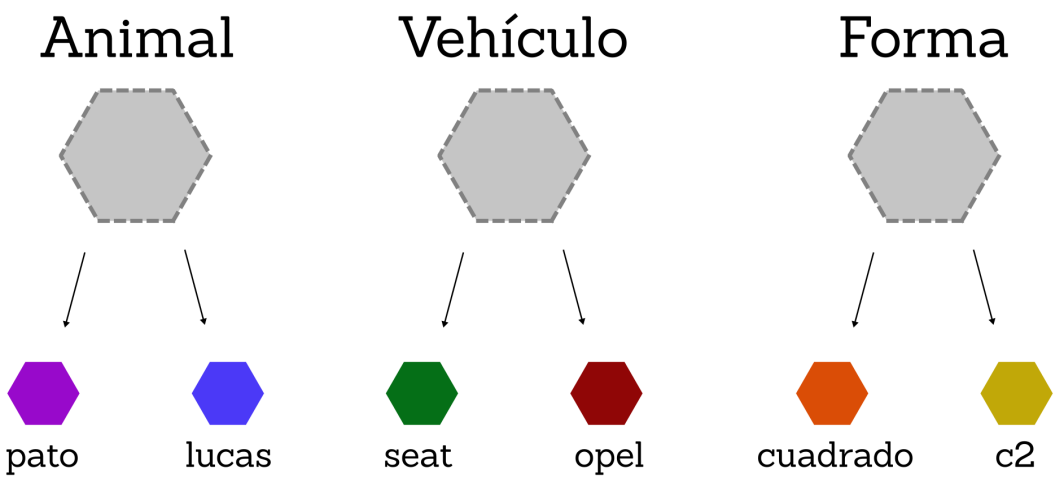
¿Qué es una clase? **Fuente: lenguajejs.com**

Una **clase** es una forma de organizar código de forma entendible con el objetivo de simplificar el funcionamiento de nuestro programa. Además, hay que tener en cuenta que las clases son «conceptos abstractos» de los que se pueden crear objetos de programación, cada uno con sus características concretas.

Esto puede ser complicado de entender con palabras, pero se ve muy claro con ejemplos:



En primer lugar, tenemos la **clase**. La clase es el **concepto abstracto** de un objeto, mientras que el **objeto** es el elemento final que se basa en la clase. En la imagen anterior tenemos varios ejemplos:

- En el **primer ejemplo** tenemos dos variables: **pato** y **lucas**. Ambos son animales, por lo que son objetos que están basados en la clase **Animal**. Tanto **pato** como **lucas** tienen las características que estarán definidas en la clase **Animal**: color, sonido que emiten, nombre, etc...
- En el **segundo ejemplo** tenemos dos variables **seat** y **opel**. Se trata de dos coches, que son vehículos, puesto que están basados en la clase **Vehículo**. Cada uno tendrá las características de su clase: color del vehículo, número de ruedas, marca, modelo, etc...
- En el **tercer ejemplo** tenemos dos variables **cuadrado** y **c2**. Se trata de dos formas geométricas, que al igual que los ejemplos anteriores tendrán sus propias

características, como por ejemplo el tamaño de sus lados. El elemento **cuadrado** puede tener un lado de **3** cm y el elemento **c2** puede tener un lado de **6** cm.

En JavaScript se utiliza una sintaxis muy similar a otros lenguajes como, por ejemplo, Java. Declarar una clase es tan sencillo como escribir lo siguiente:

```
// Declaración de una clase
class Animal {}

// Crear o instanciar un objeto
const pato = new Animal();
```

El nombre elegido debería hacer referencia a la información que va a contener dicha clase. Piensa que el objetivo de las clases es almacenar en ella todo lo que tenga relación (*en este ejemplo, con los animales*).

Observa que luego creamos una variable donde hacemos un **new Animal()**. Estamos creando una variable **pato** (*un objeto*) que es de tipo **Animal**, y que contendrá todas las características definidas dentro de la clase **Animal** (*de momento, vacía*).

> Una norma de estilo en el mundo de la programación es que las **clases** deben siempre **empezar en mayúsculas**. Esto nos ayudará a diferenciarlas sólo con leerlas. Si te interesa este tema, puedes echar un vistazo al tema de las convenciones de nombres en programación.

