Programación Estructurada

Introducción a JavaScript

JavaScript es un lenguaje de programación ampliamente utilizado y versátil que se utiliza principalmente para agregar interactividad y funcionalidad a las páginas web. JavaScript es un lenguaje interpretado, lo que significa que no requiere un proceso de compilación previo y puede ser ejecutado directamente por el navegador.

Una de las características más destacadas de JavaScript es su capacidad para manipular el contenido HTML y CSS de una página web en tiempo real. Esto permite a los desarrolladores crear efectos visuales dinámicos, interactuar con elementos de la página, realizar validaciones de formularios y mucho más.

Además de su uso en el desarrollo web, JavaScript también se ha expandido a otros ámbitos, como el desarrollo de aplicaciones móviles y de escritorio, gracias a frameworks como React Native y Electron, respectivamente. También es utilizado en servidores a través de Node.js, permitiendo a los desarrolladores construir aplicaciones de lado del servidor utilizando JavaScript además JavaScript es un lenguaje orientado a objetos, lo que significa que los desarrolladores pueden crear objetos y definir propiedades y métodos para interactuar con ellos. También es un lenguaje de tipado dinámico, lo que significa que no es necesario declarar explícitamente el tipo de una variable antes de usarla.

Lenguaje de scripting del lado del cliente

JavaScript es un lenguaje de programación que se ejecuta en el navegador web del cliente (navegador del usuario). Esto significa que el código JavaScript se descarga y se ejecuta en el dispositivo del usuario, lo que permite crear interactividad y manipular el contenido de la página en tiempo real.

Tipado dinámico

JavaScript es un lenguaje de tipado dinámico, lo que significa que no se requiere especificar el tipo de datos de una variable al declararla. El tipo de datos de una variable puede cambiar dinámicamente durante la ejecución del programa.

En el siguiente ejemplo, declaramos una variable llamada miVariable sin especificar ningún tipo de dato. Luego, le asignamos diferentes valores, incluidos números, cadenas de texto, booleanos, objetos y funciones. En cada asignación, la variable cambia su tipo de dato sin necesidad de declarar un tipo específico.

Este comportamiento es una característica distintiva del tipado dinámico en JavaScript, lo que proporciona flexibilidad pero también requiere precaución para evitar errores al manipular variables que pueden cambiar de tipo en diferentes partes del código.

```
// Declaración de una variable sin tipo de dato específico
let miVariable;

// Asignación de un valor numérico a la variable
miVariable = 10;
console.log(miVariable); // Salida: 10

// Ahora, asignamos una cadena de texto a la misma variable
miVariable = "Hola, soy una cadena";
console.log(miVariable); // Salida: Hola, soy una cadena

// También podemos asignar un valor booleano a la variable
miVariable = true;
console.log(miVariable); // Salida: true

// Incluso podemos asignarle un objeto o una función a la variable
miVariable = { nombre: "Juan", edad: 30 };
console.log(miVariable); // Salida: { nombre: 'Juan', edad: 30 }

miVariable = function() {
  return "Soy una función";
};
console.log(miVariable()); // Salida: Soy una función
```

Figure 1 Ejemplo de tipado dinámico

Variables y constantes

Las variables en JavaScript se utilizan para almacenar valores y se declaran utilizando las palabras clave var, let o const. Las constantes, declaradas con const, son variables cuyo valor no puede cambiarse una vez que se les asigna un valor.

En el siguiente ejemplo, declaramos varias variables y constantes. Las variables (nombre, edad, ciudad y esEstudiante) pueden cambiar su valor en cualquier momento, mientras que las constantes (PI, URL_API y DIAS_SEMANA) tienen valores fijos y no pueden ser modificadas una vez que se les asigna un valor.

Observa cómo al intentar modificar el valor de una constante (PI = 3.14;), se generará un error porque esto no está permitido en JavaScript. Sin embargo, podemos cambiar el valor de las variables (edad = 31;) sin problemas.

