



# JavaScript

## Objetos y Destructuración

*< Ing := Carlos H. Rueda C++ />*

\* \* \* \* \*

# 1. Objetos

En JavaScript, los objetos son una colección de datos relacionados que se pueden utilizar para representar cualquier cosa, como personas, lugares, cosas o incluso conceptos abstractos.



## Ejemplos de objetos

- **Una persona:** nombre, edad, altura, peso.
- **Un coche:** marca, modelo, color, peso.
- **Un libro:** título, autor, género.
- **Un número:** valor, signo.
- **Una función:** parámetros, cuerpo.

\* \* \* \* \*





# 1.1 Declaración de objetos

Para declarar un objeto en JavaScript, se utilizan las llaves `{}`. Dentro de las llaves, se especifican las propiedades y métodos del objeto, separados por comas `,`. Cada propiedad o método se define con una clave y un valor separados por dos puntos `:`.

## Ejemplo de declaración de objetos

```
const persona = {  
  nombre: 'Dani',  
  edad: 30,  
  esTrabajador: true  
};
```

## 1.2 Propiedades de objetos

👁 Las propiedades de un objeto son datos que se asocian a una clave. Las propiedades se pueden utilizar para representar cualquier tipo de dato, incluidos números, cadenas, objetos, arrays.

\* \* \* \* \*



## Ejemplo de propiedades de objetos

```
const persona = {  
  nombre: 'Dani',  
  edad: 30,  
  esTrabajador: true,  
  familia: ['Miguel', 'Maria'],  
  direccion: {  
    calle: 'Calle de la piruleta',  
    numero: 13,  
    ciudad: 'Barcelona'  
  }  
}
```



## 1.3 Métodos de objetos

👁👁 Los métodos de un objeto son funciones que se asocian a una clave. Los métodos se pueden utilizar para realizar acciones sobre el objeto.

\* \* \* \* \*



## Ejemplo de métodos de objetos

```
const persona = {  
  nombre: 'Dani',  
  edad: 30,  
  esTrabajador: true,  
  familia: ['Miguel', 'Maria'],  
  direccion: {  
    calle: 'Calle de la piruleta',  
    numero: 13,  
    ciudad: 'Barcelona'  
  },  
  caminar: function () {  
    console.log('Estoy caminando');  
  }  
};
```



## 1.4 Acceso a propiedades y métodos de objetos

👁👁 Las propiedades y métodos de un objeto se pueden acceder utilizando el operador `.`

\* \* \* \* \*



## Ejemplo de acceso a propiedades y métodos de objetos

```
const persona = {  
  nombre: 'Dani',  
  edad: 30,  
  esTrabajador: true,  
  familia: ['Miguel', 'Maria'],  
  direccion: {  
    calle: 'Calle de la piruleta',  
    numero: 13,  
    ciudad: 'Barcelona'  
  },  
  caminar: function () {  
    console.log('Estoy caminando');  
  }  
};  
  
console.log(persona.nombre); // Dani  
console.log(persona.esTrabajador); // true  
persona.caminar(); // Estoy caminando
```

## 1.5 this

### ¿Qué significa "this"?

El objeto al que hace referencia "**this**" depende de cómo se invoque (utilice o llame).

La interpretación de la palabra clave "this" varía según la manera en que se emplea:

- En un método de objeto, "this" se refiere al objeto.
- Solamente, "this" hace referencia al objeto global.
- En una función, "this" se refiere al objeto global.

\* \* \* \* \*





## ¿Qué significa "this"?

La interpretación de la palabra clave "this" varía según la manera en que se emplea:

- En una función, en modo estricto, "this" es indefinido.
- En un evento, "this" se refiere al elemento que recibió el evento.
- Métodos como `call()`, `apply()` y `bind()` pueden referir "this" a cualquier objeto.

\* \* \* \* \*

## 👁 Ejemplo con this

```
const empleado = {  
  nombre: "Ana",  
  apellido: "Gómez",  
  identificación: 7890,  
  nombreCompleto: function() {  
    return this.nombre + " " + this.apellido;  
  }  
};
```

En el ejemplo anterior, **this** se refiere al objeto **empleado**.

- **this.nombre** significa la propiedad **nombre** de **this**.
- **this.nombre** significa la propiedad **nombre** del objeto **person**.