Elementos de una clase

Una clase tiene diferentes características que la forman, vamos a ir explicándolas todas detalladamente. Pero primero, una tabla general para verlas en conjunto:

Elemento	Descripción
Propiedad	Variable que existe dentro de una clase. Puede ser pública o privada.
Propiedad pública	Propiedad a la que se puede acceder desde fuera de la clase.
Propiedad computada	Función para acceder a una propiedad con modificaciones (getter/setter).
Método	Función que existe dentro de una clase. Puede ser pública o privada.
Método público	Método que se puede ejecutar desde dentro y fuera de la clase.
Método estático	Método que se ejecuta directamente desde la clase, no desde la instancia.
Constructor	Método que se ejecuta automáticamente cuando se crea una instancia.

Como vemos, todas estas características se dividen en dos grupos: las **propiedades** (*a grandes rasgos, variables dentro de clases*) y los **métodos** (*a grandes rasgos, funciones dentro de clases*). Veamos cada una de ellas en detalle, pero empecemos por los **métodos**.

¿Qué es un método?

Hasta ahora habíamos visto que los **métodos** eran funciones que viven dentro de una variable, más concretamente de un objeto. Los objetos de tipo String tienen varios métodos, los objetos de tipo Number tienen otros métodos, etc... Justo eso es lo que definimos en el interior de una clase.

Si añadimos un método a la clase **Animal**, al crear cualquier variable haciendo un **new Animal()**, tendrá automáticamente ese método disponible. Ten en cuenta que podemos crear varias variables de tipo **Animal** y serán totalmente independientes cada una:

```
// Declaración de clase
class Animal {
  // Métodos
  hablar() {
    return "Cuak";
  }
}

// Creación de una instancia u objeto
const pato = new Animal();
pato.hablar(); // 'Cuak'

const donald = new Animal();
donald.hablar(); // 'Cuak'
```

Observa que el método **hablar()**, que se encuentra dentro de la clase **Animal**, existe en las variables **pato** y **donald** porque realmente son de tipo **Animal**. Al igual que con las funciones, se le pueden pasar varios parámetros al método y trabajar con ellos como venimos haciendo normalmente con las funciones.

¿Qué es un método estático?

En el caso anterior, para usar un método de una clase, como por ejemplo **hablar()**, debemos crear el objeto basado en la clase haciendo un **new** de la clase. Lo que se denomina crear un objeto o una instancia de la clase. En algunos casos, nos puede interesar crear **métodos estáticos** en una clase porque para utilizarlos no hace falta crear ese objeto, sino que se pueden ejecutar directamente sobre la clase directamente:

```
class Animal {
  static despedirse() {
    return "Adiós";
  }

  hablar() {
    return "Cuak";
  }
}

Animal.despedirse(); // 'Adiós'
```

Como veremos más adelante, lo habitual suele ser utilizar métodos normales (*no estáticos*), porque normalmente nos suele interesar crear varios objetos y guardar información diferente en cada uno de ellos, y para eso tendríamos que instanciar un objeto.

Una de las limitaciones de los **métodos estáticos** es que en su interior sólo podremos hacer referencia a elementos que también sean estáticos. No podremos acceder a propiedades o métodos no estáticos, ya que necesitaríamos instanciar un objeto para hacerlo.

> Los **métodos estáticos** se suelen utilizar para crear funciones de apoyo que realicen tareas concretas o genéricas, porque están relacionadas con la clase en general.

¿Qué es un constructor?

Se le llama **constructor** a un tipo especial de método de una clase, que se ejecuta automáticamente a la hora de hacer un **new** de dicha clase. Una clase **solo puede tener**

un constructor, y en el caso de que no se especifique un constructor a una clase, tendrá uno vacío de forma implícita.

```
// Declaración de clase
class Animal {
  // Método que se ejecuta al hacer un new
  constructor() {
    console.warn("Ha nacido un pato.");
  }
  // Métodos
  hablar() {
    return "Cuak";
  }
}

// Creación de una instancia u objeto
const pato = new Animal(); // 'Ha nacido un pato'
```

El **constructor** es un mecanismo muy interesante y utilizado para tareas de inicialización o que quieres realizar tras haber creado el nuevo objeto.

