

# Herencia (Parte I)

## Herencia

### Competencias

- Utilizar herencia para la reutilización de componentes en el contexto de la Programación Orientada a Objetos.
- Implementar herencias a partir de prototipos mediante la sintaxis de ES5 y ES6.

### Introducción

Una de las buenas prácticas que debemos tener en consideración cuando desarrollamos software es la de no repetir código, el principio “Don’t Repeat Yourself”, debe ser uno de los primeros a tener en consideración cada vez que debemos desarrollar una nueva tarea. La Programación Orientada a Objetos posee varios pilares, uno de ellos es la herencia, la cual nos permite utilizar un enfoque práctico para reutilizar código y no repetirnos a nosotros mismos.

Es importante conocer este concepto, tanto teóricamente como en la práctica, dado que frente a un problema dado, si hacemos un buen análisis y definimos de manera adecuada la herencia que podríamos establecer, nos simplificará de muy buena forma nuestro desarrollo. Al final de este capítulo, serás capaz de entender y aplicar el concepto de herencia en JavaScript, y además de conocer cómo funciona la cadena de prototipos.

## Herencia

Para iniciar con el concepto de Herencia, primero veamos el significado de la palabra de manera general según la Real Academia Española (RAE), quien indica que Herencia es un: “Conjunto de caracteres que los seres vivos reciben de sus progenitores.”, en otras palabras, serán los rasgos y características que un hijo recibe de su padre, abuelos y ancestros. Ahora bien, en programación, específicamente en POO, cuando hablamos de herencia, nos referimos a la forma con la cual una clase permite heredar características (métodos y atributos) a otra clase. Permitiendo construir nuevas clases a partir de otras, a fin de reutilizar código, generando así una jerarquía de clases. En efecto, si una clase hereda a partir de otra, esta puede utilizar los métodos y atributos definidos en la clase de la cual está heredando, además de poder definir nuevos métodos y atributos propios.

Para ejemplificar esto, veamos el siguiente diagrama en donde se puede observar como existe una clase superior o padre denominada “vehículo”, y de ella dependen otras clases hijas que pueden heredar los atributos y métodos de la clase padre.

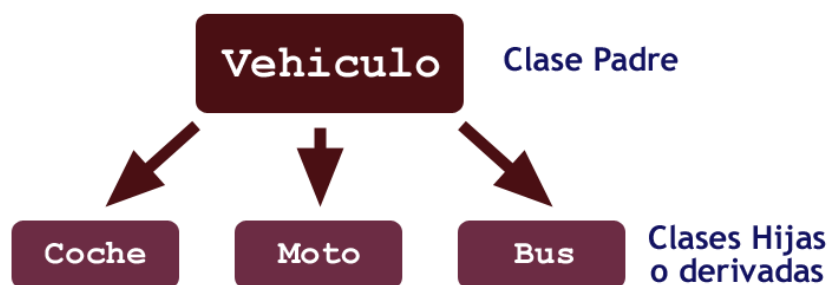


Imagen 1. Herencia de Clases.

Fuente: [desarrolloweb](http://desarrolloweb.com)

En la imagen anterior, se puede observar como una clase padre o “*superclase*”, denominada “vehículo”, podría poseer algunas propiedades o atributos como: marca, color, matrícula, entre otros, solo por mencionar algunos atributos básicos. Pero, si observamos las clases hijas que derivan de la clase padre, estas también son vehículos, lo único es que cambian de forma, ya sea un coche (sedán, deportivo), una moto o un bus, es decir, que éstas clases hijas podrían fácilmente poseer los mismos atributos (marca, color, matrícula) y acceso a los métodos que tenga la clase padre, como por ejemplo “encender, rodar, entre otros”. Por consiguiente, pueden heredar esas propiedades que posee la clase padre, utilizándolas, evitando escribir nuevamente esas propiedades para cada una de las clases hijas.

## Ejercicio guiado: Herencia con vehículos

Crear una clase padre con el nombre de "Vehículo" en conjunto con los atributos "marca", "color", "matrícula", además de las tres clases hijas "coche", "moto" y "bus", heredando los atributos de la clase vehículo. Finalmente, instanciar cada una de las clases hijas y mostrar una de las propiedades para cada una de ellas.

- **Paso 1:** Lo primero que debemos hacer, es crear una carpeta en nuestro lugar de trabajo y dentro de esta carpeta se debe crear un archivo de JavaScript con el nombre de "script.js", luego, dentro del archivo debes crear la clase padre, la cual llevará el nombre de "Vehículo", y dentro de su constructor, las propiedades mencionadas anteriormente "marca, color y matrícula", como se muestra a continuación:

```
class Vehiculo{
  constructor(marca, color, matricula) {
    this.marca = marca;
    this.color = color;
    this.matricula = matricula;
  }
}
```

- **Paso 2:** En el código anterior, se puede observar como el constructor de la clase alberga tres propiedades que serán comunes igualmente para las clases que se derivan de ella, es decir, para las clases hijas denominadas: coche, moto y bus. Por ende, en ES6 simplemente tenemos que utilizar la palabra reservada **"extends"** después de la declaración de la clase e indicar el nombre de la clase de la cual queremos que herede todos los métodos y propiedades. Como se muestra a continuación en el siguiente trozo de código:

```
class Vehiculo{
  constructor(marca, color, matricula) {
    this.marca = marca;
    this.color = color;
    this.matricula = matricula;
  }
}

class Coche extends Vehiculo{ //la clase coche es hija de vehículo
}
```

```
class Moto extends Vehiculo{ //la clase moto es hija de vehículo
}
class Bus extends Vehiculo{ //la clase bus es hija de vehículo
}
```

- **Paso 3:** Se puede observar en el código anterior como la clase **Coche** se extiende de la clase **Vehículo**, lo cual, permitirá a la clase Coche utilizar las propiedades que ya tiene declarada en el constructor la clase Vehiculo. Ahora bien, para lograr esto se debe implementar una nueva instrucción en el constructor de la clase hija “Coche” mediante la palabra reservada “**super**”, pasando como argumento los valores de las propiedades que deseamos inicializar de la clase padre, por lo que no es necesario inicializar las propiedades en la clase hija. Ahora, agreguemos el constructor a las clases hijas “marca, color y matrícula” como se muestra a continuación.

```
class Vehiculo{
    constructor(marca, color, matricula) {
        this.marca = marca;
        this.color = color;
        this.matricula = matricula;
    }
}

class Coche extends Vehiculo{
    constructor(marca,color,matricula){
        super(marca,color,matricula);
    }
}

class Moto extends Vehiculo{
    constructor(marca,color,matricula){
        super(marca,color,matricula);
    }
}

class Bus extends Vehiculo{
    constructor(marca,color,matricula){
        super(marca,color,matricula);
    }
}
```

- **Paso 4:** En el código anterior, se puede observar como las clases de Coche, Moto y Bus son extensiones de una clase padre o principal, en este caso la clase Vehiculo, heredando las propiedades que posee esta clase mediante la palabra reservada “super” en el constructor de cada clase hija. Es importante destacar que los argumentos que se envían mediante la instrucción `super`, deben ser de las mismas características con que fueron inicializados en la clase padre. Ahora comprobemos que realmente se pueden utilizar las propiedades de una clase padre, pasando los valores desde las clases hijas, instanciando cada una de las clases hijas y mostrando por consola las propiedades directamente.

```
let coche1 = new Coche('Toyota', 'Negro', '123ABC');
let moto1 = new Moto('Honda', 'Rojo', '456CDF');
let bus1 = new Bus('Fuso', 'Blanco', '678EDC');

console.log(coche1);
console.log(moto1);
console.log(bus1);
console.log(coche1.marca);
console.log(moto1.color);
console.log(bus1.matricula);
```

- **Paso 5:** Si observas detalladamente en el código anterior, nunca se hizo una instancia para la clase padre con el nombre de “Vehículo”, esto se debe a que realmente no hace falta, ya que las clases hijas heredarán mediante la palabra reservada `extends` todas esas características de su clase padre. Ahora, el resultado lo podemos obtener al ejecutar el archivo “script.js” mediante la terminal con Node, con el comando: `node script.js`

```
Coche { marca: 'Toyota', color: 'Negro', matricula: '123ABC' }
Moto { marca: 'Honda', color: 'Rojo', matricula: '456CDF' }
Bus { marca: 'Fuso', color: 'Blanco', matricula: '678EDC' }
Toyota
Rojo
678EDC
```

En el resultado anterior, podemos observar cómo se muestran los valores de cada una de las propiedades que tienen las clases hijas, sin necesidad de inicializarlas directamente en su constructor, por consiguiente, podemos reutilizar parte del código ya creado.

