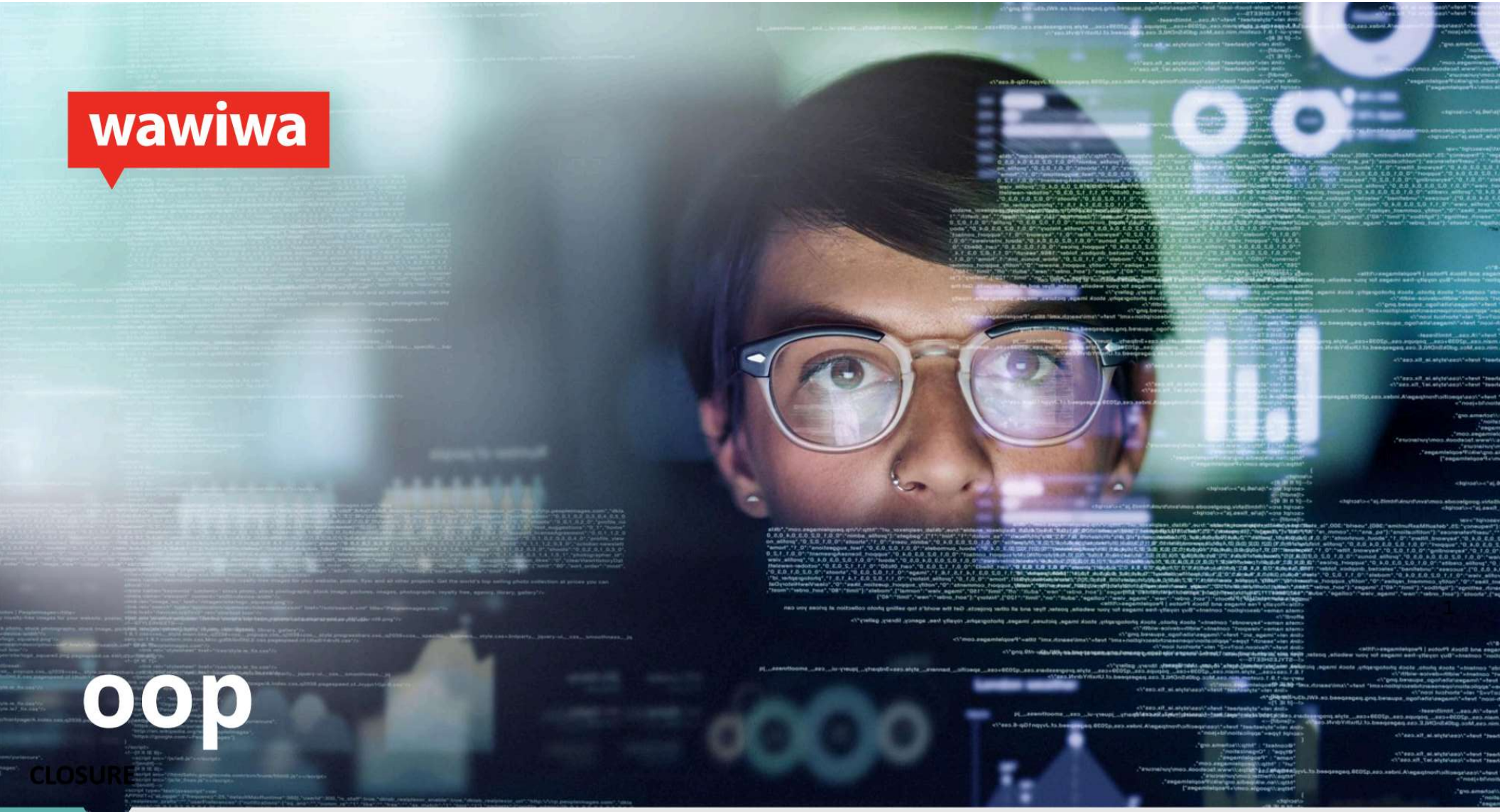


oop

CLOSURE

wawiwa

La vieja manera de definir un objeto

```
// Esto funciona bien, pero es un poco extenso
// si sabemos que queremos crear un objeto
// ¿por qué debemos crear explícitamente
// un nuevo objeto vacío y retornarlo?

function createNewPerson(name) {
    const obj = {};
    obj.name = name;
    obj.greeting = function() {
        alert('Hi! I\'m ' + obj.name + '.');
    };

    return obj;
}

const salva = createNewPerson('Salva');
salva.name;
salva.greeting();
```

