

Unidad 5: Funciones Constructoras y Almacenamiento

# **Función Constructora**

## **¿Qué es una función constructora?**

### **Definición y Propósito**

En JavaScript, una función constructora es fundamentalmente una función especial que se utiliza para inicializar un nuevo objeto. La idea detrás de una función constructora es proporcionar una forma de crear objetos que modelen algo en el mundo real o en la lógica de un programa, como una Persona, un Pedido, o un Coche, por ejemplo.

Antes de la introducción de las clases en ECMAScript 2015 (ES6), las funciones constructoras eran la manera estándar en JavaScript para crear tipos de datos que simularan las clases como se conocen en otros lenguajes de programación orientados a objetos como Java o C#. A través de las funciones constructoras, se podían definir propiedades y métodos específicos para un tipo de objeto, reutilizando la misma estructura base cada vez que se instanciaba un objeto nuevo.

**Ejemplo de Uso Pre-ES6**

| function Persona(nombre, edad, calle) {  this.nombre = nombre;  this.edad = edad;  this.calle = calle;  this.describir = function() {  return `Nombre: ${this.nombre}, Edad: ${this.edad}, Calle: ${this.calle}`;  };  }  // Crear un nuevo objeto Persona  var persona1 = new Persona("Ana", 25, "Calle Falsa 123");  console.log(persona1.describir()); |
| --- |

En este ejemplo, Persona es una función constructora. Las propiedades nombre, edad, y calle se asignan al objeto que this referencia dentro de la función constructora. new Persona crea una nueva instancia de Persona, y this en el contexto de la función constructora se refiere a esta nueva instancia.

**Comparación con Clases en ES6**

Las clases en ES6 fueron introducidas para proporcionar una sintaxis más clara y concisa para alcanzar el mismo resultado que las funciones constructoras. La sintaxis de clase es más familiar para los programadores que vienen de otros lenguajes de programación orientados a objetos y ayuda a manejar la herencia de manera más intuitiva.

| class Persona {  constructor(nombre, edad, calle) {  this.nombre = nombre;  this.edad = edad;  this.calle = calle;  }  describir() {  return `Nombre: ${this.nombre}, Edad: ${this.edad}, Calle: ${this.calle}`;  }  }  // Crear un nuevo objeto Persona  const persona1 = new Persona("Ana", 25, "Calle Falsa 123");  console.log(persona1.describir()); |
| --- |

**Conclusión**

La introducción de las clases no ha eliminado la funcionalidad de las funciones constructoras, pero sí ha proporcionado una forma más elegante y simple de lograr objetivos similares. Las clases hacen el código más legible y fácil de entender, especialmente en proyectos grandes y complejos, y facilitan la implementación de conceptos avanzados de programación orientada a objetos como la herencia y el polimorfismo.

## **LocalStorage: Guardar**

### **Introducción a Storage**

**Las API de almacenamiento web: localStorage y sessionStorage**

El almacenamiento web proporciona métodos y protocolos para almacenar datos de manera local en el navegador del usuario. Las dos principales herramientas que ofrece son: localStorage y sessionStorage.

**¿Qué son localStorage y sessionStorage?**

* **localStorage**: Permite almacenar datos de forma indefinida hasta que sean explícitamente eliminados por el usuario o por el sitio web que los guardó. Los datos persisten incluso después de cerrar el navegador y reiniciar el sistema.
* **sessionStorage**: Similar a localStorage en funcionalidad pero con una diferencia clave: los datos almacenados en sessionStorage se eliminan cuando se cierra la pestaña del navegador. Esto significa que los datos persisten durante la sesión de la pestaña, pero no entre sesiones.

