Módulo 4

Antonio Cano Galindo

¿Orientación a Objetos o a Clases?

A javascript podemos llamarlo un lenguaje orientado a objetos desde el momento en todo es un objeto: cualquier entidad creada mediante javascript (dato o función) es considerada un objeto.

Pero, ¿qué es un objeto?. Según la Wikipedia:
"Un objeto es un ente que consta de un estado y de un comportamiento, que a su vez constan respectivamente de datos almacenados y de tareas realizables durante el tiempo de ejecución. Un objeto puede ser creado instanciando una clase, como ocurre en la programación orientada a objetos, o mediante escritura directa de código y la replicación de otros objetos, como ocurre en la programación basada en prototipos.

Características de un objeto: ESTADO, COMPORTAMIENTO E IDENTIDAD.

Características de la POO clásica: ENCAPSULAMENTO, HERENCIA Y POLIMORFISMO.

Desde un punto de vista formal, Javascript es un lenguaje orientado a prototipos.

Diferencias entre lenguajes orientados a objetos

Basado en clases (Java)	Basado en prototipos (JavaScript)
Clase e instancia son dos entidades diferentes	Todos los objetos son instancias
Las clases se definen de manera explícita, y se instancian en objetos a través de su método constructor.	Las clases se definen y crean con las funciones constructoras.
Un objeto se instancia con el operador new.	Un objeto se instancia con el operador new.
La estructura de clases se crea utilizando la definición de clases.	La estructura de clases se crea asignando un objeto como prototipo.
La herencia de propiedades se realiza a través de la cadena de clases.	La herencia de propiedades se realiza a través de la cadena de prototipos.
La definición de clases especifica todas las propiedades de una instancia de una clase. No se pueden añadir propiedades en tiempo de ejecución.	La función constructora o el prototipo especifican unas propiedades iniciales. Se pueden añadir o eliminar estas propiedades en tiempo de ejecución, en un objeto concreto o a un conjunto de objetos.

Antonio Cano Galindo

Orientación a objetos en Javascript

- **Comportamiento:** Se define como "lo que es posible hacer con un objeto". En terminología más purista, el conjunto de mensajes a los que este objeto responde (en OOP se usa la terminología *mensaje* para referirse al paso de información entre objetos).

- **Estado:** Los objetos tienen un estado en todo momento, definido por el conjunto de las variables o campos que contienen. El estado es pues la *información* contenida en un objeto.

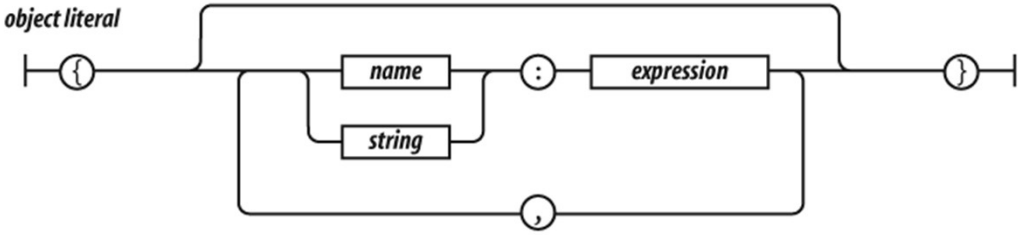
- **Identidad:** Cada objeto existe *independientemente* del resto. Pueden haber dos objetos *iguales* pero no tienen porque ser el mismo (de igual forma que dos mellizos pueden ser idénticos, pero no por ello dejan de ser dos personas).

Los datos (estado) y funciones relacionadas (comportamiento) están agrupados en una sola entidad (objeto), en lugar de estar dispersos en el código. Es decir, el objeto *encapsula* los datos y el comportamiento. Como desarrollador debemos pensar en modelar nuestro sistema como un conjunto de objetos, en lugar de como un conjunto de funciones (procedimientos) invocadas una tras otra y pasándose datos más o menos arbitrarios para solucionar el problema.

Antonio Cano Galindo

Notación literal

La forma básica de crear un objeto en javascript mediante notación literal tiene la siguiente sintaxis:




```
const objetoLiteral1 = {
    propiedad1: "valor1",
    "propiedad 2": "valor2"
}
```

El acceso a las propiedades de un objeto se realiza mediante . o []

Si el nombre de la propiedad tiene espacios en blanco o caracteres no permitidos para definir una variable, deberemos usar [].