¿Qué es una propiedad?

Las clases, siendo estructuras para guardar información, pueden guardar variables con su correspondiente información. Dicho concepto se denomina **propiedades** y en Javascript se realiza en el interior del constructor, precedido de la palabra clave **this** (*que hace referencia a «este» elemento, es decir, la clase*), como puedes ver en el siguiente ejemplo:

```
class Animal {
  constructor(n = "pato") {
    this.nombre = n;
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}
```

```
// Creación de objetos
const pato = new Animal();
pato.quienSoy(); // 'Hola, soy pato'

const donald = new Animal("Donald");
donald.quienSoy(); // 'Hola, soy Donald'
```

Como se puede ver, estas **propiedades** existen en la clase, y se puede establecer de forma que todos los objetos tengan el mismo valor, o como en el ejemplo anterior, tengan valores diferentes dependiendo del objeto en cuestión, pasándole los valores específicos por parámetro.

Observa que, las propiedades de la clase podrán ser modificadas externamente, ya que por defecto son **propiedades públicas**:

```
const pato = new Animal("Donald");
pato.quienSoy(); // 'Hola, soy Donald'

pato.nombre = "Paco";
pato.quienSoy(); // 'Hola, soy Paco'
```

Los ámbitos en una clase

Dentro de una clase tenemos dos tipos de ámbitos: **ámbito de método** y **ámbito de clase**:

En primer lugar, veamos el **ámbito dentro de un método**. Si declaramos variables o funciones dentro de un método con **var**, **let** o **const**, estos elementos existirán sólo en el método en cuestión. Además, no serán accesibles desde fuera del método:

```
class Clase {
  constructor() {
    const name = "Manz";
    console.log("Constructor: " + name);
  }

  metodo() {
    console.log("Método: " + name);
  }
}
```

```
const c = new Clase(); // 'Constructor: Manz'

c.name; // undefined
c.metodo(); // 'Método: '
```

Observa que la variable **name** solo se muestra cuando se hace referencia a ella dentro del **constructor()** que es donde se creó y donde existe.

En segundo lugar, tenemos el **ámbito de clase**. Podemos crear propiedades precedidas por **this.** (*desde dentro del constructor*) y desde **ES2020** desde la parte superior de la clase, lo que significa que estas propiedades tendrán alcance en toda la clase, tanto desde el constructor, como desde otros métodos del mismo:

```
class Clase {
  role = "Teacher”; // ES2020+

  constructor() {
    this.name = "Manz";
    console.log("Constructor: " + this.name);
  }

  metodo() {
    console.log("Método: " + this.name);
  }
}

const c = new Clase(); // 'Constructor: Manz'

c.name;     // 'Manz'
c.metodo(); // 'Método: Manz'
c.role;     // 'Teacher'
```

Ojo, estas propiedades también pueden ser modificadas desde fuera de la clase, simplemente asignándole otro valor. Si quieres evitarlo, añade el **#** antes del nombre de la propiedad al declararla.

La palabra clave this

Como te habrás fijado en ejemplos anteriores, hemos introducido la palabra clave **this**, que hace referencia al **elemento padre** que la contiene. Así pues, si escribimos **this.nombre** dentro de un método, estaremos haciendo referencia a la propiedad **nombre** que existe

dentro de ese objeto. De la misma forma, si escribimos **this.hablar()** estaremos ejecutando el método **hablar()** de ese objeto.

Veamos el siguiente ejemplo, volviendo al símil de los animales:

```
class Animal {
  constructor(n = "pato") {
    this.nombre = n;
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre + ". ~" + this.hablar();
  }
}

const pato = new Animal("Donald");

pato.quienSoy(); // 'Hola, soy Donald. ~Cuak'
```

Ten en cuenta que si usas **this** en contextos concretos, como por ejemplo fuera de una clase te devolverá el objeto **Window**, que no es más que una referencia al objeto global de la pestaña actual donde nos encontramos y tenemos cargada la página web.

