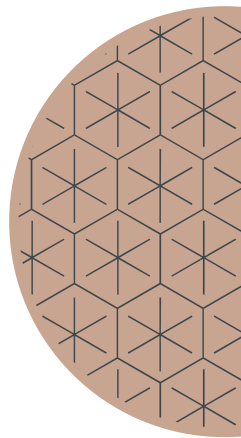


**Programación con  
estructuras  
definidas por el  
usuario.**

# Modelo de objetos predefinidos en JavaScript

En esta unidad veremos los siguientes conceptos:

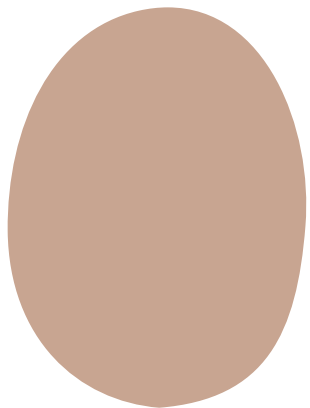
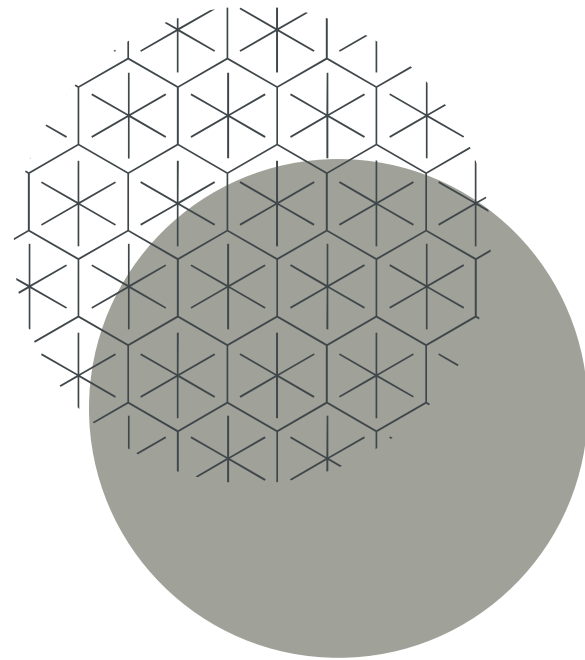
- Objetos en Javascript
- Uso de JSON





# Objetos

---



# Objetos

Si has aprendido otros lenguajes de programación, entonces tendrás conocimientos sobre los objetos. Sin embargo, los objetos en JavaScript son diferentes, y son uno de los tipos de datos más fundamentales junto con los arrays.

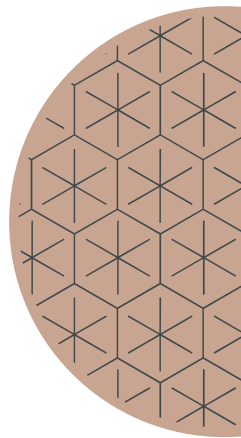
En JavaScript, cada objeto consiste en pares nombre:valor llamados propiedades. Los objetos son tipos de datos no primitivos.

# Objetos

Ejemplo de objeto:

```
let user = {  
  userName: 'Rick',  
  age: 33  
};
```

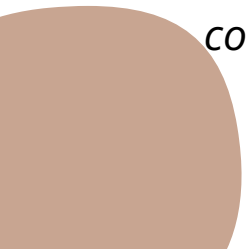
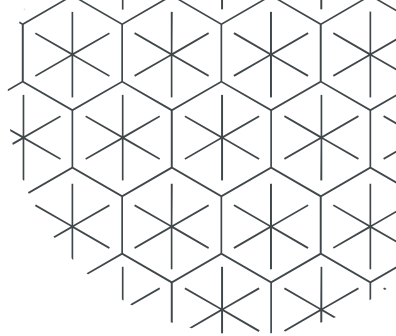
En el ejemplo anterior, en la variable "usuario" almacenamos un objeto con dos propiedades. Así es como creamos un objeto JavaScript con un literal de objeto.