Las constantes son útiles cuando necesitamos valores que no cambian a lo largo del programa, como constantes matemáticas (PI) o URLs de APIs. Las variables, por otro lado, nos permiten almacenar y

modificar datos que pueden cambiar durante la ejecución del programa, como información del usuario o datos que se obtienen de una fuente externa.

```
// Variables
let nombre = "Jose";
let edad = 30;
let ciudad = "Hamburgo";
let esEstudiante = true;

// Imprimir los valores de las variables en la consola
console.log("Nombre:", nombre);         // Salida: Nombre: Jose
console.log("Edad:", edad);             // Salida: Edad: 30
console.log("Ciudad:", ciudad);         // Salida: Ciudad: Hamburgo
console.log("Es estudiante:", esEstudiante);  // Salida: Es estudiante: true

// Modificar el valor de una variable
edad = 31;
console.log("Nueva edad:", edad);      // Salida: Nueva edad: 31
```

Figure 2 Ejemplo de Variables

```
// Constantes
const PI = 3.14159;
const URL_API = "https://api.example.com";
const DIAS_SEMANA = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];

// Intentar modificar el valor de una constante (esto generará un error)
// PI = 3.14; // Esto generará un error

// Imprimir los valores de las constantes en la consola
console.log("Valor de PI:", PI);            // Salida: Valor de PI: 3.14159
console.log("URL de la API:", URL_API);     // Salida: URL de la API: https://api.example.com
console.log("Días de la semana:", DIAS_SEMANA); // Salida: Días de la semana: ["Lunes", "Martes", "Miércoles",
                                                //"Jueves", "Viernes", "Sábado", "Domingo"]
```

Figure 3 Ejemplo de Constantes

Tipos de Datos

JavaScript es un lenguaje de programación de tipado dinámico, lo que significa que las variables no tienen un tipo de dato fijo y pueden contener diferentes tipos de datos en diferentes momentos, es decir , una variable puede tener un dato entero en un momento y después ser actualizado el valor por un dato tipo string.

- Números (Number): Representan valores numéricos, ya sea enteros o decimales. Por ejemplo: 10, 3.14, -5.
- Cadenas de texto (String): Representan secuencias de caracteres encerrados entre comillas simples ('') o comillas dobles (""). Por ejemplo: "Hola", 'JavaScript‘.
- Booleanos (Boolean): Representan los valores lógicos verdadero o falso. Los dos valores posibles son true (verdadero) y false (falso).

- Nulos (Null): Representa la ausencia intencional de cualquier objeto o valor.
- Indefinidos (Undefined): Representa una variable que ha sido declarada pero no asignada a ningún valor
- Objetos (Object): Son estructuras de datos que pueden contener propiedades y métodos
- Arreglos (Array): Son colecciones ordenadas de elementos. Los elementos de un arreglo se almacenan en posiciones numeradas y se pueden acceder utilizando un índice. Los arreglos se definen utilizando corchetes [].
- Funciones (Function): Son bloques de código reutilizables que se pueden llamar y ejecutar. Las funciones en JavaScript pueden tener parámetros y devolver un valor.

```
// Variables de diferentes tipos de datos
var numero = 10;
var cadena = "Hola, mundo!";
var booleano = true;
var nulo = null;
var indefinido;
```

Figure 4 Ejemplo de Tipos de Datos

```
// Arreglo
var arreglo = [1, 2, 3, 4, 5];
console.log(arreglo);        // Imprime: [1, 2, 3, 4, 5]
console.log(arreglo[2]);     // Imprime: 3
```

Figure 5 Ejemplo de Tipos de Datos

```
// Objeto
var objeto = {
  nombre: "Juan",
  edad: 25,
  casado: false
};
console.log(objeto);         // Imprime: { nombre: 'Juan', edad: 25, casado: false }
console.log(objeto.nombre); // Imprime: Juan
console.table(objeto);       // Imprime: Valores del objeto en forma de tabla
```

Figure 6 Ejemplo de Tipos de Datos

Funciones

Las funciones son bloques de código reutilizables que realizan tareas específicas. Se pueden definir utilizando la palabra clave function y pueden tomar parámetros y devolver valores.

En los siguientes ejemplos podemos ver que las funciones están diseñadas para realizar diferentes tareas relacionadas con el cálculo y análisis de datos ingresados por el usuario. A continuación, se explica qué hace cada función:

leerDatos: Esta función recibe dos parámetros dato e indice. Se utiliza para solicitar al usuario un número a través de un prompt y almacenar ese valor en la variable dato. El parámetro indice se utiliza para proporcionar un número secuencial que se muestra en el prompt. La función luego devuelve el valor ingresado por el usuario.

sumarDatos: La función sumarDatos recibe dos parámetros, dato y suma. Toma un valor (dato) y lo suma a otro valor acumulado (suma). La función retorna la suma actualizada.

calcularPromedio: Esta función calcula el promedio de un conjunto de datos. Toma tres parámetros: promedio, suma y cantidadDatos. El parámetro suma representa la suma de todos los datos previamente calculados, mientras que cantidadDatos es el número total de datos ingresados. La función divide la suma entre cantidadDatos para obtener el promedio y luego devuelve el resultado.

contarMultiplos5: La función contarMultiplos5 toma dos parámetros, dato y contMult5. Esta función se encarga de contar cuántos números ingresados por el usuario son múltiplos de 5. Si el dato es múltiplo de 5 (es decir, dato % 5 == 0), incrementa el valor de contMult5 en 1. Al final, la función devuelve el valor actualizado de contMult5.