- objetoLiteral1.propiedad1 devuelve “valor1”.
- objetoLiteral2.propiedad 2 no está permitida.
- objetoLiteral2[“propiedad 2” muestra “valor2”.

```
objetoLiteral1: {propiedad1: 'valor1', pro…
    propiedad 2: 'valor2'
    propiedad1: 'valor1'
  > [[Prototype]]: Object
objetoLiteral2: {propiedad1: 'valor3', pr…
    propiedad 2: 'valor4'
    propiedad1: 'valor3'
  > [[Prototype]]: Object
> require: ƒ require(path) {
```

```
9
10  const objetoLiteral1 = {
11      propiedad1: "valor1",
12      "propiedad 2": "valor2"
13  }
14
15  const objetoLiteral2 = {
16      propiedad1: "valor3",
17      "propiedad 2": "valor4"
18  }
```

Si necesitamos crear un segundo objeto similar a objetoLiteral1 tendríamos que replicar manualmente el código anterior o crear una plantilla que nos permita crear múltiples clones del modelo. Los objetos objetoLiteral1 y objetoLiteral2 son similares, pero completamente independientes. No heredan comportamiento.

Propiedades de un objeto

Las propiedades de un objeto pueden ser de cualquiera de los tipos permitidos en javascript.

Podemos usar tanto tipos simples (números, strings o booleanos) como tipos complejos (arrays, funciones y, por supuesto, otro objeto). Cuando una propiedad es una función, en POO se le llama **método.**


```
23
24  const objetoCompuesto = {
25      propiedad1: "valor1",
26      objetoInterno: {
27          propiedad1: "valor11",
28          objetoInterno: {
29              propiedad1: "valor111",
30              propiedad2: 12,
31          }
32      },
33      funcion1: function () {
34          console.log(this.propiedad1);
35      },
36      funcion2: (x,y) => (objetoInterno.objetoInterno.propiedad2*x + y),
37  }
```

En la creación del objeto las propiedades se separan por comas, pudiendo dejar una coma final antes de la llave de cierre.

Propiedades dinámicas

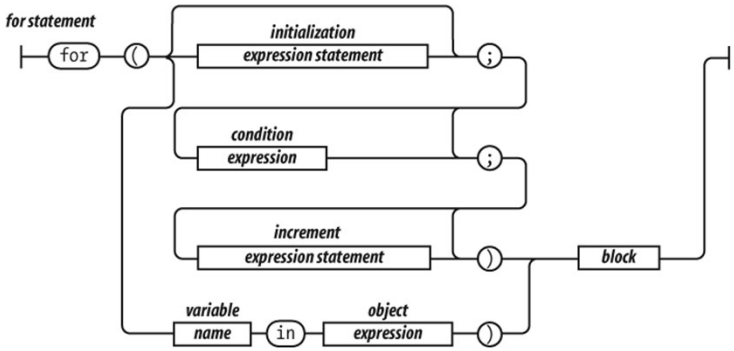
A un objeto existente se le pueden añadir propiedades en cualquier momento del ciclo de vida del objeto usando la sintaxis de <objeto>.<nueva propiedad> = <valor>. Por ejemplo: objetoLiteral1.propiedad3 = true;

Del mismo modo, también se puede borrar una propiedad en cualquier momento mediante el uso de la sentencia delete <objeto>.<propiedad>. Por ejemplo: delete objetoLiteral1.propiedad1 o delete objetLiteral1["propiedad 2"];

Acceso a las propiedades de un objeto

Acceso a una propiedad individual: con el operador . o usando [] objeto.propiedad o objeto.["propiedad"].

Enumeración de propiedades: for (let item in objeto). Con esta sentencia se itera sobre todas las propiedades del objeto, permitiendo que en cada iteración la variable item contenga cada una de las propiedades del objeto.



Object.keys(objeto) obtiene un array con los nombres de todas las propiedades de un objeto.

Object.values(objeto) obtiene un array con todos los valores de un objeto.

Podemos testear si un propiedad pertenece a un objeto con el operador in: en el ejemplo anterior la expresión if ("propiedad1" in objetoLiteral1) es true.

Métodos Getters y Setters. Acceso a propiedades

Podemos definir métodos especiales que controlen el acceso y gestionen de forma adecuada el contenido de las propiedades, protegiendo accesos no autorizados o gestionando la coherencia interna del objeto.