wawiwa

La función constructor

```
// Reemplaza tu función previa con lo siguiente:
function Person(name) {
    this.name = name;

    this.greeting = function() {
        alert('Hi! I\'m ' + this.name + '.');
    };
}

const person1 = new Person('Bob');
const person2 = new Person('Sarah');
person1.name;
person1.greeting();
person2.name;
person2.greeting();
```

La función constructor es una versión de una clase de Javascript. Pon atención que tiene todas las funcionalidades que se espera de una función, sin embargo no retorna nada o explicitamente crea un objeto: básicamente solamente define propiedades y métodos También pon atención que esta clave usada aquí también — está diciendo básicamente que cuando una instancia de este objeto sea creada, el nombre del objeto será propiamente igual al valor casado a la llamada del constructor , y el método greeting() también usará el nombre pasado a la llamada del constructor.

Después de que los nuevos objetos hayan sido creados, las variables person1 y person2 contienen los siguientes objetos:

```
{
      name: 'Bob',
      greeting: function() {
            alert('Hi! I\'m ' + this.name + '.');
      }
}

{
      name: 'Sarah',
      greeting: function() {
            alert('Hi! I\'m ' + this.name + '.');
      }
}
```

wawiwa

La función constructor

```
function Person(first, last, age, gender, interests) {
     this.name = { first : first, last : last };
     this.age = age;
     this.gender = gender;
     this.interests = interests;
     this.bio = function () {
          alert(this.name.first + ' ' + this.name.last +
               ' is ' + this.age + ' years old. He likes ' +
               this.interests[0] + ' and ' + this.interests[1] + '.');
     };
     this.greeting = function() {
          alert('Hi! I\'m ' + this.name.first + '.');
     };
}

const person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
person1['age'];
person1.interests[1];
person1.bio();
```

Ahora puedes ver que puedes acceder a las propiedades y métodos como lo hacías antes — prueba esto en tu consola JS:

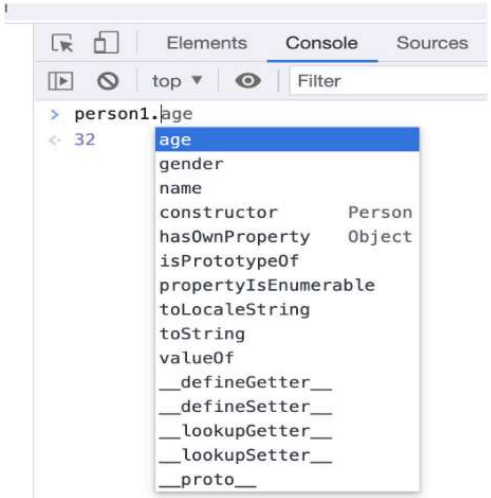
wawiwa

Mecanismo de Prototipos

```
function Person(first, last, age, gender, interests) {
    // definición de propiedades y métodos
    this.name = { 'first': first, 'last' : last };
    this.age = age;
    this.gender = gender;
}

const person1 = new Person('Bob', 'Smith', 32, 'male',
['music', 'skiing']);
// Si escribes "person1." en tu consola Javascript, deberias de
ver que el navegador trata de autocompletar en este objeto:
```

Los prototipos son el mecanismo por el cual los objetos Javascript heredan características de uno a otro.
Explicaremos cómo las cadenas de prototipos funcionan y se ven y como la propiedad prototipo puede ser usada para agregar métodos a constructores existentes.

Para entender objetos prototipo en este ejemplo, hemos definido una función constructor.