## 2. JSON

**JSON (JavaScript Object Notation)** es un formato ligero de intercambio de datos. Es fácil para los humanos leer y escribir, y fácil para las máquinas analizar y generar. Se basa en un subconjunto del lenguaje de programación JavaScript, utilizando pares clave-valor para almacenar datos. JSON se utiliza comúnmente para intercambiar datos entre aplicaciones web y servidores.



\* \* \* \* \*



## Algunos de los beneficios de usar JSON:

- **Fácil de leer y escribir:** JSON es fácil para los humanos leer y escribir porque utiliza una sintaxis simple que se basa en literales de objetos JavaScript.
- **Fácil de analizar y generar:** JSON es fácil para las máquinas analizar y generar porque tiene una sintaxis bien definida y hay muchas bibliotecas disponibles para trabajar con JSON.

\* \* \* \* \*

## Algunos de los beneficios de usar JSON:

- **Ligero:** JSON es un formato ligero de intercambio de datos porque utiliza un esquema de codificación simple.
- **Independiente del lenguaje:** JSON es independiente del lenguaje, lo que significa que se puede utilizar con cualquier lenguaje de programación.

\* \* \* \* \*





## Algunos de los usos comunes de JSON:

- **Intercambio de datos entre aplicaciones web y servidores:** JSON se utiliza comúnmente para intercambiar datos entre aplicaciones web y servidores. Por ejemplo, una aplicación web podría enviar una solicitud con formato JSON a un servidor para recuperar datos, o un servidor podría enviar una respuesta con formato JSON a una aplicación web.

\* \* \* \* \*





## Algunos de los usos comunes de JSON:

- **Almacenamiento de datos en archivos:** JSON a menudo se utiliza para almacenar datos en archivos. Por ejemplo, una aplicación web podría almacenar datos de usuario en un archivo JSON.
- **Configuración de aplicaciones:** JSON a veces se utiliza para configurar aplicaciones. Por ejemplo, una aplicación podría almacenar sus configuraciones en un archivo JSON.

\* \* \* \* \*



## 2.1 JSON.stringify()

Ejemplo de cómo usar JSON en JavaScript:

```
const persona = {  
  nombre: 'Juan Pérez',  
  edad: 30,  
  ocupación: 'Ingeniero de software'  
};  
  
const cadenaJSON = JSON.stringify(persona);  
console.log(cadenaJSON);
```

El anterior código imprimirá la siguiente cadena JSON en la consola:

## JSON

```
{  
  "nombre": "Juan Pérez",  
  "edad": 30,  
  "ocupación": "Ingeniero de software"  
}
```

El método `JSON.stringify()` convierte un objeto JavaScript en una cadena JSON.



## Ejemplo de cómo analizar una cadena JSON en JavaScript:

```
const cadenaJSON = '{"nombre": "Juan Pérez", "edad": 30, "ocupación": "Ingeniero de software"}';  
const persona = JSON.parse(cadenaJSON);  
console.log(persona);
```

Este código analizará la cadena JSON y creará un objeto JavaScript a partir de ella. El método `JSON.parse()` convierte una cadena **JSON** en un objeto JavaScript.

\* \* \* \* \*

## 2.2 El formato JSON se evalúa como objetos de JavaScript

👁 El formato JSON es sintácticamente idéntico al código utilizado para crear objetos en JavaScript. Debido a esta similitud, un programa en JavaScript puede convertir fácilmente datos en formato JSON en objetos JavaScript nativos.

\* \* \* \* \*

# Reglas de Sintaxis JSON

- Los datos se encuentran en pares de nombre/valor.
- Los datos se separan por comas.
- Las llaves curvas contienen objetos.
- Los corchetes cuadrados contienen arrays.

\* \* \* \* \*





## Datos JSON - nombre y valor

Los datos JSON se escriben como pares nombre-valor, al igual que las propiedades de los objetos JavaScript. Un par nombre-valor consta de un nombre de campo (entre comillas dobles), seguido de dos puntos, seguido de un valor:

```
"nombre": "Juan"
```

## Ejemplo de datos JSON

```
{  
  "nombre": "Juan Pérez",  
  "edad": 30,  
  "ocupación": "Ingeniero de software"  
}
```

En este ejemplo, cada par nombre-valor se representa como una línea independiente. El nombre del campo está entre comillas dobles, seguido de dos puntos y luego el valor del campo.