Los métodos getter nos permiten el acceso en lectura a la propiedad, mientras que los setter son los encargados de guardar los valores de dicha propiedad.

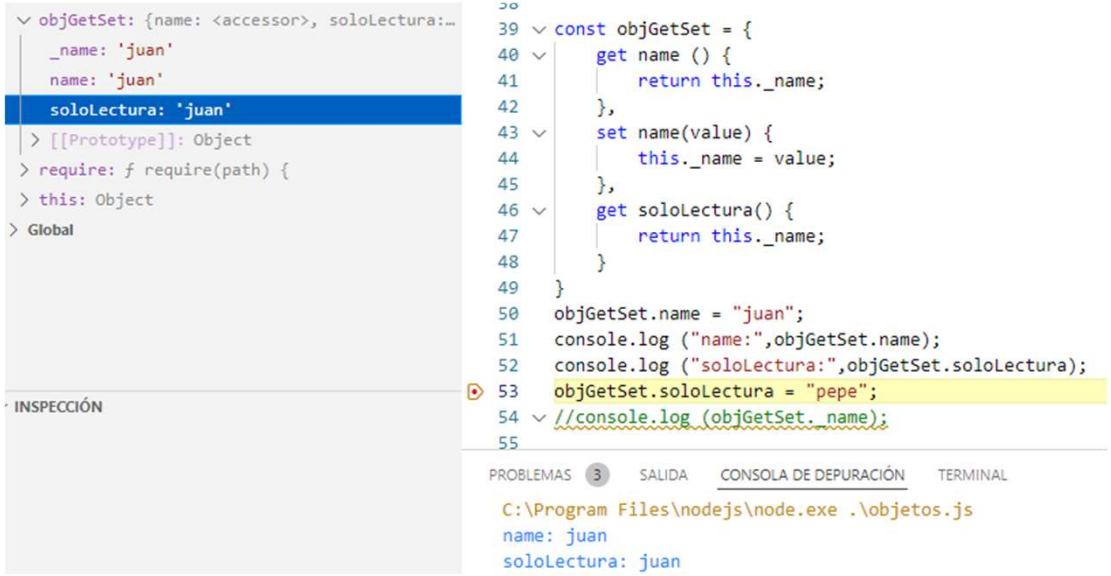
En la imagen aparece declarada la propiedad "_name", pero en ningún sitio está declarada "name".

Las funciones get name () y set name(value) se encargan de gestionar el acceso a dicha propiedad "virtual" name.

La sentencia objGetSet.soloLectura = "pepe" generará un error en tiempo de ejecución:

```
Uncaught TypeError TypeError: Cannot set property soloLectura
of #<Object> which has only a getter
```

Existe el convenio de no acceder a las propiedades prefijadas con _, ya que se consideran valores internos del objeto. Por tanto, no deberíamos usar objGetSet._name



Creación de objetos

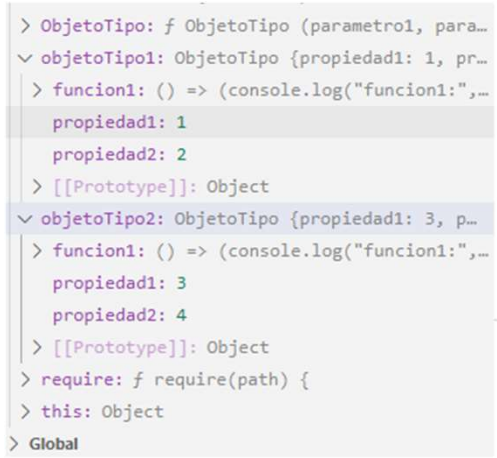
Podemos crear un objeto sin propiedades con const objeto = {} o const objeto = new Object() y añadir las propiedades de forma dinámica y posterior a la creación.

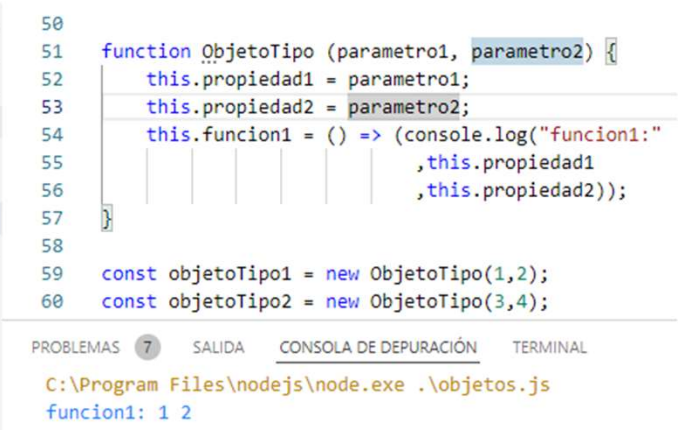
También tenemos la posibilidad de usar una función constructora de objetos:

La propiedad interna Prototype de los objetos objetoTipo1 y objetoTipo2 es Object.