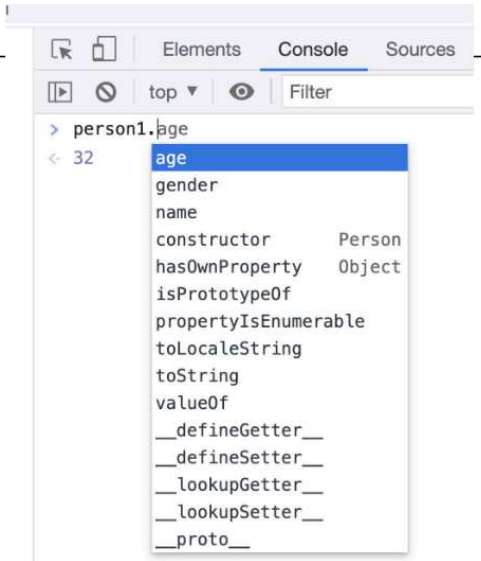


wawiwa

Mecanismo de Prototipos

```
function Person(first, last, age, gender, interests) {
    // definición de propiedades y mecanismos
    this.name = { 'first': first, 'last' : last };
    this.age = age;
    this.gender = gender;
}

const person1 = new Person('Bob', 'Smith', 32, 'male',
['music', 'skiing']);
// Si escribes "person1." en tu consola Javascript, deberias de
ver que el navegador trata de autocompletar en este objeto:
```

En esta lista, verás los miembros definidos en el constructor de person1 — Person() — name, age, gender, interests, bio, y greeting. **Sin embargo, veras algunos otros miembros— toString, valueOf, etc — Estas están definidas en el objeto prototipo del objeto persona1, que es el objeto prototipo del objeto Object.prototype.**



Mecanismos de prototipo

¿Qué pasa si llamas al método en person1, que está definido en Object.prototype? Por ejemplo:
person1.valueOf()

valueOf() retorna el valor del objeto que está siendo llamado. En este caso, lo que sucede es:

- El navegador inicialmente checa si el objeto person1 tiene el método valueOf() disponible, como se define en su constructor, Person(), o no.
- Entonces, el navegador verifica si el objeto prototipo de person1 tiene un método valueOf() disponible en él. No lo tiene, luego el navegador verifica el objeto prototipo del objeto prototipo de person1, y lo tiene. Entonces, se llama al método y todo está bien.

wawiwa

El prototipo propiedad - bueno saberlo

Mira la página de referencia de Object,
Del lado izquierdo hay un gran número de propiedades y métodos — muchas más que cualquier número de miembros heredados. Algunos son heredados y algunos no lo son — ¿por qué pasa esto?

Los que son heredados son los que definidos en la propiedad del prototipo (Podrías llamarlo un "subespacio de nombres") — esto es, los que empiezan con Object.prototype., y no los que empiezan solo con Object.

El valor del prototipo propiedad es un objeto.
Object.prototype.toString(), Object.prototype.valueOf(), etc., están disponibles para cualquier tipo de objeto que heredan de Object.prototype, incluidas las nuevas instancias creadas del constructor Person().

Object.is(), Object.keys(), y otros miembros no son definidos dentro de la cubeta del prototipo, no son heredados por ninguna instancia del objetos o tipos de objetos que heredan de Object.prototype. Son métodos/propiedades disponibles solamente en el constructor Object() mismo.

Herencia en JavaScript

Hasta ahora hemos visto algo de herencia en acción — hemos visto cómo las cadenas de prototipo funcionan, cómo se heredan los miembros al subir por una cadena. Pero en su mayoría, esto ha involucrado funciones integradas en el navegador. ¿Cómo creamos un objeto en JavaScript que hereda de otro objeto?

```
function Person(first, last, age, gender, interests) {
     this.name = { first, last };
     this.age = age;
     this.gender = gender;
     this.interests = interests;
};

// Los métodos son definidos en
// el prototipo del constructor. Por ejemplo:
Person.prototype.greeting = function() {
     alert('Hi! I\'m ' + this.name.first + '.');
};

function Teacher(first, last, age, gender, interests, subject) {
     Person.call(this, first, last, age, gender, interests);
     this.subject = subject;
}
```

La función call().

