades relacionados con la ventana del navegador, como la ubicación, el historial y la manipulación del DOM.
4. **Global (en Node.js)**: Es el objeto global en el entorno de ejecución de Node.js. Proporciona acceso a métodos y propiedades específicos de Node.js, como el sistema de archivos, el manejo de eventos y la gestión de módulos.

**OPERADORES LÓGICOS**

1. **AND (&&)**: Conjunción. Se representa con el símbolo **&&** en la mayoría. Devuelve verdadero si ambas condiciones son verdaderas, de lo contrario, devuelve falso.

**Ejemplo**: (5<2) && (5>3). Esto devuelve **FALSE**.

1. **OR (||)**: Disyunción. Se representa con el símbolo **||**. Devuelve verdadero si al menos una de las condiciones es verdadera, de lo contrario, devuelve falso.

**Ejemplo**: (5<2) || (5>3). Esto devuelve **TRUE**.

1. **NOT (!)**: Negación. Se representa con el símbolo **!.** Se utiliza para negar una condición, es decir, si una condición es verdadera, NOT la convierte en falsa, y viceversa.

**Ejemplo**: ! (5<2). Esto devuelve **TRUE**.

El orden de precedencia de los operadores lógicos sigue una convención similar al álgebra booleana. Por lo general, el orden de precedencia de los operadores lógicos es el siguiente, de mayor a menor precedencia: **NOT, AND Y OR.**

Este orden de precedencia garantiza que las expresiones lógicas se evalúen de manera consistente y lógica. Sin embargo, es importante recordar que el uso de paréntesis ( ) para agrupar operaciones puede anular el orden de precedencia predeterminado y especificar el orden de evaluación deseado.

**Bucles o ciclos de interacción**

Son estructuras de control en la programación que permiten ejecutar repetidamente un bloque de código mientras se cumpla una condición específica. Los bucles son fundamentales para la automatización de tareas repetitivas y para procesar conjuntos de datos de manera eficiente.

Hay varios tipos de bucles, pero los más comunes son:

**1**.**While** (Mientras): Este bucle ejecuta un bloque de código mientras una condición especificada sea verdadera. La condición se evalúa antes de cada iteración. Si la condición es falsa al inicio, el bloque de código no se ejecuta en absoluto. No se saben la cantidad de pasos hasta finalizar la ejecución. La variable integradora se crea por fuera y la condición de parada va dentro del paréntesis.

¿Cuándo se utiliza este bucle?

Se utiliza cuando no estás seguro de cuántas veces se ejecutará el bloque de código.

La condición se evalúa antes de ejecutar el bloque de código. Si la condición es falsa desde el principio, el bloque de código no se ejecuta en absoluto.

Si la condición es falsa después de una iteración, el bucle se detiene inmediatamente.

**Ejemplo:** let i = 0;

while (i < 5) {

console.log(i);

i++;

}

**2.For** (Para): Este bucle ejecuta un bloque de código un número específico de veces. Por lo general, se utiliza cuando se sabe de antemano cuántas veces se debe ejecutar el bloque de código. Se sitúa como segundo parámetro entre ( ). Va todo dentro del paréntesis, variable integradora y condición de parada.

**Ejemplo:** (let i= 0 ; I <5; i++) {

Console.log (i);

}

**3.Do-While** (Hacer Mientras): Similar al bucle while, pero la condición se evalúa después de cada iteración. Esto garantiza que el bloque de código se ejecute al menos una vez, incluso si la condición es falsa desde el principio. Sentencia especificada, hasta que la condición de comprobación evalúe falsa.

¿Cuándo se utiliza este bucle?

Se utiliza cuando deseas que el bloque de código se ejecute al menos una vez, incluso si la condición es falsa desde el principio.

La condición se evalúa después de ejecutar el bloque de código. Esto garantiza que el bloque de código se ejecute al menos una vez, independientemente de la condición.

Si la condición es falsa después de una iteración, el bucle se detiene.