\* \* \* \* \*

# Objetos JSON

Los objetos JSON se escriben entre llaves. Al igual que en JavaScript, los objetos pueden contener múltiples pares nombre-valor:

```
{  
  "nombre": "Juan Pérez",  
  "edad": 30,  
  "ocupación": "Ingeniero de software"  
}
```

\* \* \* \* \*



```
{  
  "nombre": "Juan Pérez",  
  "edad": 30,  
  "ocupación": "Ingeniero de software"  
}
```

En este ejemplo, el objeto JSON contiene tres pares nombre-valor. Cada par nombre-valor está representado como una propiedad del objeto, con el nombre de la propiedad entre comillas dobles y el valor de la propiedad sin comillas.

\* \* \* \* \*



# Arreglos JSON

```

{
  "empleados": [
    {
      "nombre": "Juan Pérez",
      "apellido": "López"
    },
    {
      "nombre": "Ana",
      "apellido": "González"
    },
    {
      "nombre": "Pedro",
      "apellido": "Sánchez"
    }
  ]
}

```

En el ejemplo anterior, el objeto "empleados" es un arreglo. Contiene tres objetos. Cada objeto es un registro de una persona (con un nombre y un apellido).

\* \* \* \* \*



# Conversión de texto JSON a objeto JavaScript

Un uso común de JSON es leer datos de un servidor web y mostrarlos en una página web. Para simplificar, se puede demostrar usando una cadena como entrada. Primero, cree una cadena JavaScript que contenga sintaxis JSON:

```
let texto = '{ "empleados" : [' +  
  '{ "nombre":"Juan" , "apellido":"López" },' +  
  '{ "nombre":"Ana" , "apellido":"González" },' +  
  '{ "nombre":"Pedro" , "apellido":"Sánchez" } ]}';
```

Luego, use la función incorporada de JavaScript **JSON.parse()** para convertir la cadena en un objeto JavaScript:

```
const obj = JSON.parse(texto);
```

En este ejemplo, la variable **obj** contendrá un objeto JavaScript que representa los datos JSON en la cadena **texto**. El objeto tendrá una propiedad **empleados** que es una matriz de objetos, donde cada objeto representa a un empleado.

\* \* \* \* \*



## 2.3 new FormData

👁 El constructor `new FormData()` crea un objeto `FormData`. Un objeto `FormData` es una colección de pares `nombre-valor` que se pueden usar para enviar datos a través de una solicitud HTTP. El constructor `new FormData()` puede tomar dos argumentos opcionales:

- `form`: Un elemento HTML `<form>`. Si se especifica este argumento, el objeto `FormData` se inicializará con los datos del formulario.
- `submitter`: Un elemento HTML `<input>`. Si se especifica este argumento, el objeto `FormData` se inicializará con el valor del elemento `<input>`. Por ejemplo, el siguiente código crea un objeto `FormData` que contiene los datos de un formulario:



```
const form = document.querySelector("form");  
const formData = new FormData(form);
```

Este código crea un objeto `FormData` llamado `formData`. El objeto `formData` contiene los datos de todos los campos del formulario `form`. El objeto `FormData` tiene varios métodos que se pueden usar para agregar, eliminar y obtener datos:

- `append()`: Agrega un nuevo `nombre-valor` al objeto `FormData`.
- `delete()`: Elimina un `nombre-valor` del objeto `FormData`.
- `get()`: Obtiene el valor de `nombre-valor` del objeto `FormData`.
- `getAll()`: Obtiene todos los valores `nombre-valor` del objeto `FormData`.

Por ejemplo, el siguiente código agrega un nuevo par `nombre-valor` al objeto `formData`:

```
formData.append("nombre", "Juan Pérez");
```

\* \* \* \* \*





Este código agrega un nuevo par `nombre-valor` al objeto `formData`. El nombre del par es `nombre` y el valor del par es `Juan Pérez`. El siguiente código elimina un par `nombre-valor` del objeto `formData`:

```
formData.delete("nombre");
```

\* \* \* \* \*





Este código elimina el par `nombre-valor` del objeto `formData`. El siguiente código obtiene el valor de un par `nombre-valor` del objeto `formData`:

```
const nombre = formData.get("nombre");
```

\* \* \* \* \*



Este código obtiene el valor del par `nombre-valor` del objeto `formData`. El valor del par es `Juan Pérez`. El siguiente código obtiene todos los valores de un par `nombre-valor` del objeto `formData`:

```
const nombres = formData.getAll("nombre");
```

\* \* \* \* \*



# 3. Desestructuración

La desestructuración en JavaScript es una característica que permite asignar los valores de una estructura de datos a variables individuales. La desestructuración se puede usar con objetos, matrices, conjuntos y mapas.