# Objetos

```
//Objects
let user = {
  userName: 'Rick',
  age: 33
};
//para acceder a los datos usamos la notación punto y los guardamos en variables
let userName = user.userName;
let userAge = user.age;

//mostramos sus valores
console.log(userName);
console.log(userAge);
```



# Crear objetos con new

Hay otra forma de crear un objeto utilizando la palabra clave "new". Este método requerirá más escritura, pero al final, el resultado será el mismo

# Crear objetos con new



```
> //Create an object using the new keyword
const newUser = new Object();
newUser.name = 'Rick Sekuloski';
newUser.student = false;
newUser.occupation = 'Web Developer & Programmer';
newUser.students = 20000;
console.log(newUser);
```

▼ {name: "Rick Sekuloski", student: false, occupation: "Web Developer & Programmer", students: 20000} ⓘ

- name: "Rick Sekuloski"
- occupation: "Web Developer & Programmer"
- student: false
- students: 20000

▶ [[Prototype]]: Object

undefined



# Crear objetos con new

La primera forma de declarar un objeto como literal es más utilizada entre los desarrolladores. También podemos tener un objeto vacío, un objeto sin propiedades name:value o pares key:value, y esto es una práctica de codificación aceptable.

```
//empty obj  
let studentObj = {};
```

# Crear objetos con new



```

> //car object
  let carObj1 = {
    name: 'Ford',
    model: 'Mustang',
    color: 'blue-white',
  }
  let carObj2 = carObj1;
  carObj2.model = 'Fiesta';
  console.log('Car Object1', carObj1);
  console.log('Car Object1', carObj2);

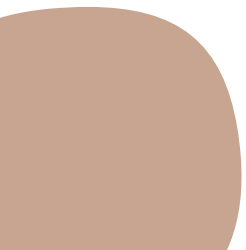
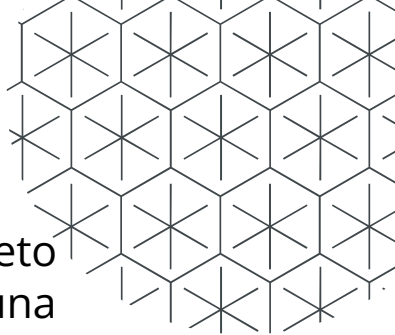
Car Object1 ▶ {name: "Ford", model: "Fiesta", color: "blue-white"}
Car Object1 ▶ {name: "Ford", model: "Fiesta", color: "blue-white"}
< undefined

```

# Crear objetos con new

Aquí tenemos dos variables que apuntan al mismo objeto almacenado en algún lugar de la memoria y que sólo mantienen una referencia. Por lo tanto, cambiar el modelo del coche con el segundo objeto también afecta al primer objeto.

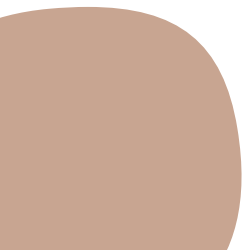
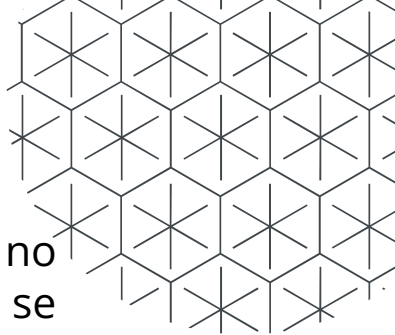
No importa qué objeto estemos usando, todavía estamos cambiando el valor del objeto almacenado en la memoria, y ambos objetos tienen una referencia a ese objeto. Por eso el valor del modelo se cambia de 'Mustang' a 'Fiesta', y cuando imprimimos las dos variables del objeto, obtenemos exactamente el mismo resultado.



# Crear objetos con new

Si recuerdas, anteriormente dije que los objetos o valores no primitivos se pasan por referencia, y los tipos de datos primitivos se pasan por valor.

Si te has fijado, el objeto tiene una coma al final de la última propiedad. Esta coma final es muy útil cuando añadimos una propiedad adicional en nuestro objeto, de esta manera siempre estamos seguros de que podemos añadir una nueva propiedad sin causar un error de sintaxis.