Ambos objetos tiene como función constructora ObjetoTipo, al igual que heredan del constructor las propiedades y métodos informados en la función constructora.

Las propiedades del objeto creado por la función constructora se informan usando la palabra reservada this., la cual representa al objeto creado por la función.





Un objeto se crea usando new y el nombre de la función constructora. Si no se especifican los parámetros que espera el constructor asumirán el valor undefined.

Las variables objetoTipo1 y objetoTipo2 comparten el mismo prototipo.

Creación de objetos

Cuando una función es ejecutada con new, realiza los siguientes pasos:

Se crea un nuevo objeto vacío y se asigna a this.
Se ejecuta el cuerpo de la función. Normalmente se modifica this y se le agrega nuevas propiedades.
Se devuelve el valor de this.

En otras palabras, new ObjetoTipo(...) realiza algo como:

```
function ObjetoTipo(parametro1) {
  // this = {};  (implícitamente)

  // agrega propiedades a this
       this.propiedad1 = parametro1;

  // return this;  (implícitamente)
}
```

Uso de this

Uso de this

- Dentro de una función constructora o un método se refiere al propio objeto
- De forma aislada se refiere al objeto global.
- Dentro de una función se refiere al objeto global, salvo que se use modo estricto, en cuyo caso será undefined.
- Dentro de un evento se refiere al element que ha recibido el evento.

Constructores nativos de objetos y alternativas

new String()	Usamos literales con “”
new Number()	Usamos literales numéricos
new Boolean()	Usamos literales true / false
new Object()	Usamos literales { }
new Array()	Usamos literales [ ]
new RegExp()	Usamos literales de patrón / [ ] /
new Function()	Usamos expresiones de función [ ] { }
new Date()	

Prototipos

Todos los objetos en javascript heredan del prototipo Object, directa o indirectamente.

Una función constructora tiene una propiedad prototype que nos da acceso a las propiedades definidas en la función, permitiendo mantener dichas propiedades (altas, bajas y modificaciones).

Del mismo modo, cada objeto creado mediante la función constructora tendrá una variable __proto__ que será estrictamente igual que el prototipo de su función constructora.

El uso de la variable __proto__ está desaconsejado. En su lugar se han de usar los siguientes métodos de Object:

create(proto, [descriptors]) – crea un objeto vacio con el prototipo proto y descriptores de propiedades opcionales.
getPrototypeOf(obj) – devuelve el prototipo del objeto obj.
setPrototypeOf(obj, proto) – fija el prototipo de un objeto con proto.

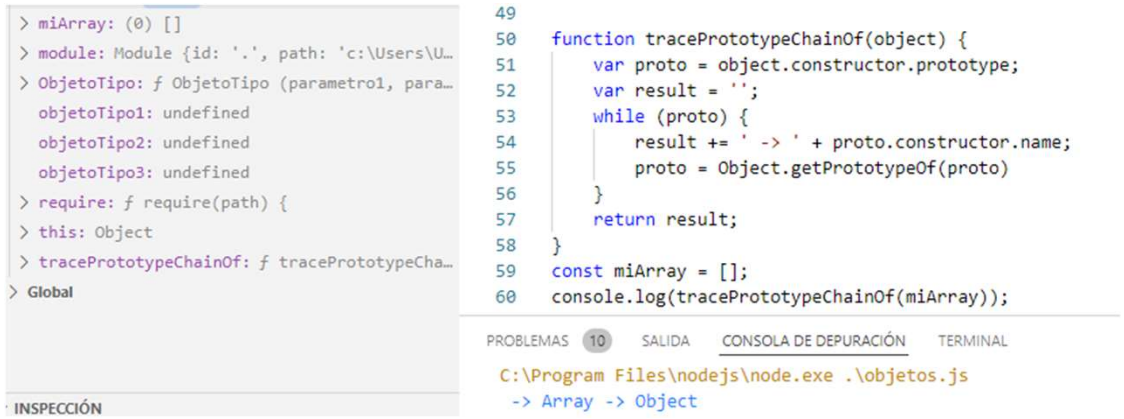
Herencia prototípica

Asumiendo que un objeto puede tener un prototipo, el cual es un objeto que a su vez puede tener otro prototipo, podemos tener una secuencia o cadena de prototipos cuyo elemento final siempre es Object.