**Ejemplo:** let i = 0;

do {

console.log(i);

i++;

} while (i < 5);

Los bucles permiten que el flujo de ejecución de un programa se repita hasta que se cumplan ciertas condiciones, lo que los hace esenciales para la implementación de algoritmos y la manipulación de datos en la programación. Sin embargo, es importante tener cuidado para evitar bucles infinitos, donde la condición nunca se vuelve falsa, lo que podría causar que el programa se bloquee o se vuelva no funcional.

**Recursos adicionales**

**1.Switch:** es una estructura de control que se utiliza para seleccionar una de varias opciones basadas en el valor de una expresión dada. Se utiliza cuando se tienen múltiples case y se desea ejecutar diferentes bloques de código según el valor de una variable.

**Ejemplo:** switch (expresion) {

case valor1:

break;

case valor2:

break;

default:

// código a ejecutar si ninguno de los casos coincide

}

Después de evaluar cada caso, se utiliza la palabra clave **break** para salir del **switch** y evitar que se ejecuten los casos siguientes.

**2.Break:** la palabra clave break se utiliza para salir de una estructura de control, como switch, for, while, o do-while.

En el contexto de switch, se utiliza para salir del switch después de que se ha ejecutado el bloque de código correspondiente a un caso.

Si break no se usa, el control fluirá hacia el siguiente caso, lo que puede no ser lo deseado.

No devuelve ningún valor, solo corta el bucle.

**3.Continue:** La palabra clave **continue** se utiliza en bucles (**for**, **while**, **do-while**) para saltar una iteración y continuar con la siguiente.

En el contexto de un bucle, **continue** hace que la ejecución salte al final del bucle actual y continúe con la siguiente iteración.

En un **switch**, **continue** no se utiliza porque no hay iteraciones que saltar, se emplea **break** para salir del **switch** si es necesario.

En resumen, **switch** se usa para seleccionar entre múltiples opciones basadas en el valor de una expresión, **break** se usa para salir de una estructura de control, y **continue** se usa para saltar una iteración en un bucle y continuar con la siguiente.

**ARRAYS**

También conocidos como arreglos, son estructuras de datos que permiten almacenar múltiples elementos del mismo tipo en una sola variable bajo un mismo nombre. Estos elementos están dispuestos de manera contigua en la memoria, lo que facilita el acceso y la manipulación de los datos. Los arrays pueden ser unidimensionales (vectores), bidimensionales (matrices) o tener incluso más dimensiones. Son una herramienta fundamental en la programación para almacenar y manipular conjuntos de datos de manera eficiente. Van entre [ ].

**Elementos:** se refiere a los valores individuales almacenados en el array. Por ejemplo, si tienes un array de números como **[2, 4, 6, 8, 10]**, cada uno de esos números (2, 4, 6, 8, 10) es un elemento del array.

**Índice:** es la posición de un elemento dentro del array. Los índices suelen comenzar desde 0 en muchos lenguajes de programación. En el ejemplo anterior, el índice del primer elemento (2) sería 0, el índice del segundo elemento (4) sería 1, y así sucesivamente. En general, el índice **i** indica la posición del **i**-ésimo elemento en el array.

**Elemento e índice NO SON LO MISMO!**

En resumen, los elementos son los valores almacenados en el array, mientras que los índices son las posiciones que ocupan esos valores dentro del array.

**Métodos de los arrays**

Son funciones nativas/preestablecidas en los lenguajes de programación.