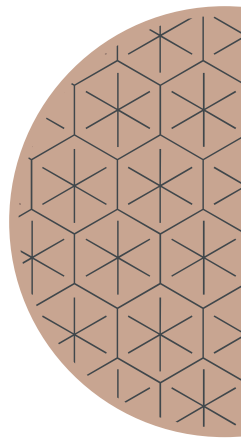


# Creación de objetos con `Object.create()`

El método `Object.create()` crea un nuevo objeto utilizando su primer argumento como el prototipo del objeto recién creado.

# Crear objetos con new

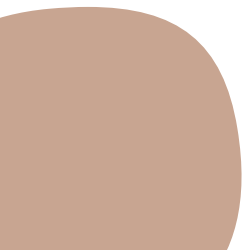
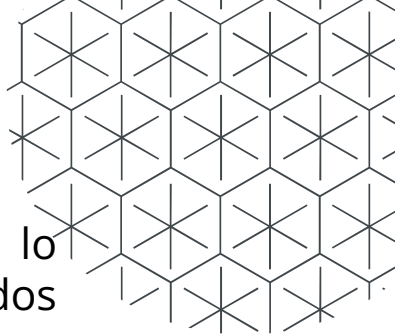
```
const person = {  
  isHuman: false,  
  myDetails: function() {  
    console.log(My full name is ${this.name} ${this.lastName}. Am I human?  
    ${this.isHuman});  
  }  
};  
const rick = Object.create(person);  
rick.name = 'Rick'; rick.lastName = 'Sekuloski';  
rick.isHuman = true;  
rick.myDetails();
```



# Crear objetos con new

También se puede crear un nuevo objeto que no herede nada, lo cual es malo porque aún queremos utilizar algunos de los métodos básicos que ofrece `Object.prototype`.

```
const ob4 = Object.create(Object.prototype);
```



# Primitivos pasados como valor

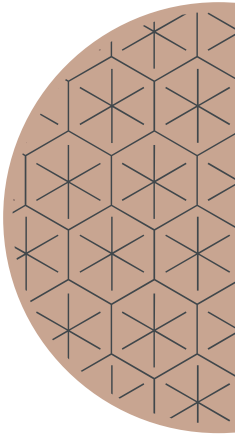
Cuando tenemos un tipo primitivo asignado a una variable, entonces esa variable contiene el valor primitivo real, no la referencia a la memoria sino la ubicación exacta a la memoria.

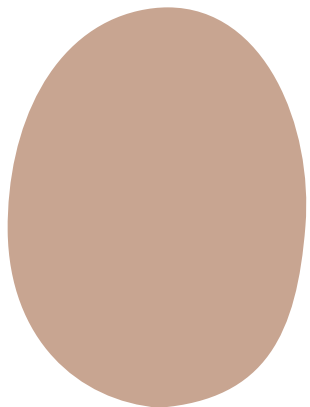
Esto es totalmente opuesto a los objetos, ya que las variables de los objetos sólo contienen una referencia al lugar donde se almacena el objeto en la memoria, pero no el valor real.



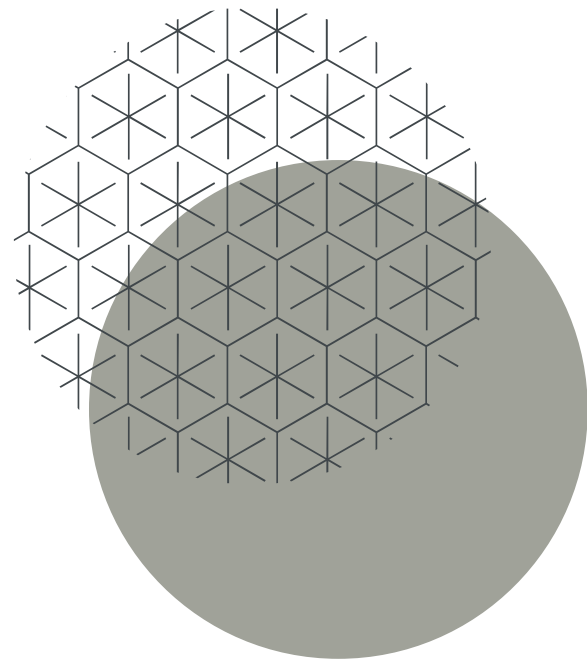
# Crear objetos con new

```
let a = 5;  
let b = 'name';  
let c = null;  
c = a;  
a = 10;  
console.log('Value of a:',a);  
console.log('Value of c:',c);
```