Esta función te permite llamar una función que está definida en otro lugar, pero en el contexto actual. El primer parámetro especifica el valor de “this” que tu quieras usar cuando se ejecute la función, y los otros parámetros son esos que deben ser pasados a la función cuando se invoque.

Queremos que el constructor Teacher() tome los mismos parámetros que el constructor Person() del que hereda, así que los especificamos todos como parámetros en la invocación de call().

La última línea dentro del constructor define la nueva propiedad subject que los profesores van a tener, que las personas genéricas no tienen.

wawiwa

Herencia del constructor sin parámetros

Ten en cuenta que si el constructor del que estás heredando no toma sus valores de propiedad de parámetros, no necesitas especificarlos como argumentos adicionales en **call()**. Entonces, por ejemplo, si tuvieras algo realmente simple como esto:

```
function Brick() {
    this.width = 10;
    this.height = 20;
}

// Podrías heredar las propiedades de width y height
// haciendo esto (así como siguiendo los pasos que se describen
// a continuación):
function BlueGlassBrick() {
    Brick.call(this);
    this.opacity = 0.5;
    this.color = 'blue';
}
```

Ten en cuenta que solo hemos especificado `this` dentro de call() — no se requieren otros parámetros, ya que no estamos heredando ninguna propiedad del padre que se establezca a través de parámetros.

wawiwa

EcmaScript 2015 clases

ECMAScript 2015 Classes ECMAScript 2015 introduce la sintaxis de clases en JavaScript como una forma de escribir clases reutilizables utilizando una sintaxis más limpia, que se asemeja más a las clases en C++ o Java.
En esta sección convertiremos los ejemplos de Person y Teacher de herencia prototipal a clases, para mostrarte cómo se hace.

```
// Veamos una versión reescrita del ejemplo de Person, en estilo de clase:
class Person {
     constructor(first, last, age, gender, interests) {
          this.name = { first, last };
          this.age = age;
          this.gender = gender;
          this.interests = interests;
     }

     greeting() {
          console.log(`Hi! I'm ${this.name.first}`);
     };

     farewell() {
          console.log(
               `${this.name.first} has left the building. Bye for now!`);
     };
}
```

wawiwa

EcmaScript 2015 clases

La declaración de clase (class) indica que estamos creando una nueva clase.
El método constructor() define la función constructora que representa nuestra clase Person.
greeting() y farewell() son métodos de clase. Cualquier método que desees asociar con la clase se define dentro de ella, después del constructor. En este ejemplo, hemos utilizado plantillas de cadena (template literals) en lugar de concatenación de cadenas para que el código sea más fácil de leer.
Ahora podemos instanciar objetos utilizando el operador `new`, de la misma manera que lo hicimos antes:

```
const han = new Person('Han', 'Solo', 25, 'male', ['Smuggling']);
const leia = new Person('Leia', 'Organa', 19, 'female', ['Government']);

han.greeting(); // Hi! I'm Han
leia.farewell(); // Leia has left the building. Bye for now
```

wawiwa

EcmaScript 2015 clases de herencia, bueno saber

Herencia con sintaxis de clase: Anteriormente creamos una clase para representar a una persona. Tienen una serie de propiedades que son comunes a todas las personas; en esta sección crearemos nuestra clase Teacher especializada, haciendo que herede de Person utilizando la sintaxis moderna de clase. Esto se llama crear una subclase o hacer una subclase.
Para crear una subclase, usamos la palabra clave `extends` para indicar a JavaScript la clase en la que queremos basar nuestra clase.

```
class Teacher extends Person {
    constructor(subject, grade) {
        // Ahora 'this' es inicializado cuando se llama al constructor padre.
        super();
        this.subject = subject;
        this.grade = grade;
    }
```