1. **Acceso a elementos**
   * Código: **array[i]**
   * Descripción: Permite acceder a un elemento en particular del array utilizando su posición.
2. **Agregar elementos al final**
   * Código: **push()**
   * Descripción: Agrega un elemento al final del array.
3. **Eliminar elementos del final**
   * Código: **pop()**
   * Descripción: Elimina el último elemento del array.
4. **Agregar elementos al principio**
   * Código: **unshift()**
   * Descripción: Agrega un elemento al principio del array.
5. **Eliminar elementos del principio**
   * Código: **shift()**
   * Descripción: Elimina el primer elemento del array.
6. **Buscar un elemento**
   * Código: **indexOf()**
   * Descripción: Busca la posición de un elemento en el array.
7. **Eliminar elementos por valor**
   * Código: **splice()**
   * Descripción: Elimina un elemento específico del array basado en su valor.
8. **Ordenar elementos**
   * Código: **sort()**
   * Descripción: Ordena los elementos del array.
9. **Iterar sobre los elementos**
   * Código: **forEach()**
   * Descripción: Permite recorrer todos los elementos del array.
10. **Obtener la longitud del array**
    * Código: **length**
    * Descripción: Retorna el número total de elementos en el array.
11. **Incluye un elemento**
    * Código: **includes()**
    * Descripción: Determina si un array incluye un determinado elemento y retorna **true** o **false** en consecuencia.
12. **Verificar si todos los elementos cumplen una condición**
    * Código: **every()**
    * Descripción: Verifica si todos los elementos del array cumplen una condición especificada y retorna **true** o **false** en consecuencia.
13. **Dividir un string en un array de substrings**
    * Código: **split()**
    * Descripción: Divide un string en un array de substrings basado en un separador especificado.
14. **Unir todos los elementos de un array en un string**
    * Código: **join()**
    * Descripción: Une todos los elementos de un array en un único string, opcionalmente separados por un separador especificado.
15. **Crear un nuevo array con los resultados de llamar a una función para cada elemento**
    * Código: **map()**
    * Descripción: Crea un nuevo array con los resultados de llamar a una función proporcionada para cada elemento del array.
16. **Crear un nuevo array con los elementos que cumplan una condición**
    * Código: **filter()**
    * Descripción: Crea un nuevo array con todos los elementos que cumplan una condición especificada por una función proporcionada.
17. **Encontrar el primer elemento que cumple una condición**
    * Código: **find()**
    * Descripción: Retorna el primer elemento en el array que cumple con la condición especificada por una función proporcionada.
18. **Encontrar el índice del primer elemento que cumple una condición**
    * Código: **findIndex()**
    * Descripción: Retorna el índice del primer elemento en el array que cumple con la condición especificada por una función proporcionada.
19. **Reducir el array a un solo valor**
    * Código: **reduce()**
    * Descripción: Aplica una función a un acumulador y a cada valor del array (de izquierda a derecha) para reducirlo a un único valor.
20. **Reducir el array a un nuevo array**
    * Código: **reduceRight()**
    * Descripción: Aplica una función a un acumulador y a cada valor del array (de derecha a izquierda) para reducirlo a un nuevo array.

**OBJETOS**

Los objetos son una de las estructuras de datos fundamentales en la programación, utilizados para almacenar colecciones de datos de manera estructurada. En JavaScript, los objetos son una colección de pares clave-valor, donde cada clave es un identificador único que se utiliza para acceder a su correspondiente valor.

1. **Concepto de Objeto**:
   * Un objeto es una entidad independiente que agrupa datos (llamados propiedades) y funcionalidades (llamadas métodos). Se puede pensar en un objeto como una caja que contiene tanto datos como las acciones que se pueden realizar con esos datos.
2. **Clave (key)**:
   * Una clave, también conocida como nombre de propiedad, es una cadena única que actúa como identificador para acceder a un valor específico dentro de un objeto. Cada clave dentro de un objeto debe ser única.
3. **Valor (value)**:
   * El valor es la información asociada a una clave en un objeto. Puede ser cualquier tipo de dato válido en JavaScript, como un número, una cadena, un booleano, un array, otro objeto, o incluso una función (método).
4. **Sintaxis de Objeto en JavaScript**:
   * En JavaScript, los objetos se definen utilizando llaves **{}**. La sintaxis básica es **{ clave: valor }**, donde **clave** es el nombre de la propiedad y **valor** es el valor asociado a esa propiedad.