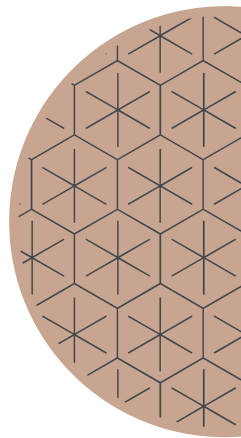


**POO**



# POO - Clases

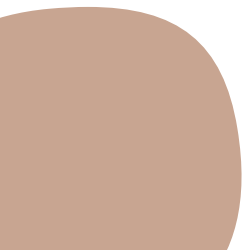
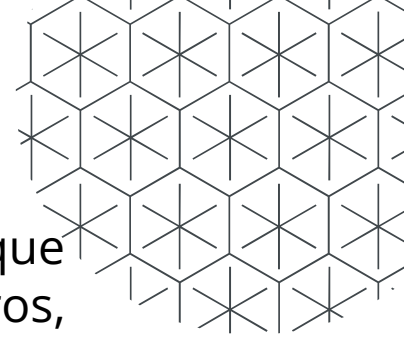
En esta sección, veremos en detalle por qué llamamos a JavaScript un lenguaje orientado a objetos. Pero, ¿qué es la programación orientada a objetos? Si has aprendido un lenguaje de programación como Java, entonces sabes que Java es un lenguaje que utiliza Objetos. Además, en Java, puedes crear clases, y puedes instanciar objetos de esas clases.



# POO - Clases

Pero Java no es un lenguaje orientado a objetos puro porque también tiene tipos de datos primitivos como enteros, booleanos, long, etc. pero utiliza las principales características orientadas a objetos como:

- Herencia
- Encapsulación
- Polimorfismo
- Asociación
- Agregación



# POO - Clases

Para algunos desarrolladores, entender JavaScript es más complejo que otros lenguajes. Como ya sabes los objetos de JavaScript, son un poco diferentes en comparación con los objetos que probablemente has visto en otra programación orientada a objetos como C++ o Java.

Es confuso cómo puedes tener métodos o clases o incluso herencia en JavaScript. La sintaxis y la definición de las clases son similares a las de Java, pero lo que sucede bajo es bastante diferente.

# Classes

*//Classes in JavaScript*

*class Person {*

*constructor(firstName, lastName, age, position, country){*

*this.firstName = firstName;*

*this.lastName = lastName;*

*this.age = age;*

*this.position = position;*

*this.country = country;*

*}*

*printDetails() {*

*console.log(Hi! My details are:*

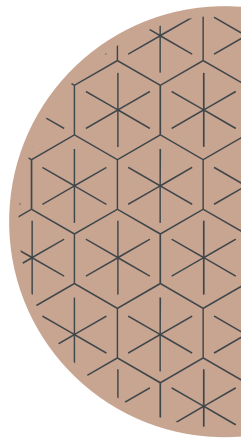
*Name: \${this.firstName},*

*Name: \${this.lastName},*

*);*

*}*

*}*



# Clases

En la clase, tenemos un constructor (), y esta palabra clave declara el cuerpo de la función constructora Person. Las clases de JavaScript no son más que azúcar sintáctico para una función constructora y un objeto prototipo. Una cosa importante a mencionar es cómo podemos instanciar un objeto desde una clase.

```
const samuel = new Person('Samuel', 'L Jackson', 59, 'Actor', 'America');  
samuel.printDetails();
```

# Clases

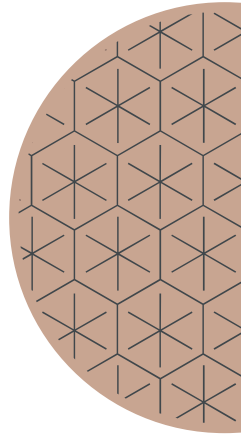
Desde ES6; se recomienda utilizar la sintaxis de las clases en lugar de las clases basadas en prototipos de JavaScript.