wawiwa

EcmaScript 2015 clases de herencia

```
class Person {
    constructor(first, last, age, gender, interests) {
        this.name = { first, last };
        this.age = age;
        this.gender = gender;
        this.interests = interests;
    }
}
```

wawiwa

EcmaScript 2015 clases de herencia

Dado que el operador `super()` es en realidad el constructor de la clase padre, pasarle los argumentos necesarios del constructor de la clase padre también inicializará las propiedades de la clase padre en nuestra subclase, heredándolas de esta manera:

```
class Teacher extends Person {
    constructor(first, last, age, gender, interests, subject, grade) {
        super(first, last, age, gender, interests);

        // subject y grade son específicos de Teacher
        this.subject = subject;
        this.grade = grade;
    }
}

const snape = new Teacher('Severus', 'Snape', 58, 'male', ['Potions'], 'Dark arts', 5);
snape.greeting(); // Hi! I'm Severus.
snape.farewell(); // Severus has left the building. Bye for now.
snape.age; // 58
snape.subject; // Dark arts
```

wawiwa

EcmaScript 2015 clases de getters y setters

Puede haber momentos en los que queramos cambiar los valores de las propiedades en las clases que creamos, o momentos en los que no sepamos cuál será el valor final de una propiedad en particular. Usando el ejemplo de Teacher, es posible que no sepamos qué materia enseñará el profesor antes de crearlo, o que su materia cambie entre períodos.

Podemos manejar estas situaciones con getters y setters.

Vamos a mejorar la clase Teacher con getters y setters.
La clase comienza de la misma manera que la vimos la última vez.

Los getters y setters trabajan en pares. Un getter devuelve el valor actual de la variable y su setter correspondiente cambia el valor de la variable al que define.

wawiwa

EcmaScript 2015 clases de getters y setters

```
class Teacher extends Person {
    constructor(first, last, age, gender, interests, subject, grade) {
        super(first, last, age, gender, interests);
        // subject y grade son especificos de Teacher
        this._subject = subject;
        this.grade = grade;
    }

    get subject() {
        return this._subject;
    }
    set subject(newSubject) {
        this._subject = newSubject;
    }
}
```

wawiwa

EcmaScript 2015 clases de getters y setters

Para mostrar el valor actual de la propiedad _subject del objeto snape, podemos usar el método getter snape.subject.

Para asignar un nuevo valor a la propiedad _subject, podemos usar el método setter snape.subject = "nuevo valor".

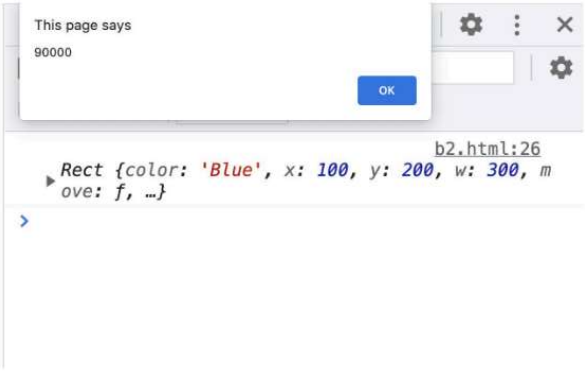
El siguiente ejemplo muestra ambas características en acción:

```
Comprueba el valor por defecto
console.log(snape.subject); // Retorna "Dark arts"

// Cambia el valor
snape.subject = "Balloon animals"; // Configura _subject a "Balloon animals"

// Comprueba otra vez y ve si coincide con el nuevo valor
console.log(snape.subject); // Retorna "Balloon animals"
```

wawiwa

Hazlo tú mismo 48 (Cuaderno de práctica en casa) con ecmascript 2015

1. Crea una clase “Rectangle” que herede de “Shape”. Agrega dos atributos a la clase. El width del rectangle w, y el height de rectangle h.
2. Agrega a la clase una función llamada área que calcule el area del rectangulo area (w * h)
3. Crea una instancia de la clase con color rojo,, x 100, y 200, width 300 y height 300
4. Ejecuta la función de área (area function) y muestra la clase en la consola.




wawiwa
¿Preguntas?