**Ejemplo:** var persona = {

nombre: "Juan",

edad: 30,

casado: false

};

1. **Acceso a las propiedas**:

Las propiedades de un objeto se pueden acceder utilizando la notación de corchetes que se llama **bracket-notation** o notación de punto que se llama **dot-notation.**

**Bracket-notation**

En la notación de corchetes, utilizas corchetes ([ ]) y dentro de ellos colocas el nombre de la propiedad como una cadena.

Esto es útil cuando el nombre de la propiedad es dinámico o contiene caracteres especiales que no son válidos en la notación de punto.

Se utiliza cuando no conozco la propiedad del objeto.

**Ejemplo:** var persona = {

nombre: "Juan",

edad: 30

};

console.log(persona["nombre"]); // Salida: Juan

console.log(persona["edad"]); // Salida: 30

**Dot-notation**

En la notación de punto, utilizas un punto (.) seguido del nombre de la propiedad para acceder al valor de esa propiedad. Lo utilizo cuando conozco la propiedad del objeto.

**Ejemplo:** var persona = {

nombre: "Juan",

edad: 30

};

console.log(persona.nombre); // Salida: Juan

console.log(persona.edad); // Salida: 30

Ambas notaciones proporcionan una forma de acceder a las propiedades de un objeto, pero la notación de corchetes ofrece más flexibilidad al permitir nombres de propiedades dinámicos o con caracteres especiales. La notación de punto, por otro lado, es más concisa y se usa comúnmente cuando el nombre de la propiedad es estático.

**Breve resumen**

**1**.Para acceder a la propiedad de un objeto, tenemos que escribir el nombre del objeto y el nombre de la propiedad. Lo podemos hacer con **dot-notation** o **braket-notation.** **Ejemplo:** (persona.edad)

**2**.Para cambiar o agregar (si no existe) el valor de una propiedad tenemos que acceder a ella e igualar al nuevo valor. **Ejemplo:** persona.edad = 32

**3**.Para eliminar propiedades se utiliza la palabra reservada delete**. Ejemplo:** delete.persona.edad

**Métodos de los objetos**

Los objetos en JavaScript pueden contener propiedades que son pares de clave-valor, y también pueden contener métodos, que son funciones asociadas a esos objetos.

Son funciones asociadas a un objeto. Estos métodos permiten que los objetos realicen acciones específicas.

La estructura básica de un método implica definir una función dentro de un objeto. Esto permite al objeto realizar tareas específicas cuando se invoca ese método.

**1.Object.keys():**

Este método devuelve un array de las claves (propiedades) de un objeto.

Sintaxis: Object.keys(objeto)

Ejemplo: var persona = {

nombre: "Juan",

edad: 30

};

console.log(Object.keys(persona)); // Salida: ["nombre", "edad"]

**2.Object.values()**:

Este método devuelve un array de los valores de las propiedades de un objeto

Sintaxis: Object.values(objeto)

Ejemplo: var persona = {

nombre: "Juan",

edad: 30

}

console.log(Object.values(persona)); // Salida: ["Juan", 30]

**3.Object.entries()**:

Este método devuelve una matriz de pares clave-valor de un objeto como arrays [clave, valor].

Sintaxis: Object.entries(objeto)

Ejemplo: var persona = {

nombre: "Juan",

edad: 30

};

console.log(Object.entries(persona)); // Salida: [["nombre", "Juan"], ["edad", 30]]

**4.Object.assign()**: Este método se utiliza para copiar los valores de todas las propiedades propias enumerables de uno o más objetos fuente a un objeto destino.

Sintaxis: Object.assign(objetoDestino, objetoFuente1, objetoFuente2, ...)

Ejemplo: var destino = {};

var fuente = { nombre: "Juan", edad: 30 };

Object.assign(destino, fuente);

console.log(destino); // Salida: { nombre: "Juan", edad: 30 }

**5.hasOwnProperty():** método del objeto **Object** en JavaScript que se utiliza para determinar si un objeto tiene una propiedad con un nombre específico como propiedad propia del objeto. Esto significa que el método **hasOwnProperty()** solo devuelve **true** si la propiedad especificada es una propiedad directa del objeto y no ha sido heredada a través de la cadena de prototipos del objeto.