Para usarlas tenemos que recordar que hay que utilizar la palabra clave `class` seguida del nombre de la clase, que debe comenzar con una letra mayúscula. La clase tiene su cuerpo, y eso se marca con los corchetes. Dentro de la clase, tenemos la palabra clave `constructor`, que es la misma ya que estamos definiendo la función constructora.



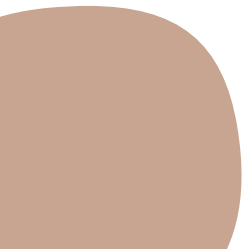
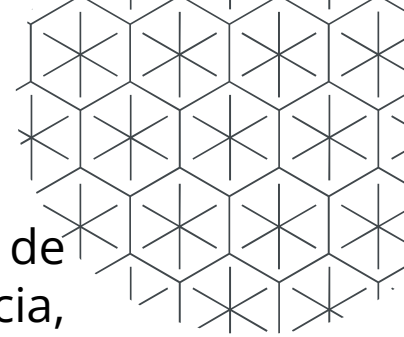
# Clases

En otros ejemplos, encontrarás que no necesitamos definir el constructor en absoluto. Esto significa que no tenemos variables que necesiten ser inicializadas y con valores, pero porque omitamos la palabra clave constructor no significa que no habrá un constructor para esta clase. Una función constructora vacía será creada implícitamente para nosotros en el fondo.



# Herencia

Uno de los temas más candentes en cualquier lenguaje de programación OOP es la herencia. Para hablar de herencia, necesitamos tener una clase padre, y queremos que los hijos de esa clase obtengan toda la funcionalidad y propiedades que la clase padre tiene para ofrecer, pero lo que es más importante es que los hijos de esa clase puedan añadir sus propias funcionalidades y propiedades nuevas.



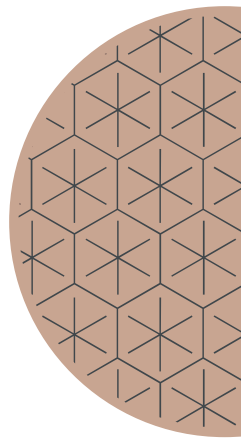
# Herencia

Ahora vamos a ver cómo vamos a lograr la herencia en JavaScript a través de ejemplos.

Digamos que tenemos una clase llamada Animal. Dentro del constructor de la clase, estableceremos e inicializaremos dos propiedades del animal (nombre y velocidad). También tendremos dos métodos que son muy "genéricos", y que todo animal puede hacer.

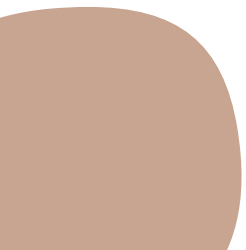
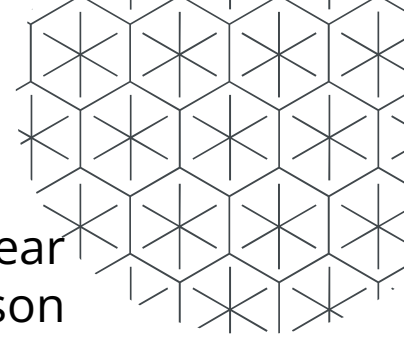
# Herencia

```
class Animal {  
  constructor(name) {  
    this.speed = 0;  
    this.name = name;  
  }  
  startRunning(speed) {  
    this.speed = speed;  
    return The ${this.name} runs with speed of ${this.speed}km per hour.;  
  }  
  stopRunning() {  
    this.speed = 0;  
    console.log(The ${this.name} stopped running and now sits still.);  
  }  
}
```



# Herencia

Después de declarar esta clase básica Animal, podemos crear otra clase, por ejemplo, Perro. Los perros también son animales, y deben estar basados en la clase Animal, lo que significa que pueden acceder a los métodos y propiedades genéricas de los animales, pero también los perros pueden hacer algo que otros animales no pueden. Para heredar propiedades de su padre, esta es la sintaxis que debemos utilizar



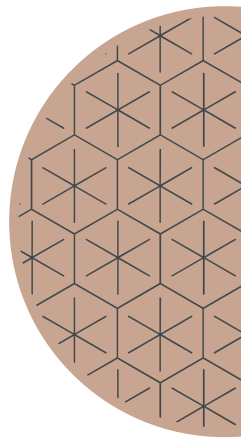
# Herencia

```
class Dog extends Animal {  
  bark() {  
    return The ${this.name} barks!;  
  }  
}  
let dog = new Dog("Dog");  
console.log(dog.startRunning(10));  
console.log(dog.bark()); //The Dog barks!
```

# Herencia

En este ejemplo, el objeto que instanciamos de la clase Dog tiene acceso tanto a los métodos de Dog como a los de Animal, como startRunning y stopRunning.