\* \* \* \* \*





## 3.1 Desctructuración de objetos

La desestructuración de objetos se puede usar para asignar los valores de las propiedades de un objeto a variables individuales.

\* \* \* \* \*



Por ejemplo, el siguiente código asigna los valores de las propiedades `nombre` y `edad` del objeto `persona` a las variables :

```
const persona = {  
  nombre: "Juan Pérez",  
  edad: 30  
};  
  
const { nombre, edad } = persona;  
  
console.log(nombre); // "Juan Pérez"  
console.log(edad); // 30
```

En este ejemplo, el operador `{ ... }` se usa para especificar las variables a las que se asignarán los valores de las propiedades.

## 3.2 Desestructuración de matrices

La desestructuración de matrices se puede usar para asignar los valores de los elementos de una matriz a variables individuales.



\* \* \* \* \*



Por ejemplo, el siguiente código asigna los primeros dos elementos de la matriz `numeros` a las variables `numero1` y `numero2`:

```
const numeros = [1, 2, 3, 4, 5];  
  
const [numero1, numero2] = numeros;  
  
console.log(numero1); // 1  
console.log(numero2); // 2
```

En este ejemplo, el operador `[ ... ]` se usa para especificar las variables a las que se asignarán los valores de los elementos de la matriz.

## 3.2 Desestructuración de conjuntos

La desestructuración de conjuntos se puede usar para asignar los valores de los elementos de un conjunto a variables individuales.

\* \* \* \* \*



Por ejemplo, el siguiente código asigna los primeros dos elementos del conjunto `numeros` a las variables `numero1` y `numero2`:

```
const numeros = new Set([1, 2, 3, 4, 5]);  
  
const [numero1, numero2] = numeros;  
  
console.log(numero1); // 1  
console.log(numero2); // 2
```

En este ejemplo, el operador `[ ... ]` se usa para especificar las variables a las que se asignarán los valores de los elementos del conjunto.



## 3.3 Desestructuración de mapas

La desestructuración de mapas se puede usar para asignar los valores de las claves y los valores de un mapa a variables individuales.

\* \* \* \* \*



Por ejemplo, el siguiente código asigna las claves `nombre` y `edad` y sus valores correspondientes del mapa `persona` a las variables `nombre` y `edad`:

```
const persona = new Map([
  ["nombre", "Juan Pérez"], ["edad", 30]
]);

const { nombre, edad } = persona;
console.log(nombre); // "Juan Pérez"
console.log(edad); // 30
```

En este ejemplo, el operador `{ ... }` se usa para especificar las variables a las que se asignarán los valores de las claves y los valores del mapa.



## 3.4 Ventajas y uso

### Ventajas de la destructuración

La destructuración tiene varias ventajas, entre ellas:

- Hace el código más conciso y fácil de leer.
- Reduce la cantidad de código necesario para acceder a los valores de una estructura de datos.
- Puede mejorar el rendimiento del código al evitar la creación de objetos temporales.



## Ejemplos de uso

La destructuración se puede usar en una variedad de contextos, entre ellos:

- Para acceder a los valores de los parámetros de una función.
- Para asignar los valores de los resultados de una función a variables individuales.
- Para modificar los valores de una estructura de datos.
- Para crear nuevas estructuras de datos a partir de otras estructuras de datos.

## 4. Clonar objetos o elementos

Clonar un objeto en JavaScript se refiere a la creación de una copia independiente de un objeto existente. Esta copia tiene la misma estructura y valores que el objeto original, pero son dos entidades distintas en la memoria, lo que significa que cualquier modificación realizada en uno de ellos no afectará al otro.

\* \* \* \* \*





En JavaScript, hay dos formas principales de clonar objetos o elementos:

- **Uso de la función `Object.assign()`**: Esta función copia los valores de los objetos o elementos de origen a los objetos o elementos de destino.
- **Uso del operador `...`**: Este operador también se puede usar para copiar los valores de los objetos o elementos de origen a los objetos o elementos de destino.