App: Esta función es la principal y muestra una aplicación que permite al usuario ingresar una cantidad determinada de datos numéricos (hasta 100 datos) y realiza varios cálculos con ellos.

```
/**
 *
 * @param {*} dato  // valor a leer
 * @param {*} indice // este es valor que recorre el loop
 * @returns
 */
const leerDatos = (dato,indice) => {
    dato = parseInt(prompt('Ingrese Número '+ (indice+1)));
    return dato;
}
```

Figure 7 Ejemplo de función leerDatos

```
/**
 *
 * @param {*} dato
 * @param {*} suma
 * @returns
 */
const sumarDatos = (dato,suma) => {
    suma = suma + dato;
    return suma;
}
```

Figure 8 Ejemplo de Función sumarDatos

```
/**
 *
 * @param {*} dato
 * @param {*} contMult5
 * @returns
 */
const contarMultiplos5 = ( dato, contMult5 ) => {
    if( dato % 5 == 0 ){ // si dato % 5 == 0
        contMult5++;
    }
    return contMult5
}
```

Figure 9 Ejemplo de función contarMultiplos5

```
/**
 *
 * @param {*} promedio
 * @param {*} suma
 * @param {*} cantidadDatos
 * @returns
 */
const calcularPromedio = ( promedio , suma , cantidadDatos ) => {
    promedio = suma / cantidadDatos ;
    return promedio;
}
```

Figure 10 Ejemplo de función calcularPromedio

```
/**
 *
 * @param {*} porcentajeMult5
 * @param {*} cantidadDatos
 * @param {*} contMult5
 * @returns
 */
const calcularPorcentajeMultiplos5 = (porcentajeMult5,cantidadDatos,contMult5) => {
    porcentajeMult5 =( contMult5/cantidadDatos )*100;
    return porcentajeMult5;
}
```

Figure 11 Ejemplo de función calcularPorcentajeMultiplos5

```
const App = () => {
    var cantidadDatos = 100;
    var datos = [];
    var indice = 0;
    var suma =0;
    var promedio = 0;
    var contMult5 = 0;
    var porcentajeMult5 = 0;
        //1
        datos[indice] =leerDatos(datos[indice],indice);

        while(indice<cantidadDatos && datos[indice]!== 0){
            indice++;
            datos[indice] = leerDatos(datos[indice],indice);
        }
        for (let i = 0; i < datos.length; i++) {
            //2
            suma = sumarDatos(datos[i], suma);
            // parte del 4
            contMult5 = contarMultiplos5(datos[i], contMult5);
        }
            //3
        promedio = calcularPromedio(promedio, suma, datos.length);
            //4
        porcentajeMult5 = calcularPorcentajeMultiplos5(porcentajeMult5,datos.length,contMult5);
        console.log('Promedio de datos ->'+promedio);
        console.log('Suma de datos ->'+suma);
        console.log('Porcentaje de multiplos de 5 es ->'+porcentajeMult5);

}
```

Figure 12 Ejemplo de función principal App

```
// Aplicación en funcionamiento;
App();
```

Figure 13 Evocación de función principal App

Objetos y propiedades

Los objetos en JavaScript son colecciones de datos organizados en pares de clave-valor. Las propiedades de un objeto representan las características del objeto y se acceden utilizando la notación de punto o la notación de corchetes.

En el siguiente ejemplo, hemos definido un objeto llamado "casa" con las siguientes propiedades

- direccion: una cadena de texto que representa la dirección de la casa.
- ciudad: una cadena de texto que representa la ciudad donde se encuentra la casa.
- numHabitaciones: un número entero que indica la cantidad de habitaciones que tiene la casa.
- numBanos: un número entero que indica la cantidad de baños que tiene la casa.
- tieneJardin: un valor booleano (verdadero o falso) que indica si la casa tiene un jardín.
- tieneGaraje: un valor booleano que indica si la casa tiene un garaje.
- precio: un número que representa el precio de la casa en dólares.
- Luego, utilizamos console.log() para mostrar cada una de las propiedades del objeto "casa" por separado en la consola.

```
// Definición del objeto Casa
const casa = {
    direccion: "Calle Principal 123",
    ciudad: "Ciudad Ejemplo",
    numHabitaciones: 3,
    numBanos: 2,
    tieneJardin: true,
    tieneGaraje: false,
    precio: 200000, // Precio en dólares
  };

  // Accediendo a las propiedades del objeto Casa
  console.log("Dirección:", casa.direccion);
  console.log("Ciudad:", casa.ciudad);
  console.log("Número de habitaciones:", casa.numHabitaciones);
  console.log("Número de baños:", casa.numBanos);
  console.log("Tiene jardín:", casa.tieneJardin);
  console.log("Tiene garaje:", casa.tieneGaraje);
  console.log("Precio:", casa.precio);
```