### **Diferencias clave**

| **Característica** | **localStorage** | **sessionStorage** |
| --- | --- | --- |
| **Persistencia** | Permanente hasta eliminación manual o por el sitio | Persiste solo durante la sesión de la pestaña del navegador |
| **Compartición** | Disponible en todas las pestañas y ventanas | Disponible solo en la pestaña que creó el almacenamiento |
| **Uso típico** | Guardar preferencias del usuario a largo plazo | Guardar información sobre el estado de la sesión actual |

**¿Cuándo usar localStorage y sessionStorage?**

* **localStorage** es ideal para:
  + Guardar preferencias de usuario que no requieren ser borradas al cerrar el navegador.
  + Almacenar datos para el autocompletado de formularios en visitas futuras.
  + Guardar tokens de autenticación y configuraciones para uso a largo plazo.
* **sessionStorage** se recomienda para:
  + Información que debe desaparecer después de la sesión, como datos de un formulario en progreso en una compra o configuraciones temporales de visualización.
  + Datos sensibles que no deben persistir más allá de la sesión actual, como detalles de una transacción financiera.

### **Ejemplo práctico**

**localStorage:**

| // Guardar datos  localStorage.setItem('usuario', 'JuanPerez');  // Obtener datos  let usuario = localStorage.getItem('usuario');  console.log(usuario); // 'JuanPerez'  // Eliminar datos  localStorage.removeItem('usuario'); |
| --- |

**sessionStorage:**

| // Guardar datos  sessionStorage.setItem('detalleCompra', 'Libro de JavaScript');  // Obtener datos  let detalle = sessionStorage.getItem('detalleCompra');  console.log(detalle); // 'Libro de JavaScript'  // Eliminar datos al cerrar la pestaña  sessionStorage.removeItem('detalleCompra'); |
| --- |

Utilizar estas herramientas adecuadamente permite mejorar la experiencia del usuario manteniendo la información necesaria al alcance sin recurrir a servidor, optimizando así la performance y recursos del sistema.

## **Guardar datos en LocalStorage**

**Almacenamiento de Datos en localStorage con setItem**

El objeto localStorage es una herramienta poderosa para almacenar datos directamente en el navegador del usuario, lo que permite preservar información entre sesiones de navegación. A continuación, exploraremos cómo utilizar el método setItem para guardar diferentes tipos de datos, como cadenas de texto (strings) y números.

**Uso Básico de setItem**

El método setItem de localStorage permite guardar pares clave-valor, donde tanto la clave como el valor deben ser cadenas de texto. La sintaxis general es:

| localStorage.setItem('clave', 'valor'); |
| --- |

### **Ejemplos de Uso**

* **Almacenar una cadena de texto:**

Guarda un saludo en el almacenamiento local:

| localStorage.setItem('saludo', 'Hola, mundo!'); |
| --- |

* **Almacenar un número:**

Aunque localStorage solo puede almacenar cadenas de texto, puedes guardar números como texto y luego convertirlos de vuelta a números cuando los leas:

| localStorage.setItem('puntuacion', '10'); |
| --- |

Para recuperar el número y convertirlo a su tipo original:

| let puntuacion = parseInt(localStorage.getItem('puntuacion'));  console.log(puntuacion); // 10 |
| --- |

* **Almacenar datos complejos (como objetos o arrays):**

Para guardar objetos o arrays, primero debes convertirlos a una cadena JSON:

| let usuario = {  nombre: 'Ana',  edad: 25  };  localStorage.setItem('usuario', JSON.stringify(usuario)); |
| --- |

Para recuperarlo, usa JSON.parse:

| let usuarioRecuperado = JSON.parse(localStorage.getItem('usuario'));  console.log(usuarioRecuperado.nombre); // Ana |
| --- |

### **Consideraciones al usar localStorage**

* **Limitación de Tipo**: localStorage solo almacena cadenas. Cualquier otro tipo de dato, incluidos los números, booleanos o arrays, debe ser convertido a cadena antes de almacenarse y revertido a su tipo original al ser leído.
* **Seguridad**: No es seguro almacenar datos sensibles, como información de autenticación o datos personales, porque el localStorage es accesible desde el cliente y podría ser leído por cualquier script en la página.
* **Capacidad de Almacenamiento**: localStorage tiene una capacidad limitada (aproximadamente 5MB por dominio), por lo que es importante gestionar lo que se almacena y evitar guardar grandes cantidades de datos.