Bajo el capó, la palabra clave extends funciona utilizando el prototipo, por lo que establece el prototipo Dog.prototype.[[Prototype]] a Animal.prototype. Así, si el objeto solicita un método que no se encuentra en Dog.prototype, empezará a buscar en el de Animal.prototype.



# Propiedades y métodos Static

En JavaScript, también podemos asignar un método a la función de la clase, no a su prototipo. Estos métodos se llaman métodos estáticos. Para declarar un método estático, debemos utilizar la palabra clave `static`. Los métodos estáticos se utilizan para implementar funciones que pertenecen a la clase pero no al objeto.



# Propiedades y métodos Static

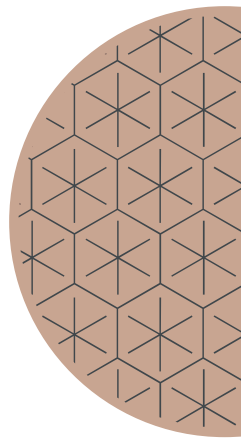
```
class User{  
  constructor(name){  
    this.name = name;  
  }  
  static staticMethod(){  
    console.log(this.name);  
  }  
}
```

```
User.staticMethod();
```

# Overriding métodos

Recuerda de la sección de herencia que cuando las instancias de perro no podían encontrar el método de la clase Dog, empezaban a buscar en la superclase padre para ese método en particular. En nuestro caso, esta era la clase Animal.

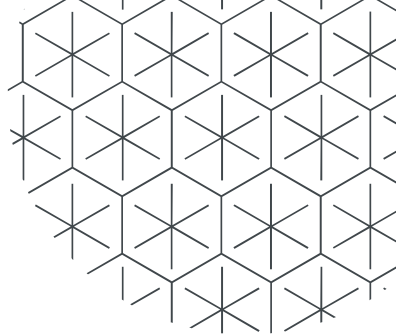
Así que, si añadimos otro método en la clase Dog con el mismo nombre que en la clase Animal, el proceso se llama override de métodos. ¿Por qué? El nombre se explica por sí mismo, significa que tenemos un método con el mismo nombre en la clase padre y en la clase hija.



# Overriding métodos

```
class Dog extends Animal {  
  stopRunning (){//this method will be used by the instances of the Dog class  
    super.stopRunning();  
    console.log('The dog is licking its paws');  
  }  
  bark() {  
    return `The ${this.name} barks!`;  
  }  
}
```

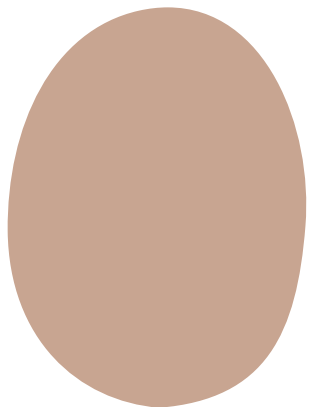
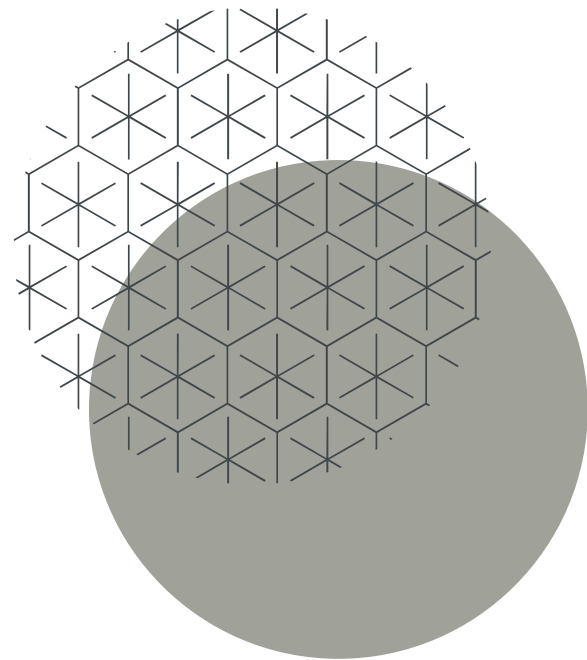
- super (...), podemos usarlo para llamar al constructor padre (necesitamos usar esta palabra clave dentro del constructor hijo)
- super.method (...), podemos usarlo para llamar al método declarado en el padre.