> Es importante tener mucho cuidado con la palabra clave **this**, ya que en muchas situaciones creeremos que devolverá una referencia al elemento padre que la contiene, pero devolverá el objeto **Window** porque se encuentra fuera de una clase o dentro de una función con otro contexto. Asegúrate siempre de que **this** tiene el valor que realmente crees que tiene.

¿Qué es un getter?

Los **getters** son la forma de definir propiedades computadas de lectura en una clase. Veamos un ejemplo sobre el ejemplo anterior de la clase **Animal**:

```
class Animal {
  constructor(n) {
    this._nombre = n;
  }

  get nombre() {
    return "Sr. " + this._nombre;
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}

// Creación de objetos
const pato = new Animal("Donald");

pato.nombre; // 'Sr. Donald'
pato.nombre = "Pancracio"; // 'Pancracio'
pato.nombre; // 'Sr. Donald'
```

Si observas los resultados de este último ejemplo, puedes comprobar que la diferencia al utilizar **getters** es que las propiedades con **get** no se pueden cambiar, son de sólo lectura.

¿Qué es un setter?

De la misma forma que tenemos un **getter** para obtener información mediante **propiedades computadas**, también podemos tener un **setter**, que es el mismo concepto, pero en lugar de obtener información, para establecer información.

Si incluímos un **getter** y un **setter** a una propiedad en una clase, podremos modificarla directamente:

```
class Animal {
  constructor(n) {
    this.nombre = n;
  }
```

```
  get nombre() {
    return "Sr. " + this._nombre;
  }

  set nombre(n) {
    this._nombre = n.trim();
  }

  hablar() {
    return "Cuak";
  }
  quienSoy() {
    return "Hola, soy " + this.nombre;
  }
}

// Creación de objetos
const pato = new Animal("Donald");

pato.nombre; // 'Sr. Donald'
pato.nombre = "   Lucas  "; // '   Lucas  '
pato.nombre; // 'Sr. Lucas'
```

Observa que de la misma forma que con los **getters**, podemos realizar tareas sobre los parámetros del **setter** antes de guardarlos en la propiedad interna. Esto nos servirá para hacer modificaciones previas, como por ejemplo, en el ejemplo anterior, realizando un **trim()** para limpiar posibles espacios antes de guardar esa información.

Iteración sobre el objeto con for..in

El siguiente bucle `for...in` itera sobre todas las propiedades enumerables que no son símbolos del objeto y registra una cadena de los nombres de propiedad y sus valores.

```
let obj = {a: 1, b: 2, c: 3};

for (let prop in obj) {

console.log(`obj.${prop} = ${obj[prop]}`);

}

// Produce: // "obj.a = 1" // "obj.b = 2" // "obj.c = 3"
```

For…of // For…in

La sentencia **for...of** ejecuta un bloque de código para cada elemento de un objeto iterable, como lo son: String, Array, entre otros.

Sintaxis

```
for (variable of iterable) {
  statement
}
```

variable
En cada iteración el elemento (propiedad enumerable) correspondiente es asignado a variable.
iterable
Objeto cuyas propiedades enumerables son iteradas.

Ejemplos
Iterando un Array

```
let iterable = [10, 20, 30];

for (let value of iterable) {
  value += 1;
  console.log(value);
}
// 11
// 21
// 31
```

Es posible usar const en lugar de let si no se va a modificar la variable dentro del bloque.

```
let iterable = [10, 20, 30];

for (const value of iterable) {
  console.log(value);
}
// 10
// 20
// 30
```

Iterando un String

```
let iterable = "boo";

for (let value of iterable) {
  console.log(value);
}
// "b"
// "o"
```

```
// "o"
```

Diferencia entre for...of y for...in

El bucle **for...in** iterará sobre todas las propiedades de un objeto. La sintaxis de **for...of** es específica para las colecciones, y no para todos los objetos.

El siguiente ejemplo muestra las diferencias entre un bucle for...of y un bucle for...in en arrays.

```
let arr = [3, 5, 7];
arr.foo = "hola";

for (let i in arr) {
   console.log(i); // logs "0", "1", "2", "foo"
}

for (let i of arr) {
   console.log(i); // logs "3", "5", "7"
}
```

localStorage y sessionStorage: ¿qué son?