Sintaxis: objeto.hasOwnProperty(nombrePropiedad)

Ejemplo: var persona = {

nombre: "Juan",

edad: 30

};

console.log(persona.hasOwnProperty("nombre")); // Salida: true

console.log(persona.hasOwnProperty("apellido")); // Salida: false

En este ejemplo, **persona** tiene la propiedad **"nombre"** como una propiedad propia, por lo que **persona.hasOwnProperty("nombre")** devuelve **true**. Sin embargo, **persona** no tiene una propiedad **"apellido"** como una propiedad propia, por lo que **persona.hasOwnProperty("apellido")** devuelve **false**.

**Bucles**

**FOR:** Se utiliza principalmente para iterar sobre **arrays** u otras secuencias con un índice numérico. Recorre elemento por elemento.

Sintaxis: for (inicialización; condición; actualización) {

// bloque de código a ejecutar

}

Ejemplo: var array = [1, 2, 3, 4, 5];

for (var i = 0; i < array.length; i++) {

console.log(array[i]);

}

**FOR IN:** permite iterar sobre un objeto utilizando 2 variables pivot. Una representa el **objeto** que recorremos y la otra propiedad en la que se está realizando la iteración utilizando ambos valores podremos acceder al valor de cada propiedad del objeto. Recorre propiedad por propiedad.

Sintaxis: for (variable in objeto) {

// bloque de código a ejecutar

}

Ejemplo: var persona = { nombre: "Juan", edad: 30 };

for (var propiedad in persona) {

console.log(propiedad + ": " + persona[propiedad]);

}

**THIS**

El uso de **this** es importante en JavaScript para acceder a propiedades y métodos de un objeto dentro de una función, lo que permite la reutilización de funciones en diferentes contextos de objeto. Sin embargo, su comportamiento puede ser confuso en ciertos escenarios, por lo que es importante comprender cómo funciona en diferentes situaciones.

**Callback**

Es una función que se ejecuta cuando se produce un evento o se completa una tarea asincrónica, y se pasa como argumento a otra función para manejar esa acción.

los callbacks son comunes en situaciones como cargar archivos, hacer solicitudes a servidores, manejar eventos del usuario, etc. Son una forma de controlar el flujo del programa y ejecutar código en respuesta a eventos o acciones específicas.

Ejemplo: // Función que simula la acción de mezclar los ingredientes

function mezclarIngredientes(callback) {

console.log("Mezclando los ingredientes...");

setTimeout(function() {

console.log("Ingredientes mezclados.");

// Llamamos al callback una vez que los ingredientes están mezclados

callback();

}, 2000); // Simulamos 2 segundos para mezclar los ingredientes

}

// Función que simula la acción de hornear el pastel

function hornearPastel() {

console.log("Poniendo la mezcla en el horno...");

setTimeout(function() {

console.log("El pastel está listo. ¡Horneado con éxito!");

}, 3000); // Simulamos 3 segundos para hornear el pastel

}

// Llamamos a la función mezclarIngredientes y pasamos hornearPastel como callback mezclarIngredientes(hornearPastel);

Ejemplo 2: // Función que simula una operación asincrónica

function operacionAsincronica(callback) {

console.log("Iniciando operación asincrónica...");

// Simulamos una demora de 2 segundos

setTimeout(function() {

console.log("Operación asincrónica completada.");

// Llamamos al callback una vez que la operación se completa

callback();

}, 2000);

}

// Función que será nuestro callback

function miCallback() {

console.log("Callback ejecutado. ¡La operación ha finalizado!");

}

// Llamamos a la función operacionAsincronica y pasamos miCallback como argumento operacionAsincronica(miCallback)Principio del formulario