## Ejercicio guiado: Herencia con personal administrativo

Una institución de educación solicita que se cree un sistema para centralizar la información del personal que trabaja en ella, como es el caso del personal administrativo, docente y obrero, partiendo de un Diagrama de Clases UML. Se solicita:

- Crear las clases, inicializar los constructores para las propiedades donde corresponda y comprobar cómo automáticamente se hereda un constructor sin la necesidad de indicarlo en una clase hija. Finalmente, instanciar y mostrar por consola los valores instanciados en las clases hijas.

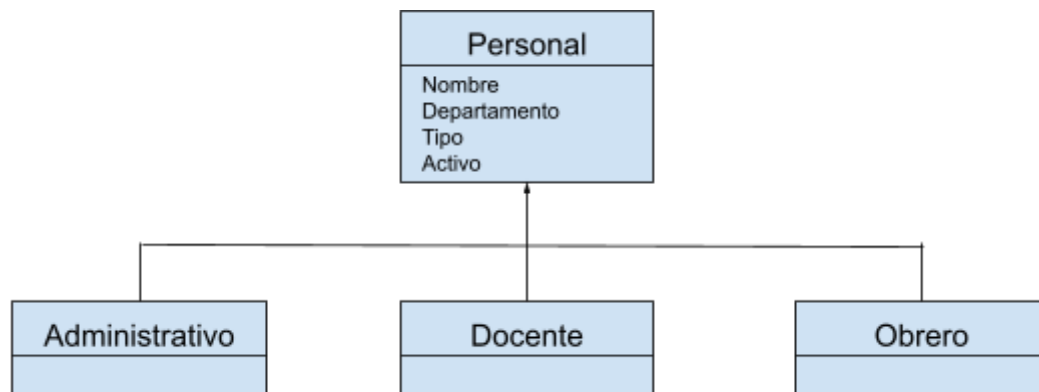


Imagen 2. Diagrama de representación de herencia de clases.

Fuente: Desafío Latam

En la imagen anterior, es decir, en el Diagrama de Clases, se puede observar como las clases Administrativo, Docente y Obrero dependen y son parte de una clase superior denominada Personal, heredando las propiedades indicadas en la clase principal, por ende, no se agregan a las clases hijas.

- **Paso 1:** Crear una carpeta en nuestro lugar de trabajo y dentro de esta carpeta se debe crear un archivo de JavaScript con el nombre de "script.js", luego, dentro del archivo primeramente debes crear la clase padre, la cual llevará el nombre de "Personal", y dentro de su constructor, las propiedades mencionadas anteriormente "nombre, departamento, tipo y activo", como se muestra a continuación:

```
class Personal{  
  constructor(nombre, departamento, tipo, activo) {  
    this.nombre = nombre;  
    this.departamento = departamento;  
    this.tipo = tipo;  
    this.activo = activo;  
  }  
}
```

- **Paso 2:** Crear las clases hijas mediante la instrucción "extends" después de la declaración de la clase, para así heredar todos los atributos de la clase padre. Seguidamente, aplicaremos el constructor con la palabra reservada **super** para llamar y acceder a las propiedades de la clase padre. En este caso, solo aplicaremos el constructor directamente a las clases hijas Administrativo y Docente, mientras que a Obrero, no aplicaremos ningún constructor, vamos a ahora como quedaría el código:

```
class Administrativo extends Personal {  
  constructor(nombre, departamento, tipo, activo){  
    super(nombre, departamento, tipo, activo);  
  }  
}  
class Docente extends Personal {  
  constructor(nombre, departamento, tipo, activo){  
    super(nombre, departamento, tipo, activo);  
  }  
}  
class Obrero extends Personal {  
}
```

- **Paso 3:** Se puede observar en el código anterior, como las clases **Administrativo** y **Docente** se extiende de la clase **Personal**, lo cual permitirá a las clases hijas utilizar las propiedades que ya se tienen declaradas en el constructor la clase padre. Ahora bien, también se puede observar como la clase **Obrero** no tiene ni un tipo de constructor, es decir, se encuentra completamente en blanco. Pero, gracias a la herencia, esta clase también heredará y podrá utilizar las propiedades o atributos de la clase padre cuando creamos la instancia. Como se muestra a continuación:

```
let admin1 = new Administrativo('Jocelyn', 'Contenido', 'Fijo', true);
let docente1 = new Docente('Juan', 'FrontEnd', 'Contratado', true);
let obrero1 = new Obrero('Manuel', 'Electricidad', 'Fijo', true);
console.log(admin1);
console.log(docente1);
console.log(obrero1);
```

- **Paso 4:** Ahora, el resultado lo podemos obtener al ejecutar el archivo “script.js” mediante la terminal con Node, con el comando: `node script.js`, resultando:

```
Administrativo {
  nombre: 'Jocelyn',
  departamento: 'Contenido',
  tipo: 'Fijo',
  activo: true
}
Docente {
  nombre: 'Juan',
  departamento: 'FrontEnd',
  tipo: 'Contratado',
  activo: true
}
Obrero {
  nombre: 'Manuel',
  departamento: 'Electricidad',
  tipo: 'Fijo',
  activo: true
}
```

Revisando el resultado obtenido anteriormente, se puede apreciar, cómo cada uno de los objetos pertenecientes a las clases hijas, cuenta con las propiedades heredadas de la clase padre, sin tener que repetir cada inicialización de cada propiedad, igualmente como la clase **Obrero**, a pesar que no cuenta con un constructor, automáticamente hereda todas las propiedades de la clase padre por ser una extensión.



En conclusión, la Herencia nos permite crear una subclase o clase hija que posea automáticamente las propiedades y métodos de la superclase o clase padre. De esta forma podemos reutilizar código y agilizar el proceso de programación. Igualmente, recuerda que un constructor o un método para que puedan llamar a miembros de la clase padre debe utilizar la palabra reservada “**super**”, que es una referencia a la clase superior. Por otra parte, en el caso que la clase hija no disponga de un constructor, siempre se incluye un constructor por defecto, que llama al constructor de la clase padre con los parámetros que se hayan pasado al realizar la instancia con *new*.

### Ejercicio propuesto (1)

A partir del siguiente diagrama de clases, crea un programa en JavaScript que permite utilizar los atributos de una clase superior en las clases hijas, además de agregar atributos propios para cada una de las clases hijas, instanciando cada clase y mostrando el resultado.