\* \* \* \* \*



## 4.1 Object.assign()

### Definición

- La función `Object.assign()` toma dos o más objetos o elementos como parámetros y copia los valores de los objetos o elementos de origen a los objetos o elementos de destino.
- `Object.assign()` es un método que se utiliza para copiar las propiedades enumerables de uno o más objetos fuente hacia un objeto destino.

# Sintaxis

```
Object.assign(destino, fuente1, fuente2, ...)
```

- **destino**: el objeto que recibirá las propiedades copiadas.
- **fuentes1**, **fuentes2**: los objetos fuente de los cuales se copian las propiedades.

\* \* \* \* \*

Por ejemplo, el siguiente código clona el objeto `person`:

```
const person = {  
  name: "John Doe",  
  age: 30,  
  address: {  
    street: "Main Street",  
    city: "New York",  
    state: "NY"  
  }  
};  
  
const clonedPerson = Object.assign({}, person);  
  
console.log(clonedPerson);
```



El anterior código producirá el siguiente resultado:

```
{  
  name: "John Doe",  
  age: 30,  
  address: {  
    street: "Main Street",  
    city: "New York",  
    state: "NY"  
  }  
}
```

\* \* \* \* \*



## 4.2 \*\*Uso del operador

El operador  también se puede usar para copiar los valores de los objetos o elementos de origen a los objetos o elementos de destino.

\* \* \* \* \*



Por ejemplo, el siguiente código clona el objeto `person`:

```
const person = {  
  name: "John Doe",  
  age: 30,  
  address: {  
    street: "Main Street",  
    city: "New York",  
    state: "NY"  
  }  
};  
  
const clonedPerson = { ...person };  
  
console.log(clonedPerson);
```

Este código producirá el mismo resultado que el código anterior.



## Ventajas y desventajas de cada método

El uso de la función `Object.assign()` tiene la ventaja de que es más explícito. El código es más fácil de entender y de mantener.

El uso del operador `...` tiene la ventaja de que es más conciso. El código es más corto y fácil de escribir.

En general, la mejor opción para clonar objetos o elementos depende de las necesidades específicas de la aplicación.

\* \* \* \* \*

# Más Ejemplos de assign()

## Copiar propiedades de un objeto a otro

```
const obj1 = {a: 1, b: 2};  
const obj2 = {};
```

```
Object.assign(obj2, obj1);  
// obj2 es {a: 1, b: 2}
```

\* \* \* \* \*

# Más Ejemplos de assign()

## Fusionar objetos

```
const o1 = {a: 1};  
const o2 = {b: 2};  
const o3 = {c: 3};  
  
const obj = Object.assign({}, o1, o2, o3);  
// obj es {a: 1, b: 2, c: 3}
```

\* \* \* \* \*



# Más Ejemplos de assign()

## Sólo se copian propiedades enumerables

```
const obj1 = Object.defineProperty({}, 'x', {enumerable: false});  
const obj2 = {};  
Object.assign(obj2, obj1);  
// obj2 está vacío
```

\* \* \* \* \*

# Cuidado con tipos por referencia

`Object.assign()` copia por referencia, hay que tener cuidado con objetos y arreglos.

El método `Object.assign()` copia las propiedades de los objetos fuente al objeto destino de forma superficial, es decir, copia por referencia en lugar de crear una copia real profunda.

\* \* \* \* \*

Esto solo es problema cuando en las fuentes hay referencias a objetos o arreglos, veamos un ejemplo:

```
const objFuente = {  
  prop1: 'valor1',  
  prop2: ['a', 'b']  
}  
  
const objDestino = Object.assign({}, objFuente);  
  
objFuente.prop2.push('c');  
  
console.log(objDestino);  
// { prop1: 'valor1', prop2: ['a', 'b', 'c'] }
```



Como ves, al modificar el arreglo `prop2` del objeto fuente, este cambio se refleja también en `objDestino`. Esto es porque la propiedad `prop2` hace referencia al mismo espacio en memoria en ambos objetos.

Esto podría llevar a efectos inesperados y bugs en nuestro código si no lo tenemos en cuenta.

\* \* \* \* \*

**La solución** es hacer una **copia profunda manualmente** de esas propiedades con estructuras anidadas, por ejemplo usando el **operador spread**:

```
const objDestino = {  
  prop1: objFuente.prop1,  
  prop2: [ ... objFuente.prop2]  
}
```

De esta forma sí se crea una copia independiente del arreglo.