# JSON

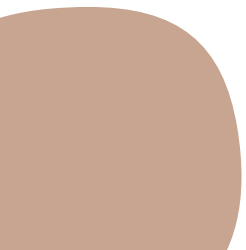
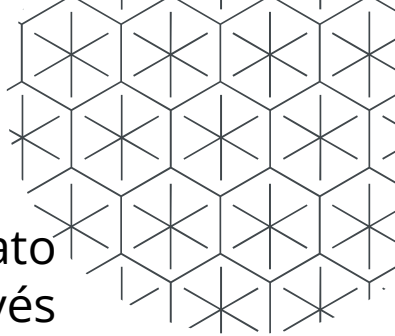
---



# ¿Qué es JSON?

JSON es un formato para codificar datos en un formato legible para el ser humano para almacenar y enviar a través de una red.

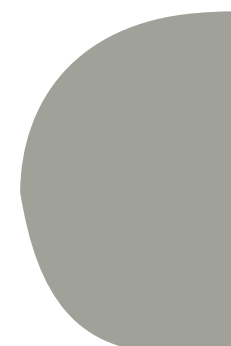

Aunque comenzó en JavaScript, se utiliza en todos los lenguajes de programación modernos.



# ¿Por qué es usado?

El formato JSON es sintácticamente similar al código para crear objetos JavaScript. Por ello, un programa JavaScript puede convertir fácilmente los datos JSON en objetos JavaScript.

Como el formato es sólo de texto, los datos JSON pueden enviarse fácilmente entre ordenadores y ser utilizados por cualquier lenguaje de programación.



# ¿Por qué es usado?

JavaScript tiene una función integrada para convertir cadenas JSON en objetos JavaScript:

*JSON.parse()*

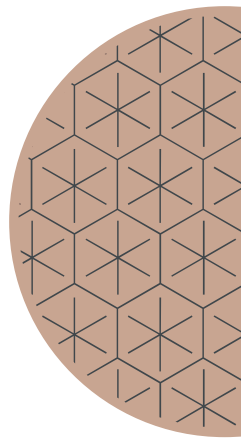
JavaScript también tiene una función integrada para convertir un objeto en una cadena JSON:

*JSON.stringify()*

# Visión general del formato JSON

JSON almacena los datos como:

- Pares clave/valor.
- Los datos se separan con comas.
- Los datos de texto van entre comillas dobles.
- Los datos numéricos no llevan comillas.
- Las arrays van entre corchetes [].
- Los objetos van entre corchetes {}
- La salida es una cadena de texto.





# ¿Por qué es mejor que el XML?

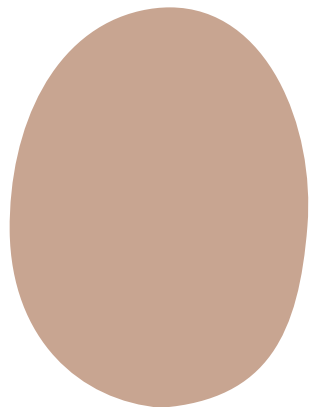
XML es mucho más difícil de parsear que JSON. JSON se parsea directamente en un objeto JavaScript listo para usar.

## Uso de **XML**

- Obtener un documento XML
- Utilizar el XML DOM para recorrer el documento
- Extraer valores y almacenarlos en variables

## Uso de **JSON**

- Obtener una cadena JSON
- JSON.Parse la cadena JSON



**FIN**