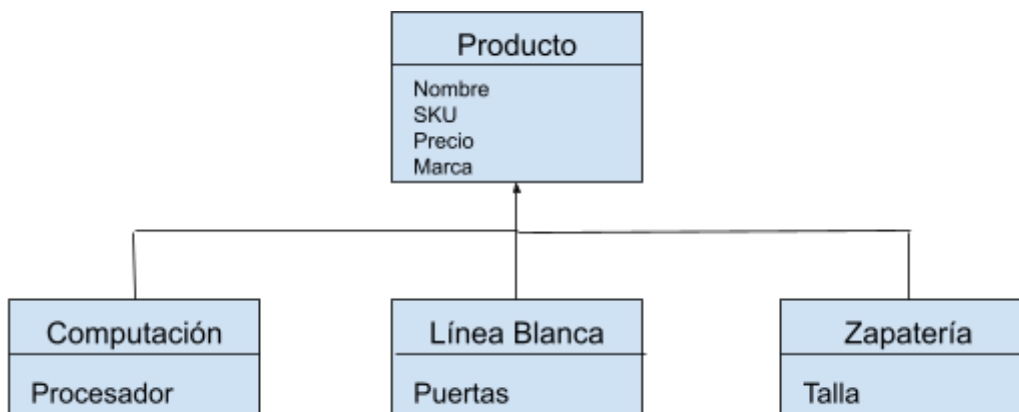


Imagen 3. Diagrama de representación de herencia de clases.

Fuente: Desafío Latam

### Herencia de prototipos

En temas anteriores, logramos trabajar con funciones constructoras para crear clases mediante la notación de ES5, igualmente se agregaron métodos a las funciones constructoras mediante la propiedad “*prototype*” o prototipo. Si recordamos un poco, en JavaScript todo objeto tiene una referencia a otro objeto llamado prototipo (Prototype), por lo tanto, las clases como logramos trabajar hasta el momento son simplemente objetos que poseen prototipos. Por ende, Prototype es una propiedad global que está disponible en casi todos los objetos, además un prototipo es un objeto del que otros objetos heredan propiedades, y los objetos siempre heredan propiedades de algún objeto anterior, tal cual como se muestra en la imagen a continuación.

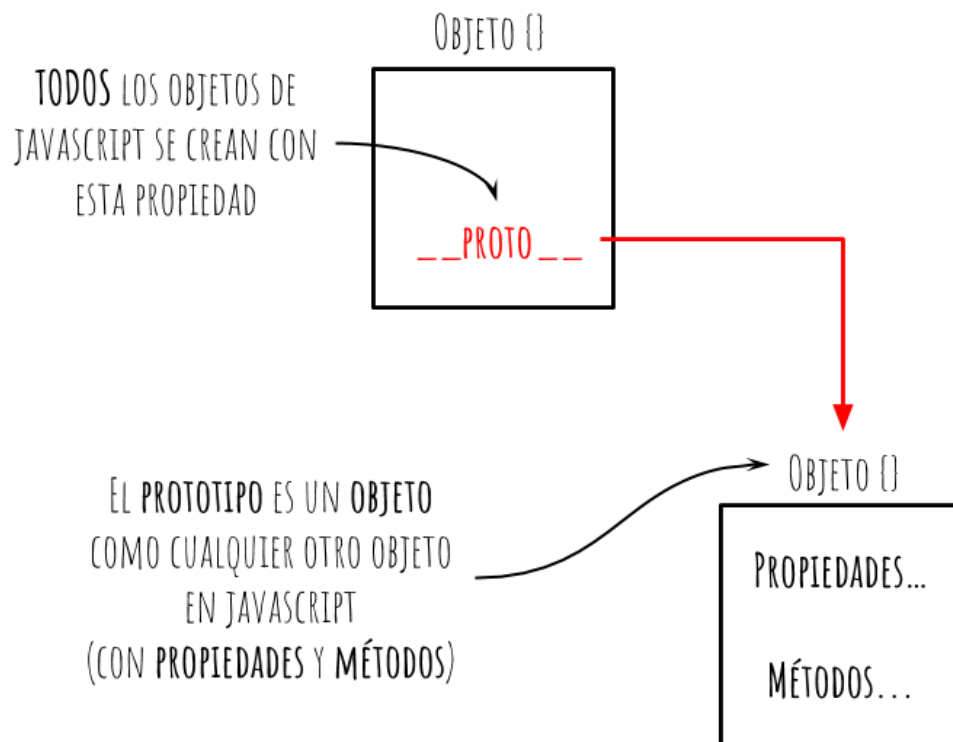


Imagen 4. Herencia basada en prototipos de JavaScript.

Fuente: [gustavodohara](#)

Este principio básico de reusabilidad de código es lo que llamamos herencia basada en clases. Sin embargo, JavaScript no posee el concepto de clases sino de prototipos. Implementar un sistema de este tipo en JavaScript es lo que conocemos como delegación basada en herencia o herencia prototípica. Tras la idea de la herencia, independientemente del sistema (clases o prototipos), consiste en declarar un objeto (o clase) cuyos métodos se heredan por referencia en otros objetos.

Veamos el siguiente ejemplo, donde creamos un objeto con una propiedad y un método, luego crearemos un nuevo objeto, al cual le asignaremos como objeto padre el primer objeto creado, esto se logrará mediante el método `Object.create()`, quien crea un objeto nuevo utilizando un objeto existente como el prototipo del nuevo objeto creado, luego mostraremos por consola ambos objetos para ver el prototipo de cada uno.

```
var persona_uno = {  
  nombre: "Juan",  
  saludar: function(){  
    console.log("Hola, soy "+this.nombre);  
  }  
}  
  
console.log(persona_uno);
```

```
var persona_dos = Object.create(persona_uno);  
console.log(persona_dos);
```

Al ejecutar el código anterior en la consola del navegador web, el resultado será:

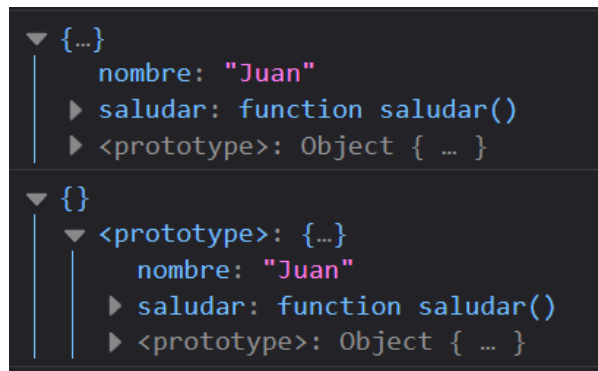


Imagen 5. Mostrando el prototipo de un objeto con herencia de otro objeto.

Fuente: Desafío Latam

Al observar con detalle la imagen anterior, podemos apreciar como el resultado mostrado en la primera parte, perteneciente a la línea de código `console.log(persona_uno);` muestra directamente el valor del objeto `persona_uno`, mientras que para la de código línea `console.log(persona_dos);` muestra el objeto `persona_dos` vacío, pero con el objeto prototype disponible perteneciente al objeto padre. Cumpliéndose el principio de herencia prototípica.

## Ejercicio guiado: Herencia prototípica

Realizar el siguiente ejercicio de clases mediante herencia prototípica con nomenclatura de ES5, para ver en acción los prototipos, primeramente con un ejercicio sin métodos en la clase padre. Luego, convertir el código anterior a uno actualizado implementado ES6.

Crear una clase padre o principal, mediante una función constructora con el nombre de "FiguraGeometrica", la cual recibe como parámetros los valores de un lado de la figura. Además, se solicita crear una clase denominada "Polígono", pero que herede las propiedades de la clase padre. Además, la clase Polígono debe poseer un método propio que permita calcular el área de la figura geométrica con las propiedades inicializadas en la clase padre.

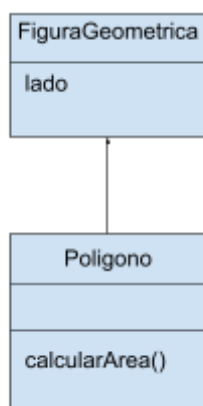


Imagen 6. Diagrama de Clases con Herencia.  
Fuente: Desafío Latam

Es importante destacar que en este ejemplo, no se busca entrar en detalles con respecto a las propiedades matemáticas de las figuras geométricas, sólo se emplea como ejemplo para comprender la herencia de propiedades de una clase padre a una clase hija.