Figure 14 Ejemplo de Objeto Casa

```
const ciudad = {
    nombre: 'Cordoba',
    region: 'Centro de Argentina',
    poblacion: 2000000,
    economia: 'Agraria, Industria, Turismo',
    musica: 'Cuarteto , rock, folklore',
    universidad: 'Universidad Nacional de Cordoba',
    seguridad: 'Región segura para visitar'
}

console.table(ciudad);
```

Figure 15 Ejemplo de Objeto Ciudad

En el siguiente ejemplo, hemos creado un objeto llamado persona que representa a una persona con algunas propiedades, como nombre, edad, ciudad y esEstudiante. Cada propiedad tiene un valor asociado.

Para acceder a las propiedades del objeto, utilizamos la notación de punto (objeto.propiedad) o la notación de corchetes (objeto['propiedad']). En el ejemplo, hemos utilizado la notación de punto para acceder a las propiedades.

También podemos modificar las propiedades existentes del objeto o agregar nuevas propiedades en cualquier momento. En este caso, hemos cambiado la edad de la persona y agregado una nueva propiedad profesion. Además, podemos eliminar una propiedad del objeto utilizando la palabra clave delete.

```javascript
// Crear un objeto representando a una persona
const persona = {
    nombre: "Jose",
    edad: 30,
    ciudad: "Hamburgo",
    esEstudiante: true
  };

  // Acceder a las propiedades del objeto
  console.log("Nombre:", persona.nombre);      // Salida: Nombre: Jose
  console.log("Edad:", persona.edad);          // Salida: Edad: 30
  console.log("Ciudad:", persona.ciudad);      // Salida: Ciudad: Hamburgo
  console.log("Es estudiante:", persona.esEstudiante); // Salida: Es estudiante: true

  // Modificar una propiedad del objeto
  persona.edad = 31;
  console.log("Nueva edad:", persona.edad);    // Salida: Nueva edad: 31

  // Agregar una nueva propiedad al objeto
  persona.profesion = "Ingeniero";
  console.log("Profesión:", persona.profesion); // Salida: Profesión: Ingeniero

  // Eliminar una propiedad del objeto
  delete persona.ciudad;
  console.log("Ciudad:", persona.ciudad);      // Salida: Ciudad: undefined
```

Figure 16 Ejemplo de Objetos y Propiedades

Eventos y manipulación del DOM

JavaScript permite interactuar con elementos HTML y responder a eventos, como clics de botones o envío de formularios. El DOM (Modelo de Objetos del Documento) es una representación en memoria de la estructura HTML de una página web que JavaScript puede manipular para cambiar su contenido y apariencia.

En el siguiente ejemplo creamos 2 elementos inputs, primero definimos una función para reutilizar código y crear inputs desde allí.

```javascript
// Función para crear un elemento input
function createInputElement(type, name) {
    const inputElement = document.createElement('input');
    inputElement.type = type;
    inputElement.name = name;
    return inputElement;
}

// Crear los inputs
const input1 = createInputElement('text', 'input1');
const input2 = createInputElement('email', 'input2');

// Obtener el contenedor donde agregar los inputs
const container = document.getElementById('container');

// Agregar los inputs al contenedor
container.appendChild(input1);
container.appendChild(input2);
```

Condicionales y bucles

JavaScript admite estructuras de control como if-else, switch, for, while y do-while, que permiten tomar decisiones y repetir acciones según ciertas condiciones.

```
const edad = 25;

if (edad < 18) {
  console.log("Eres menor de edad.");
} else if (edad >= 18 && edad < 65) {
  console.log("Eres mayor de edad pero no llegas a la jubilación.");
} else {
  console.log("Eres jubilado/a.");
}
```

Figure 17 Ejemplo If-else

```
for (let i = 0; i < 5; i++) {
    console.log("Iteración:", i);
}
```

Figure 18 Ejemplo de Loop For

```
let contador = 0;
  while (contador < 5) {
    console.log("Contador:", contador);
    contador++;
  }
```

Figure 19 Ejemplo de loop While

```
do {
  console.log("Número:", num);
  num--;
} while (num > 0);
```

Figure 20 Ejemplo de loop do-while

Bibliotecas y frameworks

Existen muchas bibliotecas y frameworks de JavaScript, como React, Angular, Vue.js, y jQuery, que facilitan el desarrollo de aplicaciones más complejas y eficientes, proporcionando funcionalidades adicionales y mejorando la organización del código.