El uso efectivo de localStorage con el método setItem puede mejorar significativamente la experiencia del usuario al permitir la personalización y la persistencia del estado entre sesiones de navegación.

## **Eliminación de Datos en localStorage y sessionStorage**

El almacenamiento web en los navegadores modernos se realiza a través de dos mecanismos principales: localStorage y sessionStorage. Estos permiten almacenar datos de manera local sin interactuar con el servidor. En algún momento, puede ser necesario eliminar estos datos, ya sea de manera individual o completamente. A continuación, se detallan los métodos para realizar estas acciones.

**Uso de removeItem para eliminar datos específicos**

removeItem es un método que permite eliminar un ítem específico del almacenamiento, utilizando la clave del dato que deseas eliminar.

**Sintaxis básica:**

| localStorage.removeItem('clave');  sessionStorage.removeItem('clave'); |
| --- |

**Ejemplo de uso:**

Para eliminar un usuario guardado en localStorage:

| localStorage.removeItem('usuario'); |
| --- |

Para eliminar una sesión de usuario específica en sessionStorage:

| sessionStorage.removeItem('sesion\_usuario'); |
| --- |

Este método es útil cuando conoces la clave del ítem que deseas eliminar y no necesitas eliminar todos los datos almacenados.

**Uso de clear para eliminar todos los datos**

El método clear elimina todos los datos almacenados en localStorage o sessionStorage, sin distinción de claves o valores. Este método es radical y útil cuando deseas limpiar completamente el almacenamiento para el dominio dado.

**Sintaxis básica:**

| localStorage.clear();  sessionStorage.clear(); |
| --- |

**Ejemplo de uso:**

Para eliminar todos los datos guardados en localStorage:

| localStorage.clear(); |
| --- |

Para eliminar todos los datos en sessionStorage, especialmente útil cuando una sesión de navegador se cierra y quieres asegurar que no queden datos residuales:

| sessionStorage.clear(); |
| --- |

**Consideraciones importantes**

1. **Persistencia de datos**: Mientras localStorage mantiene la información sin importar el cierre del navegador, sessionStorage lo hace únicamente durante la vida de la pestaña o ventana.
2. **Seguridad**: No almacenes información sensible como contraseñas o información financiera sin adecuadas medidas de cifrado, dado que tanto localStorage como sessionStorage son accesibles desde el lado del cliente.
3. **Límites de almacenamiento**: Ambos métodos tienen limitaciones de almacenamiento que, aunque suficientes para la mayoría de las tareas, deben considerarse al planificar el almacenamiento de grandes cantidades de datos.

**Prácticas recomendadas**

* **Validar claves antes de eliminar**: Antes de eliminar datos, asegúrate de que la clave exista para evitar errores.
* **Gestión de almacenamiento**: Regularmente verifica lo que está almacenado y elimina lo que ya no sea necesario para optimizar el rendimiento de la aplicación.
* **Informar a los usuarios**: Si la aplicación depende significativamente de estos mecanismos de almacenamiento, considera informar a los usuarios antes de realizar limpiezas masivas de datos.

La correcta administración de localStorage y sessionStorage mejora el rendimiento y la experiencia del usuario, manteniendo la aplicación eficiente y segura.