- **Paso 1:** Crear una carpeta en nuestro lugar de trabajo y dentro de esta carpeta se debe crear un archivo de JavaScript con el nombre de "script.js", luego, dentro del archivo debes crear la clase padre mediante la función constructora, la cual llevará el nombre de "**FiguraGeometrica**", recibiendo como parámetros las propiedades mencionadas anteriormente "lado", como se muestra a continuación:

```
function FiguraGeometrica(lado){  
  this.lado = lado;  
}
```

- **Paso 2:** Crear la función constructora para la clase de Polígono, pero como esta clase será hija, es decir, hereda de una clase padre "**FiguraGeometrica**" los atributos y métodos que ella posea, se debe hacer el llamado a las propiedades de la clase padre, para ello se debe implementar el método `call()`, el cual llama a una función con un valor "this" asignado, además de los argumentos provistos de forma individual, en este caso, los argumentos serán el lado que recibe el Polígono, y se pasara dos veces para completar los valores que necesita la función constructora de nuestra clase padre, para luego ser inicializados y asignados a las propiedades respectivas.

```
function Poligono(lado){  
  FiguraGeometrica.call(this, lado);  
}
```

- **Paso 3:** Ya nuestra clase hija "Polígono", realiza el llamado a las propiedades de la clase padre, por lo que hace falta indicarle a ambas clases que una es extensión de la otra, esto lo lograremos mediante la instrucción "**Object.create**", asignando al prototipo de la clase hija "Polígono" como nuevo objeto el prototipo de la clase padre:

```
function Poligono(lado){  
  FiguraGeometrica.call(this, lado);  
}  
Poligono.prototype = Object.create(FiguraGeometrica.prototype);
```

- **Paso 4:** El último paso para poder completar la clase hija con lo indicado en el ejercicio, sería crear un método para el cálculo del área del Polígono, el cual se realizará como se ha trabajado hasta el momento, agregándolo al prototipo de la clase, como se muestra a continuación:

```
Poligono.prototype.calcularArea = function(){  
    return this.lado * this.lado;  
}
```

- **Paso 5:** Finalmente, nos queda por instanciar la clase hija, pasar los valores del lado, llamar al método para calcular el área del Polígono y mostrarlo mediante un `console.log()`. Aprovecharemos la instancia que crearemos para el Polígono para mostrarla y observar el contenido de la misma.

```
var cuadrado = new Poligono(3);  
console.log(cuadrado);  
console.log(cuadrado.calcularArea());
```

- **Paso 6:** Ahora, el resultado lo podemos obtener al ejecutar el archivo “script.js” mediante la terminal con Node, con el comando: `node script.js`, resultando:

```
FiguraGeometrica { lado: 3 }  
9
```

En el resultado anterior, se puede observar como la clase hija “Polígono” instanciada en el objeto con nombre “cuadrado”, muestra la clase padre como su función constructora. Ya que no le estamos indicando que utilice directamente las propiedades desde la clase hija, sino desde la clase padre.

Continuando con el ejercicio anterior, vamos a transformar ese código escrito en ES5, en un código más actual implementando las clases de ES6 aplicando herencia. Por lo tanto, debemos seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en nuestro lugar de trabajo y dentro de esta carpeta se debe crear un archivo JavaScript con el nombre de "script.js", luego dentro del archivo primeramente debes crear la clase padre "**FiguraGeometrica**", recibiendo como parámetros en el constructor las propiedades mencionadas anteriormente "lado", como se muestra a continuación:

```
class FiguraGeometrica {  
  constructor(lado){  
    this.lado = lado;  
  }  
}
```

- **Paso 2:** Crear la clase hija con el nombre de **Polígono**, la cual, será una extensión de la clase creada anteriormente denominada **FiguraGeometrica**, recibiendo en su constructor los valores necesarios para asignar a los atributos, como lo son el alto y el ancho. Luego, mediante la palabra reservada `super()`, hacemos el llamado al constructor con los atributos de la clase padre y enviamos los valores recibidos en el constructor de la clase hija. Esto es lo que se hizo utilizando el método `call()` en ES5.

```
class Poligono extends FiguraGeometrica {  
  constructor(lado){  
    super(lado);  
  }  
}
```

- **Paso 3:** Para finalizar con la clase hija, solo hace falta crear el método que nos permita calcular el área del cuadrado. Este método retorna la multiplicación del atributo lado por lado, lo implementaremos mediante la nomenclatura de ES6:

```
class Poligono extends FiguraGeometrica {  
  constructor(lado){  
    super(lado);  
  }  
  calcularArea(){  
    return this.lado * this.lado;  
  }  
}
```

- **Paso 5:** Finalmente, nos queda por instanciar la clase hija, pasar los valores de alto y ancho, llamar al método para calcular el área del Polígono y mostrarlo mediante un `console.log()`. Aprovecharemos la instancia que crearemos para el cuadrado para mostrarla y observar el contenido de la misma.

```
var cuadrado = new Poligono(3);  
console.log(cuadrado);  
console.log(cuadrado.calcularArea());
```

- **Paso 6:** Ahora, el resultado lo podemos obtener al ejecutar el archivo "script.js" mediante la terminal con Node, con el comando: `node script.js`, resultando:

```
FiguraGeometrica { lado: 3 }  
9
```

En el resultado anterior, se puede observar como la clase hija "Polígono" instanciada en el objeto con nombre "cuadrado", muestra las propiedades como si fueran propias o creadas en esa clase, ya que al utilizar las palabras reservadas de ES6 "extends" y "super", automáticamente JavaScript mediante los prototipos apunta directamente al constructor de la clase hija. Esto se puede lograr también en ES5 utilizando la instrucción: `Poligono.prototype.constructor = Poligono;`

En conclusión, la Herencia en JavaScript funciona mediante prototipos, se puede tener acceso a las propiedades de un objeto desde otro objeto mientras se encuentren relacionados entre si, lo cual facilita compartir las propiedades y métodos de un objeto con otro. Pero hasta el momento solo logramos trabajar con la herencia de prototipos para heredar propiedades, veamos ahora en el siguiente tema mediante la cadena de prototipos como una clase padre hereda sus métodos a otras clases.



## Ejercicio propuesto (2)

Transforma el siguiente código de ES5 a ES6 implementando herencia de clases.

```
function Curso(titulo,turno,alumno) {
    this.titulo = titulo;
    this.turno = turno;
    this.alumno = alumno;
}

Curso.prototype.inscribirAlumno = function() {
    console.log(`El alumno: ${this.alumno}, se ha inscrito en el
curso: ${this.titulo}, perteneciente al turno: ${this.turno}`);
}

function JavaScript(titulo,turno,alumno,nivel,tema) {
    Curso.call(this,titulo,turno,alumno);
    this.nivel = nivel;
    this.tema = tema;
}

JavaScript.prototype = Object.create(Curso.prototype);

JavaScript.prototype.temaSolicitado = function () {
    console.log(`El nivel y el tema solicitado fueron: ${this.nivel} y
${this.tema} respectivamente`);
}

var alumno1 = new JavaScript('Programando con JavaScript', 'Nocturno',
'Juan', 'Basico', 'Introduccion');
console.log(alumno1);
alumno1.temaSolicitado();
alumno1.inscribirAlumno();
```

## Cadena de Prototipos

### Competencias

- Implementar la cadena de prototipos para lograr distintos niveles de herencia.
- Aplicar múltiples niveles de herencia que permitan reutilizar propiedades y métodos de clases superiores

### Introducción

Anteriormente, logramos trabajar con Herencia mediante prototipos y luego pasamos el código a la nomenclatura de ES6. Pero, si bien es bueno definir clases y realizar una herencia con las nuevas palabras clave de ES6, es aún mejor comprender cómo funcionan las cosas a nivel de definición, recordemos que ES6 solamente es “azúcar sintáctico” (**syntactical sugar**), para poder utilizar las palabras reservadas “class”, “extends” para que JavaScript sea más “parecido” a otros lenguajes de programación.