Bibliotecas frontend:

- React: Una biblioteca desarrollada por Facebook para construir interfaces de usuario interactivas y reutilizables. Utiliza el concepto de componentes y es muy popular en el desarrollo de aplicaciones web modernas.
- Vue.js: Un framework progresivo para la creación de interfaces de usuario. Es conocido por su facilidad de uso y su enfoque gradual, lo que permite que se integre fácilmente en proyectos existentes.
- Angular: Un framework desarrollado por Google que proporciona una estructura sólida para construir aplicaciones web complejas y escalables. Utiliza el patrón de diseño MVC (Model-View-Controller) y ofrece una amplia variedad de herramientas y características.
- Ember.js: Un framework con una fuerte opinión sobre cómo se deben estructurar las aplicaciones web. Ofrece una arquitectura sólida y una amplia gama de funcionalidades listas para usar.

Bibliotecas y frameworks para el manejo de estado:

- Redux: Una biblioteca para manejar el estado de las aplicaciones, especialmente en combinación con React. Ayuda a mantener un estado predecible y gestionado centralmente.
- MobX: Otra biblioteca de manejo de estado que se puede utilizar con React o con otras bibliotecas de interfaz de usuario. Facilita la sincronización entre el estado y la interfaz de usuario.
- Bibliotecas para manipulación del DOM:
- jQuery: Una biblioteca que simplifica la manipulación del DOM y el manejo de eventos en JavaScript. Aunque su popularidad ha disminuido con el tiempo, sigue siendo ampliamente utilizada en proyectos heredados.
- Bibliotecas para visualización de datos:
- D3.js: Una biblioteca que permite crear visualizaciones de datos interactivas y complejas utilizando HTML, SVG y CSS.
- Frameworks para desarrollo de aplicaciones móviles:

- React Native: Una extensión de React que permite construir aplicaciones móviles multiplataforma utilizando JavaScript y React.
- Ionic: Un framework que utiliza tecnologías web como HTML, CSS y JavaScript para crear aplicaciones móviles híbridas.

Asincronía

JavaScript es conocido por su naturaleza asincrónica, lo que significa que puede manejar múltiples tareas al mismo tiempo sin bloquear el hilo de ejecución principal. Esto se logra utilizando funciones de devolución de llamada (callbacks), Promesas o async/await.

En todos estos ejemplos, estamos simulando operaciones asíncronas mediante el uso de setTimeout para esperar unos segundos antes de obtener los datos. Los enfoques de Callbacks, Promesas y Async/Await nos permiten manejar tareas asíncronas de manera más legible y estructurada, evitando el bloqueo del hilo principal de JavaScript y mejorando la experiencia de usuario al permitir que otras tareas se ejecuten mientras se espera la respuesta asíncrona.

```
// Callback
function obtenerDatos(callback) {
    setTimeout(function() {
      console.log("Datos obtenidos");
      const datos = { id: 1, nombre: "Ejemplo" };
      callback(datos);
    }, 2000);
  }

  function procesarDatos(datos) {
    console.log("Procesando datos:", datos);
  }

  obtenerDatos(procesarDatos);
  console.log("Esta línea se ejecuta antes de obtener los datos");
```

Figure 21 Ejemplo de Calback

```
// Promesas
function obtenerDatos() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      console.log("Datos obtenidos");
      const datos = { id: 1, nombre: "Ejemplo" };
      resolve(datos);
      // Si hubiera un error: reject(new Error("Ocurrió un error"));
    }, 2000);
  });
}

obtenerDatos()
  .then(function(datos) {
    console.log("Procesando datos:", datos);
  })
  .catch(function(error) {
    console.error("Error:", error.message);
  });
console.log("Esta línea se ejecuta antes de obtener los datos");
```

Figure 22 Ejemplo de Promesas

```
// Asinc/Await
function esperar(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function obtenerDatos() {
  await esperar(2000);
  console.log("Datos obtenidos");
  return { id: 1, nombre: "Ejemplo" };
}

async function procesarDatos() {
  try {
    const datos = await obtenerDatos();
    console.log("Procesando datos:", datos);
  } catch (error) {
    console.error("Error:", error.message);
  }
}

procesarDatos();
console.log("Esta línea se ejecuta antes de obtener los datos");
```

Figure 23 Ejemplo de Asinc/Await