Cuando se intenta acceder a una propiedad o método de un objeto, javascript busca en la cadena de prototipos hasta que encuentra el especificado.

Este comportamiento nos permite usar crear una jerarquía de objetos de tal modo que los objetos hijos heredan características y comportamientos de los padres sin tener que declararlos.

La secuencia prototípica de miArray es la que se muestra en la imagen: Array -> Object -> null



Antonio Cano Galindo

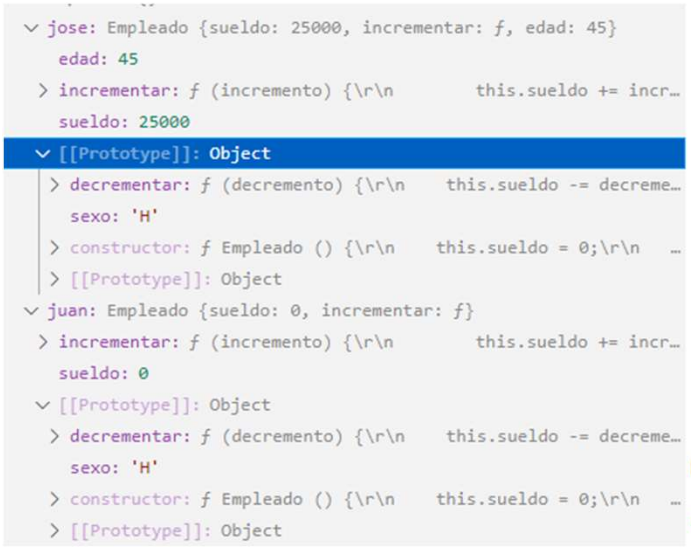
Añadir funcionalidad a un prototipo

Javascript permite modificar las propiedades de un objeto individual o a nivel de prototipo. El efecto en el primer caso es que solo ese objeto tendrá la propiedad, mientras que en el segundo todos los objetos creados a partir de ese prototipo la contendrán. Asimismo, las propiedades definidas dentro de una función constructora se replicarán en todos los objetos creados con ella.

La propiedad edad solo existe en el objeto jose, ya que la hemos creado solo para ese objeto en la línea 15.

El método incrementar está creado a nivel de objeto, de tal modo que cada objeto tiene su propio método incrementar().

Sin embargo, tanto el sexo como el método decrementar están creados a nivel de prototipo. Una vez declarados están disponibles para todas los objetos dependientes. Ahora bien, si cambiamos la propiedad sexo a 'M', se cambiará para todos los objetos.




```
3
4
5
6   function Empleado () {
7       this.sueldo = 0;
8       this.incrementar = function (incremento) {
9           this.sueldo += incremento;
10      }
11  }
12  const jose = new Empleado();
13  jose.sueldo = 25000;
14  const juan = new Empleado();
15  jose.edad = 45;
16  Empleado.prototype.sexo = "H";
17  Empleado.prototype.decrementar = function (decremento) {
18      this.sueldo -= decremento;
19  }
20  console.log (juan.sexo);
21
22
```

Herencia con prototipos

Para obtener una jerarquía de clases con herencia se debe asignar la propiedad prototype de la función constructora a un nuevo objeto de la función constructora de la que hereda.

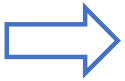
```
1  function Empleado (nombre, departamento) {
2     this.nombre = nombre || "";
3     this.departamento = departamento || "General";
4  }
5
6  function Obrero (nombre, departamento, proyectos) {
7     this.base = Empleado;
8     this.base(nombre, departamento);
9     this.proyectos = proyectos || [];
10 }
11 Object.setPrototypeOf (Obrero.prototype,Empleado.prototype);
12
13 function Ingeniero (nombre, proyectos, maquina) {
14    this.base = Obrero;
15    this.base(nombre, "Ingeniería", proyectos);
16    this.maquina = maquina || "";
17 }
18 Object.setPrototypeOf (Ingeniero.prototype,Obrero.prototype);
19
20 let ingeniero = new Ingeniero("Español Español, Juan",
21                               ["xhtml", "javascript", "html5"],
22                               "Chrome");
23 console.log(ingeniero);
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL
C:\Program Files\nodejs\node.exe .\ingenierook.js
> Ingeniero {base: ƒ, nombre: 'Español Español, Juan', departamento: 'Ingeniería', proyectos: Array(3), maquina: 'Chrome'}
```