Por ende, es importante conocer cómo funciona la cadena de prototipos para comprender que no es exactamente igual a los lenguajes cuyo pilar es la POO. Por consiguiente, el propósito de este capítulo será el aprender a aplicar herencia mediante la cadena de prototipos con la sintaxis de ES5 y en contraparte ver cómo se aplica la herencia de múltiples niveles pero de la nueva sintaxis de ES6.

## ¿Qué es una cadena de prototipos?

Primeramente revisemos el concepto de prototipos, los cuales son un mecanismo por el cual un objeto hereda propiedades y métodos de un padre, entonces en JavaScript la herencia funciona por prototipos, tal cual como lo estudiamos en el tema anterior. Ahora bien, en una Cadena de prototipos cada objeto tiene un prototipo, y este prototipo puede tener otro prototipo, así sucesivamente hasta encontrar a un prototipo que no tiene prototipo, llegando a tener un valor nulo “null”. Por definición, null no tiene un prototipo y actúa como el enlace final de esta cadena de prototipos. Tal cual como se muestra en la imagen a continuación:

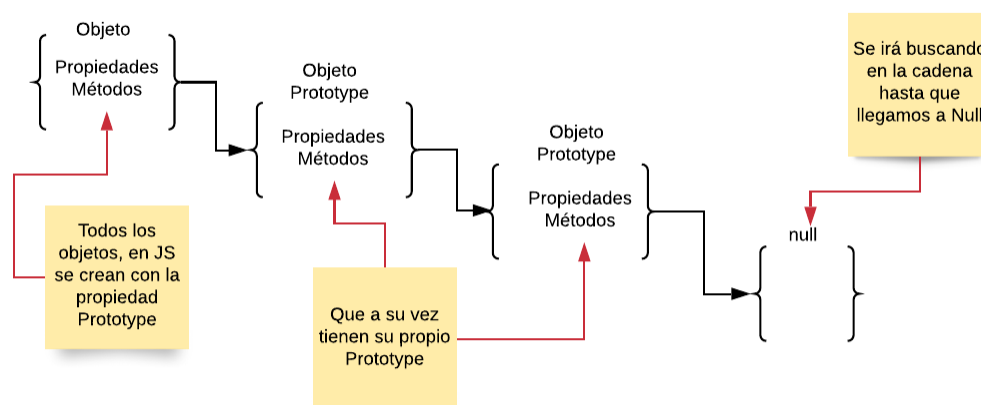


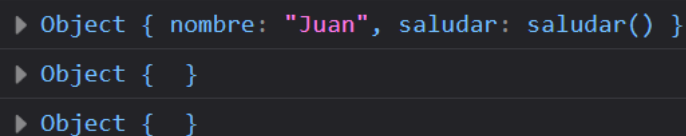
Imagen 7. Diagrama de Cadena de prototipos  
Fuente: Desafío Latam

En la imagen anterior, podemos observar cómo al crear un objeto, dejamos disponible todas sus propiedades y métodos para cualquier otro objeto que lo incluya como un objeto superior. Por ende, podemos dejar disponible un método de un objeto a otro, creando una herencia basada en cadena de prototipos, permitiendo a objetos hijos acceder y utilizar los métodos creados en un objeto padre. Es importante declarar que al referirnos a objetos, en la notación de ES6 serán las clases como se han trabajado hasta el momento.

Revisemos ahora un ejemplo, creando un objeto principal con una propiedad y un método, luego crearemos otro objeto (hijo) con el método `Object.create()`, utilizando el objeto padre como el prototipo del nuevo objeto creado (hijo), luego crearemos otro nuevo objeto siguiendo el mismo proceso para crear el objeto hijo, pero esta vez será un nieto y recibirá el prototipo del hijo.

```
var padre = {  
  nombre: "Juan",  
  saludar: function(){  
    console.log("Hola, soy "+this.nombre);  
  }  
}  
console.log(padre);  
  
var hijo = Object.create(padre);  
console.log(hijo);  
  
var nieto = Object.create(hijo);  
console.log(nieto);
```

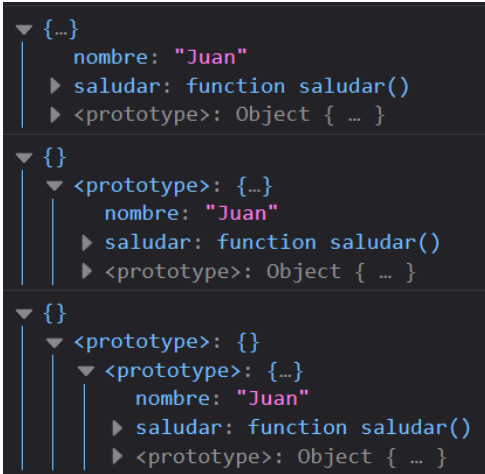
Ahora, al ejecutar el código anterior en la consola del navegador web, el primer resultado sería:



```
▶ Object { nombre: "Juan", saludar: saludar() }  
▶ Object { }  
▶ Object { }
```

Imagen 8. Resultado en al consola del navegador web  
Fuente: Desafío Latam

Por lo que podemos observar que el primer objeto perteneciente al “padre”, muestra directamente los valores que posee, mientras que los otros dos objetos (hijo y nieto) aparecen sin ningún tipo de valor porque no se les indico ningún tipo de propiedad. Pero, al hacer un clic sobre los dos objetos que se encuentran vacíos en al consola del navegador web, el resultado sería:



```
{...}  
  nombre: "Juan"  
  ▶ saludar: function saludar()  
  ▶ <prototype>: Object { ... }  
  
{}  
  <prototype>: {...}  
    nombre: "Juan"  
    ▶ saludar: function saludar()  
    ▶ <prototype>: Object { ... }  
  
{}  
  <prototype>: {}  
    <prototype>: {...}  
      nombre: "Juan"  
      ▶ saludar: function saludar()  
      ▶ <prototype>: Object { ... }
```

Imagen 9. Mostrando el prototipo de un objeto con herencia de otro objeto.  
Fuente: Desafío Latam

Ahora, se puede observar en la imagen anterior, como el primer objeto vacío (hijo) cuenta en su prototipo directamente con los valores del objeto padre. Mientras que el segundo objeto vacío, como es un nieto, cuenta en el segundo prototipo con los valores del objeto padre, en este caso abuelo, y en el primer prototipo contará con los valores “de existir” del objeto padre, en este caso el hijo. Aquí nuevamente podemos apreciar como JavaScript utiliza la herencia basada en cadena de prototipos.

## Ejercicio guiado: Herencia con comunas y sectores

### Indicaciones

- Realizar el siguiente ejercicio implementando herencia de métodos mediante la cadena de prototipos con ES5.
- Realizar el mismo ejercicio pero utilizando clases y herencia con ES6.

### Enunciado

El ejercicio posee dos clases, una clase padre denominada “Comuna”, la cual tiene como atributo “nombre y población”, además de un método denominado “calcularCenso”, que simplemente concatena y mostrará por consola el mensaje “Calculando el censo del sector...”. Por otra parte, existirá la clase “Sector”, la cual será la clase hija y podrá usar los atributos y el método que posee la clase padre. Igualmente, esta clase hija tendrá un propio atributo denominado “dirección”. En la siguiente imagen, se muestra el diagrama de clases del ejercicio.