## **JSON y Almacenamiento de Objetos**

### **¿Qué es JSON?**

JSON (JavaScript Object Notation) es un formato de texto ligero para intercambiar datos. Es fácil para los humanos leer y escribir, y fácil para las máquinas parsear e interpretar. JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza convenciones que son familiares para los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

**Uso de JSON en localStorage y sessionStorage**

En el contexto de las aplicaciones web, es común que necesites almacenar datos en el navegador del usuario. Aquí es donde localStorage y sessionStorage entran en juego. Ambos proporcionan un espacio para almacenar datos en forma de clave-valor, pero con algunas diferencias:

* **localStorage**: Almacena datos sin fecha de expiración. Los datos no se eliminarán cuando se cierre el navegador y estarán disponibles entre las sesiones de navegación.
* **sessionStorage**: Almacena datos solo para una sesión de página. Los datos se eliminan cuando se cierra la pestaña o el navegador.

Ambos, localStorage y sessionStorage, solo pueden almacenar cadenas (strings), por lo que para almacenar objetos JavaScript complejos, es necesario serializarlos a JSON. Utilizando JSON.stringify() puedes convertir un objeto JavaScript en una cadena JSON, y con JSON.parse(), puedes convertir una cadena JSON de vuelta a un objeto JavaScript.

### **Ejemplo práctico**

Supongamos que quieres almacenar un objeto de usuario en localStorage para mantener al usuario logueado entre sesiones.

| // Objeto usuario  const usuario = {  id: "1234",  nombre: "Juan",  correo: "juan@example.com"  };  // Convertir el objeto usuario a una cadena JSON para almacenamiento  localStorage.setItem('usuario', JSON.stringify(usuario));  // Recuperar el objeto usuario de localStorage y convertirlo de nuevo a un objeto JavaScript  const usuarioAlmacenado = JSON.parse(localStorage.getItem('usuario'));  console.log(usuarioAlmacenado); |
| --- |

Este método permite una manera eficiente de almacenar objetos complejos y estructuras de datos en el navegador del usuario, manteniendo la persistencia de la sesión o entre sesiones según sea necesario.

### **Recuperar y convertir JSON a objetos**

**¿Qué es JSON.parse?**

JSON.parse es una función incorporada en JavaScript que se utiliza para convertir una cadena de texto en formato JSON a un objeto de JavaScript. Este proceso es conocido como deserialización, y es fundamental cuando recuperamos datos almacenados que necesitan ser manipulados o accesados como objetos dentro de nuestras aplicaciones.

**Uso Práctico de JSON.parse**

Cuando almacenamos datos en localStorage o sessionStorage, estos datos deben ser cadenas. Si tenemos objetos JavaScript, primero los convertimos en una cadena JSON usando JSON.stringify(). Para revertir este proceso, utilizamos JSON.parse().

**Ejemplo Práctico:**

Imagina que tienes un sitio web de comercio electrónico y necesitas almacenar la información del carrito de compras del usuario entre sesiones. Podrías almacenar esta información en localStorage como un string JSON y recuperarlo cada vez que el usuario visite la página.

| // Supongamos que este es el carrito de compras del usuario  const carrito = {  items: [  { id: 1, producto: "Libro", cantidad: 2, precio: 15.00 },  { id: 2, producto: "Lápiz", cantidad: 10, precio: 1.50 }  ]  };  // Convertimos el objeto carrito en una cadena JSON y lo almacenamos  localStorage.setItem('carrito', JSON.stringify(carrito));  // Recuperamos la cadena JSON del almacenamiento y la convertimos de nuevo a un objeto JavaScript  const carritoAlmacenado = JSON.parse(localStorage.getItem('carrito'));  // Ahora podemos acceder y manipular los datos del carrito como un objeto JavaScript  console.log(`Total de productos: ${carritoAlmacenado.items.length}`); |
| --- |

Este método permite que la interfaz de usuario del sitio web mantenga la consistencia del carrito de compras sin necesidad de hacer consultas constantes a un servidor, mejorando la experiencia de usuario y reduciendo la carga en el servidor.

**Conclusiones**

El uso de JSON.parse() es crucial para aplicaciones que dependen del almacenamiento local para gestionar estados o configuraciones entre sesiones. Facilita la manipulación de datos estructurados y mejora la eficiencia de las aplicaciones web modernas.