Clases

Con la llegada de EcmaScript2015 (ES6) se ha introducido el concepto de Clase con una sintaxis parecida a la de lenguajes no orientados a prototipos. Las clases de JavaScript no son más que azúcar sintáctico sobre las funciones de herencia y constructor basadas en prototipos. A continuación se muestra la

```
1   function C () {}
2   C.prototype.x = function () {
3     return 1;
4   };
5
6   function CC () {
7   }
8   C.prototype.y = function () {
9     return 2;
10  };
11  Object.setPrototypeOf( CC.prototype, C.prototype );
12
13  const cc = new CC();
14  console.assert( cc.x() === 1 );
15  console.assert( cc.y() === 2 );
16  console.assert( cc instanceof C );
```



```
1   class C {
2     x () {
3       return 1;
4     }
5   }
6
7   class CC extends C {
8     y () {
9       return 2;
10    }
11  }
12
13  const cc = new CC();
14  console.assert( cc.x() === 1 );
15  console.assert( cc.y() === 2 );
16  console.assert( cc instanceof C );
```

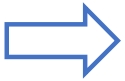
Clases

Con la llegada de EcmaScript2015 (ES6) se ha introducido el concepto de Clase con una sintaxis parecida a la de lenguajes no orientados a prototipos. Las clases de JavaScript no son más que azúcar sintáctico sobre las funciones de herencia y constructor basadas en prototipos. A continuación se muestra la equivalencia.

```js
function C () {}
C.prototype.x = function () {
  return 1;
};

function CC () {
}
C.prototype.y = function () {
  return 2;
};
Object.setPrototypeOf( CC.prototype, C.prototype );

const cc = new CC();
console.assert( cc.x() === 1 );
console.assert( cc.y() === 2 );
console.assert( cc instanceof C );
```



```js
class C {
  x () {
    return 1;
  }
}

class CC extends C {
  y () {
    return 2;
  }
}

const cc = new CC();
console.assert( cc.x() === 1 );
console.assert( cc.y() === 2 );
console.assert( cc instanceof C );
```

Clases. Uso de super

```
1  // Clase padre
2  class Forma {
3      constructor(x, y) {
4          this.x = x;
5          this.y = y;
6          console.log("Soy una forma geométrica.");
7      }
8      superficie() {
9          return 0;
10     }
11 }
12 // Clases hijas
13 class Cuadrado extends Forma {
14     constructor(x, y, lado) {
15         super(x, y);
16         this.lado = lado;
17         console.log("Soy un cuadrado.");
18     }
19     superficie() {
20         return this.lado**2;
21     }
22 }
23 class Circulo extends Forma {
24     constructor(x, y, radio) {
25         super(x, y);
26         this.radio = radio;
27         console.log("Soy un círculo.");
28     }
29     superficie () {
30         return Math.PI*this.radio**2;
31     }
32 }
33
34 let cuadrado = new Cuadrado(1,1,15);
35 let circulo = new Circulo (1,1,8);
36 console.log(cuadrado.superficie());
37 console.log(circulo.superficie());
```

Las clases hijas heredan de la clase padre mediante el uso de la palabra reservada extends.

La creación de un nuevo cuadrado se realiza mediante new Cuadrado (1,1,15). Cuando se ejecuta esta sentencia se invoca al método constructor de la clase, el cual a su vez y de forma opcional puede invocar al constructor de la clase padre mediante super(x, y). La palabra reservada super se puede usar en cualquier función redefinida en una clase hija.

Los métodos de instancia o de objeto se especifican sin la palabra function: superficie () es un ejemplo de ello.

Los métodos y propiedades de clase requieren de la palabra reservada static antes del nombre del método. Para invocarlo se necesita el nombre de la clase en vez del nombre del objeto. No se accede a través de un objeto.

Se pueden declarar propiedades y métodos privados anteponiendo # al nombre de la propiedad o método(ES2020).