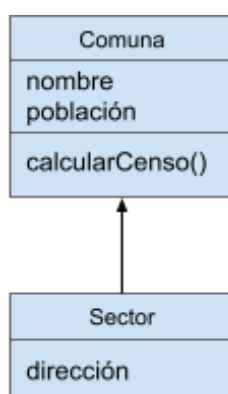


Imagen 10. Diagrama de Clases con Herencia.  
Fuente: Desafío Latam

- **Paso 1:** Crear una carpeta en nuestro lugar de trabajo y dentro de esta carpeta se debe crear un archivo con el nombre de "script.js", luego dentro del archivo se debe crear la clase padre mediante la función constructora, la cual llevará el nombre de "Comuna", recibiendo como parámetros las propiedades mencionadas anteriormente "nombre y población". Igualmente, debemos crear el método "calcularCenso" fuera de la función constructora, mediante el prototype lo asignamos al objeto Comuna, como se muestra a continuación:

```
function Comuna(nombre,poblacion){  
  this.nombre = nombre;  
  this.poblacion = poblacion;  
};  
  
Comuna.prototype.calcularCenso = function() {  
  console.log('Calculando el censo del sector...');  
};
```

- **Paso 2:** Crear la función constructora para la clase Sector, pero como esta clase será hija, hereda de una clase padre "Comuna" los atributos y métodos que ella posee, se debe hacer el llamado a las propiedades de la clase padre, para ello se debe implementar el método call(), asimismo inicializamos la propiedad interna del objeto "Sector" denominada "direccion".

```
function Sector(nombre,poblacion,direccion) {  
  Comuna.call(this, nombre,poblacion);  
  this.direccion = direccion;  
};
```

- **Paso 3:** El objeto hijo "Sector", realiza el llamado a las propiedades del objeto padre, por lo que hace falta indicarle a ambos objetos que uno es extensión de otro, esto lo lograremos mediante la instrucción "Object.create", asignando al prototipo del objeto hijo "Sector" como nuevo objeto el prototipo del objeto padre. Además, anularemos el constructor para que apunte a la función Sector; de lo contrario, usaría la propiedad del constructor en Comuna.

```
Sector.prototype = Object.create(Comuna.prototype);  
Sector.prototype.constructor = Sector;
```

- **Paso 4:** Finalmente, nos queda por instanciar el objeto hijo, pasando los valores para las propiedades. Llamando al método que posee el objeto padre denominado "calcularCenso".

```
// instancia del objeto
var centro = new Sector('Santiago', 3000000, 'Rebeca Matte 18');
// llamada a método y propiedades
console.log(centro.nombre);
console.log(centro.poblacion);
console.log(centro.direccion);
centro.calcularCenso();
```

- **Paso 5:** El resultado lo podemos obtener al ejecutar el archivo "script.js" mediante la terminal con Node, con el comando: `node script.js`, resultando:

```
Santiago
3000000
Rebeca Matte 18
Calculando el censo del sector...
```

En el resultado anterior, se puede observar el valor mostrado por el método `calcularCenso`, el cual, fue accedido desde la instancia del objeto hijo "Sector" sin la necesidad de crearlo para ese objeto, todo gracias a la cadena de prototipos.

Hasta el momento, se logró crear un objeto que herede métodos y propiedades a otros objetos implementando la nomenclatura de ES5 mediante la cadena de prototipos. Pero ahora, vamos a trabajar el ejercicio anterior implementando la nomenclatura de ES6, logrando comprender como JavaScript modificó la sintaxis del código mediante prototipos a una sintaxis de clases más prácticas y fácil de entender.

- **Paso 1:** Crear una carpeta en nuestro lugar de trabajo y dentro de esta carpeta se debe crear un archivo con el nombre de "script.js", luego dentro del archivo se debe crear la clase padre "Comuna", recibiendo como parámetros en el constructor las propiedades mencionadas anteriormente "nombre, población". Así mismo, debemos crear el método denominado "calcularCenso" dentro de la clase padre, para que muestre como string la concatenación de como se muestra a continuación:

```
class Comuna{
  constructor(nombre, poblacion){
    this.nombre = nombre;
    this.poblacion = poblacion;
  }
}
```

```
    calcularCenso(){  
        console.log('Calculando el censo del sector...');  
    }  
}
```

- **Paso 2:** Crear la clase hija con el nombre de `Sector`, la cual será una extensión de la clase creada anteriormente denominada `Comuna`, recibiendo en su constructor los valores necesarios para asignar a los atributos, como lo son el nombre y población además de agregar el atributo propio de la clase hija “direccion”. Luego, mediante la palabra reservada `super()`, hacemos el llamado al constructor con los atributos de la clase padre y enviamos los valores recibidos en el constructor de la clase hija. Esto es lo que se hizo utilizando el método `call()` en ES5.

```
class Sector extends Comuna{  
    constructor(nombre, poblacion, direccion) {  
        super(nombre, poblacion);  
        this.direccion = direccion;  
    }  
}
```

- **Paso 3:** Finalmente, nos queda por instanciar la clase hija, pasar los valores de nombre, población y dirección. Luego, se hace el llamado al método perteneciente a la clase padre desde el objeto instanciado por la clase hija.

```
var centro = new Sector('Santiago', 3000000, 'Rebeca Matte 18');  
console.log(centro.nombre);  
console.log(centro.poblacion);  
console.log(centro.direccion);  
centro.calcularCenso();
```

- **Paso 4:** Ahora, el resultado lo podemos obtener al ejecutar el archivo “script.js” mediante la terminal con Node, con el comando: `node script.js`, resultando:

```
Santiago  
3000000  
Rebeca Matte 18  
Calculando el censo del sector...
```



En el resultado anterior, se puede observar como la clase hija "Sector" instanciada en el objeto con nombre "centro", muestra las propiedades como si fueran propias o creadas en esa clase, ya que al utilizar las palabras reservadas de ES6 "extends" y "super", igualmente se muestra el resultado del método creado en la clase padre, siendo este llamado desde la clase hija.

En conclusión, la herencia nos permitirá disponer de atributos y métodos desde una clase padre a una o más clases hijas. Pero en el fondo, recordemos que JavaScript logra todo esto mediante la herencia basada en prototipos, en donde los objetos padres e hijos se vinculan entre sí para formar las cadenas de prototipos. Por consiguiente, a partir de ahora, los siguientes puntos se desarrollarán solamente con la nomenclatura de ES6, ya que comprendemos cómo funciona JavaScript en el fondo con las clases.

### Ejercicio propuesto (3)

Una empresa constructora de edificios, necesita un sistema que identifique el tipo de cliente (pequeño, mediano o grande) para poder estimar la cantidad de personal que necesite para la ejecución de la obra civil. Por ello, en este preciso momento requieren que el sistema muestre un mensaje indicando el nombre y tipo de cliente, además que se muestre el valor del código del cliente ingresado. Así mismo, la empresa solicita que el código se realice mediante la sintaxis de ES6. Por otra parte, la empresa facilitó un diagrama de clases para seguir al momento de desarrollar el código en JavaScript, este se debe ejecutar utilizando Node.

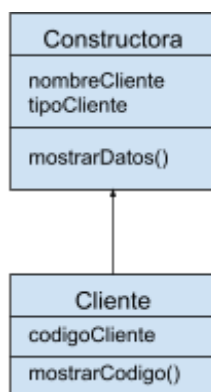


Imagen 11. Diagrama de Clases con Herencia.  
Fuente: Desafío Latam

## Múltiples niveles de herencia

También te podrías preguntar si es posible hacer herencia multinivel, es decir, tener clases padres, hijos, nietos y así sucesivamente, y la respuesta es sí. Recuerda que los objetos van dejando disponibles sus prototipos a medida que le vamos indicando a cada objeto que tendrá acceso a otro.