El objeto Storage (API de almacenamiento web) nos permite almacenar datos de manera local en el navegador y sin necesidad de realizar alguna conexión a una base de datos.

localStorage y **sessionStorage** son propiedades que acceden al objeto Storage y tienen la función de almacenar datos de manera local, la diferencia entre éstas dos es que localStorage almacena la información de forma indefinida o hasta que se decida limpiar los datos del navegador y sessionStorage almacena información mientras la pestaña donde se esté utilizando siga abierta, una vez cerrada, la información se elimina.

Validar objeto Storage en el navegador

Aunque gran parte de los navegadores hoy en día son compatibles con el objeto Storage, no está de más hacer una pequeña validación para rectificar que realmente podemos utilizar dicho objeto, para ello utilizaremos el siguiente código:

```
if (typeof(Storage) !== 'undefined') {
  // Código cuando Storage es compatible
} else {
 // Código cuando Storage NO es compatible
}
```

Guardar datos en Storage

Existen dos formas de guardar datos en Storage, que son las siguientes:

localStorage

```
// Opción 1 ->  localStorage.setItem(name, content)
// Opción 2 ->localStorage.name = content
// name = nombre del elemento
// content = Contenido del elemento

localStorage.setItem('Nombre', 'Miguel Antonio')
localStorage.Apellido = 'Márquez Montoya'
```

sessionStorage

```
// Opción 1 ->  sessionStorage.setItem(name, content)
// Opción 2 ->sessionStorage.name = content
// name = nombre del elemento
// content = Contenido del elemento

sessionStorage.setItem('Nombre', 'Miguel Antonio')
sessionStorage.Apellido = 'Márquez Montoya'
```

Recuperar datos de Storage

Al igual que para agregar información, para recuperarla tenemos dos maneras de hacerlo

localStorage

```
// Opción 1 -> localStorage.getItem(name, content)
// Opción 2 -> localStorage.name

// Obtenemos los datos y los almacenamos en variables
let firstName = localStorage.getItem('Nombre'),
    lastName  = localStorage.Apellido

console.log(`Hola, mi nombre es ${firstName} ${lastName}`)
// Imprime: Hola, mi nombre es Miguel Antonio Márquez Montoya
```

sessionStorage

```
// Opción 1 -> sessionStorage.getItem(name, content)
// Opción 2 -> sessionStorage.name

// Obtenemos los datos y los almacenamos en variables
let firstName = sessionStorage.getItem('Nombre'),
    lastName  = sessionStorage.Apellido

console.log(`Hola, mi nombre es ${firstName} ${lastName}`)
// Imprime: Hola, mi nombre es Miguel Antonio Márquez Montoya
```

Eliminar datos de Storage

Para eliminar un elemento dentro de Storage haremos lo siguiente:

localStorage

```
// localStorage.removeItem(name)
localStorage.removeItem(Apellido)
```

sessionStorage

```
// sessionStorage.removeItem(name)
sessionStorage.removeItem(Apellido)
```

Limpiar todo el Storage

Ya para finalizar veremos la forma para eliminar todos los datos del Storage y dejarlo completamente limpio

localStorage

localStorage.clear()

sessionStorage

sessionStorage.clear()

Guardar y mostrar múltiples datos desde local/session Storage

La clave está en que **localStorage solo nos permite guardar un *string***. Así que primero debemos convertir nuestro objeto a string con *JSON.stringify()* como en el siguiente ejemplo:

```
let datos = ['html5','css3','js'];
// Guardo el objeto como un string
localStorage.setItem('datos', JSON.stringify(datos));
```

```
// Obtener el string previamente salvado
let datos = localStorage.getItem('datos');
console.log('Datos: ', JSON.parse(datos));
```

Las características de *Local Storage* y *Session Storage* son:

- Permiten almacenar entre 5MB y 10MB de información; incluyendo texto y multimedia
- La información está almacenada en la computadora del cliente y NO es enviada en cada petición del servidor, a diferencia de las cookies
- Utilizan un número mínimo de peticiones al servidor para reducir el tráfico de la red
- Previenen pérdidas de información cuando se desconecta de la red
- La información es guardada por domino web (incluye todas las páginas del dominio)