### Ejercicio guiado: Múltiples niveles de herencia

Realizar un programa para un taller de reparación de motores, donde necesitan clasificar los motores en eléctricos, de explosión o a vapor. La empresa solicita crear el programa solo para los motores del tipo eléctrico, los cuales se clasifican dependiendo del tipo de corriente eléctrica que requieran para su funcionamiento, siendo esta Corriente Continua (CC) o Corriente Alterna (AC). Como se muestra en el diagrama de clases a continuación:

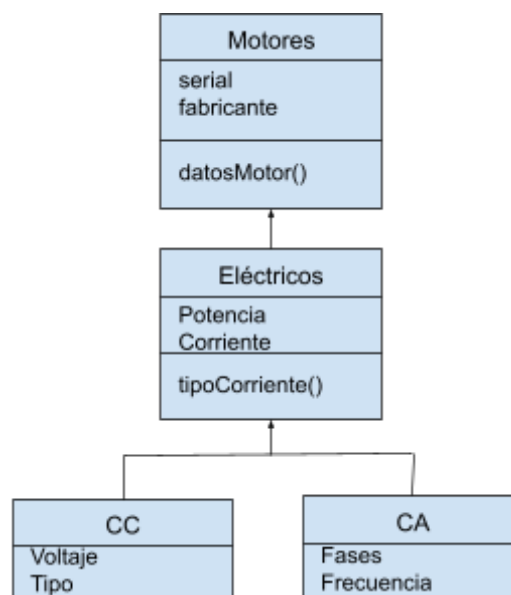


Imagen 12. Diagrama de clases.

Fuente: Desafío Latam

Ahora bien, el ejercicio solicita que mediante la implementación de clases de ES6, se lleve a código el diagrama de clases mostrado en la imagen anterior y se muestra por consola el tipo de corriente de cada motor, para ello el método “tipoCorriente” tendrá que ser llamado desde cada clase perteneciente a un tipo de motor eléctrico. Así mismo se deberán mostrar los datos del motor que incluye el serial y el fabricante.

Resolvamos ahora el ejercicio paso a paso:

- **Paso 1:** Crear una carpeta en nuestro lugar de trabajo y dentro de esta carpeta se debe crear un archivo con el nombre de "script.js", luego crearemos la clase Motores, con su constructor y método "datosMotor", para retornar los datos del motor como serial y fabricante.

```
class Motores {  
  constructor(serial,fabricante){  
    this.serial = serial;  
    this.fabricante = fabricante;  
  }  
  datosMotor(){  
    return `El número de serial es: ${this.serial} y el fabricante  
es la empresa: ${this.fabricante}`;  
  }  
}
```

- **Paso 2:** Definiremos la clase Eléctricos, la cual extiende de Motores y define el método "**tipoCorriente**", que retorna el tipo de corriente del motor eléctrico.

```
class Electricos extends Motores {  
  constructor(serial, fabricante, potencia, corriente){  
    super (serial, fabricante);  
    this.potencia = potencia;  
    this.corriente = corriente;  
  }  
  tipoCorriente(){  
    return `El tipo de corriente es: ${this.corriente}`;  
  }  
}
```

- **Paso 3:** Finalmente definiremos las clases **CC** y **CA**, las cuales serán extensiones de la clase **Eléctricos**, por lo tanto se puede decir que son nietas de la clase principal denominada **Motor**. Esto mediante la herencia permitirá acceder desde las clases más inferiores a los métodos de las clases superiores y ejecutarlos para obtener un resultado. Igualmente es importante destacar que al constructor se le deben ir indicando como parámetros todos los atributos de las clases superiores así como los propios de cada clase.

```
class CC extends Electricos {
    constructor(serial, fabricante, potencia, corriente, voltaje,
    tipo){
        super (serial, fabricante, potencia, corriente);
        this.voltaje = voltaje;
        this.tipo = tipo;
    }
}

class CA extends Electricos {
    constructor(serial, fabricante, potencia, corriente, fases,
    frecuencia){
        super (serial, fabricante, potencia, corriente);
        this.fases = fases;
        this.frecuencia = frecuencia;
    }
}
```

- **Paso 4:** Acceder a las distintas partes de nuestro código. Si generamos una instancia para cada tipo de motor, deberíamos ser capaces de acceder y ejecutar los métodos pertenecientes a las clases superiores.

```
let motorCC = new CC('133DGH', 'GE', '2000W', 'CC', '220CC', 'Shunt');
let motorCA = new CA('7566DGD', 'ABB', '2HP', 'CA', 'Monofasico',
'50Hz');
console.log(motorCC.datosMotor());
console.log(motorCC.tipoCorriente());
console.log(motorCA.datosMotor());
console.log(motorCA.tipoCorriente());
```

- **Paso 5:** Una vez definida las instancias, podremos ejecutar el código directamente desde la terminal con ayuda del comando `node script.js`. Obteniendo como resultado:

```
El número de serial es: 133DGH y el fabricante es la empresa: GE
El tipo de corriente es: CC
El número de serial es: 7566DGD y el fabricante es la empresa: ABB
El tipo de corriente es: CA
```

En el resultado anterior, se puede apreciar cómo podemos acceder a las clases superiores desde las clases inferiores sin ningún inconveniente. Utilizando los métodos que las clases padres posean sin importar el nivel o jerarquía en la que se encuentren.

### Ejercicio propuesto (4)

Realizar un programa para una tienda de instrumentos musicales, la cual necesita ordenar los instrumentos musicales que posee. La tienda solicita crear el programa para ordenar por por el tipo de instrumento (cuerda o viento) por los momentos. Por lo que requiere mostrar desde la clase superior “Tienda” la dirección del local comercial, mientras que en la clase de “Instrumento” se debe mostrar en un método el mensaje con el número de serie del instrumento. Para lograr esto, se debe crear una instancia para cada tipo de instrumento pasando como parámetros todos los datos requeridos por las clases superiores. Como se muestra en el diagrama de clases a continuación:

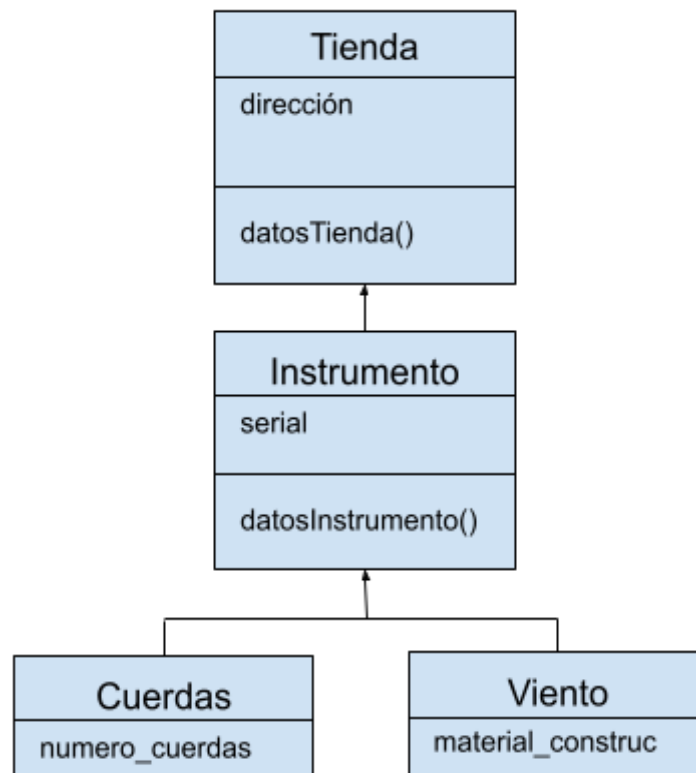


Imagen 13. Diagrama de clases.

Fuente: Desafío Latam

## Getters, Setters y HTML

### Competencias

- Construir propiedades get y set en ES6 para acceder o modificar los atributos de una clase.
- Combinar objetos, herencia y HTML para realizar ejercicio práctico.

### Introducción

En el capítulo anterior, logramos comprender como JavaScript implementa los prototipos para aplicar la herencia, al igual que la transformación que sufrió el código de ES5 a ES6 a la hora de aplicar las clases y los múltiples niveles de herencia. Pero aún no se han implementado métodos propios de ES6 a las clases que permitan acceder y/o modificar un atributo desde el exterior cuando existen múltiples niveles de herencia. Por lo tanto, en este capítulo aprenderás a aplicar métodos get y set para así acceder o modificar un atributo desde el exterior sin tener que modificarlo directamente desde el constructor, lo cual es de suma importancia porque es una buena práctica. Igualmente aprenderás a implementar el uso de las clases y herencia para manipular datos recibidos desde una archivo HTML.

## Propiedades computadas (get y set)

Al referirnos a propiedades computadas, simplemente nos estamos refiriendo al uso de getters y setters dentro de una clase, es decir, serán los métodos que nos permitirán obtener un atributo o modificarlo. En este caso, los métodos “getters” son la forma de definir propiedades computadas de lectura en una clase. Mientras que los métodos “setters” serán las propiedades computadas que permitirán modificar los atributos de una clase. También es importante recordar que el uso de la nomenclatura del guión bajo “\_” para los atributos dentro del constructor de la clase al momento de inicializarlos, lo cual permitirá diferenciarlos de los métodos get y set que apliquemos a esos atributos.

## Ejercicio guiado: Múltiples niveles de herencia con getters y setters

Crear una clase para cada nivel en conjunto con sus atributos y métodos, siendo la clase padre o superior “Animal”, mientras que las otras clases van a ir heredando de la clase padre los atributos y métodos disponibles. Como se muestra en el siguiente diagrama:

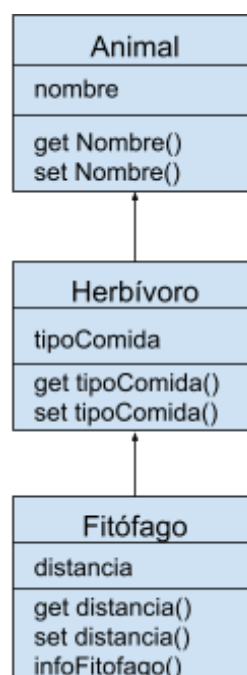


Imagen 14. Diagrama de herencia en tres niveles.

Fuente: Desafío Latam

Llevaremos este diagrama a código, que es bastante intuitivo, ahora que hemos trabajado un poco con herencia. Igualmente es importante destacar que la clase Fitófago tendrá un método particular, el cual, debe retornar el siguiente mensaje: ``${this.nombre} come ${this.tipoComida}, además se mueve ${this._distancia} km diarios``



- **Paso 1:** Crear la clase Animal, la cual tendrá el método “**get nombre**”, para retornar el nombre y “**set nombre**”, para modificar el nombre.

```
class Animal {  
    constructor(nombre) {  
        this._nombre = nombre;  
    }  
    get nombre() {  
        return this._nombre;  
    }  
    set nombre(nombre_nuevo) {  
        this._nombre = nombre_nuevo;  
    }  
}
```

- **Paso 2:** Definir la clase Herbívoro, la cual se extiende de Animal y define el método “**get tipoComida**”, quien retorna el tipo de comida que tendrá un Animal, así como el método “**set tipoComida**” que podrá modificar el tipo de comida del animal.

```
class Herbivoro extends Animal {  
    constructor(nombre, tipoComida) {  
        super(nombre);  
        this._tipoComida = tipoComida;  
    }  
    get tipoComida() {  
        return this._tipoComida;  
    }  
    set tipoComida(comidaNueva) {  
        this._tipoComida = comidaNueva;  
    }  
}
```

- **Paso 3:** Definir la clase Fitófago, la cual define “**get distance**”, permitiendo obtener la distancia que recorre el animal diariamente, al igual que el método “**set distancia**” que permitirá modificar el atributo de distancia. Entonces nos quedará de esta forma.

```
class Fitofago extends Herbivoro {
  constructor(nombre, tipoComida, distancia) {
    super(nombre, tipoComida);
    this._distancia = distancia;
  }
  get distancia() {
    return this._distancia;
  }
  set distancia(distanciaNueva) {
    this._distancia = distanciaNueva;
  }
  infoFitofago() {
    return `${this.nombre} come ${this.tipoComida}, además se mueve  
${this._distancia} km diarios`;
  }
}
```

- **Paso 4:** Acceder a las distintas partes de nuestro código. Si generamos una instancia de Fitofago, con el nombre de conejo, siendo este un animal herbívoro del tipo fitófago, deberíamos ser capaces de obtener tanto el nombre como el tipo de comida, dado que hereda de ambas clases.

```
const conejo = new Fitofago('Pepito', 'Zanahorias 🥕', '2 metros');
```

- **Paso 5:** Una vez definida la instancia, podremos acceder a todas las partes del código, por ejemplo, vamos a mostrar el nombre, tipoComida, distancia e infoFitofago.

```
console.log(conejo.nombre);
console.log(conejo.tipoComida);
console.log(conejo.distancia);
console.log(conejo.infoFitofago());
```

- **Paso 6:** Al ejecutar en la terminal el código anterior con ayuda de Node, nos entregará como resultado, el nombre, lo que come y la distancia, además de una línea con todo concatenado. Es importante que entendamos que el nombre lo obtenemos a partir de la clase **Animal**, luego lo que come, que está definido en la clase **Herbívoro** y finalmente la distancia, que está definida en la clase **Fitofago**, además el resultado de llamar a `infoFitofago()` que está definido también en **Fitofago**.

```
Pepito
Zanahorias 🥕
2 metros
Pepito come Zanahorias 🥕 además salta 2 metros
```

- **Paso 7:** Perfecto, todo funciona bien, veamos otro ejemplo, para este definiremos otra instancia, le llamaremos **"conejo2"**, pero en este caso será instancia de **Herbívoro**. Quedando de esta forma.

```
const conejo2 = new Herbivoro('Roger', 'Lechuga 🥬');
```

- **Paso 8:** Ahora ejecutaremos las mismas instrucciones que antes, pero modificando la instancia, agregando también un nuevo nombre mediante el método `set`, quedando así.

```
console.log(conejo2.nombre);
console.log(conejo2.tipoComida);
console.log(conejo2.distancia);
conejo2.nombre = "PomPom";
console.log(conejo2.nombre);
conejo2.tipoComida = "Maní 🥜"
console.log(conejo2.tipoComida);
```

- **Paso 9:** Al ejecutar en la terminal el código anterior con ayuda de Node, nos entregará como resultado, el nombre, dado que es parte de la clase superior Animal, luego nos entrega lo que come, ya que el método es propio de la instancia de clase. Posteriormente, nos retorna **“undefined”** esto se debe a que Herbívoro no es instancia de Fitofago, por consiguiente al tratar de acceder a algún elemento de una clase de la cual no es instancia, no sabe a que debe acceder. Igualmente, podremos observar cómo accedemos a los métodos set, modificamos el nombre y el tipo de comida en las clases Animal y Herbívoro respectivamente.

```
Roger  
Lechuga 🥬  
undefined  
PomPom  
Maní 🥜
```

Al finalizar este tema, podemos concluir que al implementar herencia entre clases, será posible acceder a los métodos de las clases superiores, es decir las que heredan, desde las clases inferiores y obtener los valores o modificarlos mediante los métodos get y set.

## Ejercicio propuesto (5)

Una selección de fútbol requiere un programa para agrupar a sus jugadores, entrenadores, personal médico, administrativo y de mantenimiento. Por lo que requiere que se construya un programa en ES6 mediante el uso de clases y herencia, que una vez ingresado los datos, muestre cual es la información del jugador (alimentación, si es titular, el número de dorsal, si tiene algún tipo de lesión, el nombre y la edad). Además, mostrar solamente los datos del jugador (nombre y edad). Esto se debe hacer mediante un diagrama de clases entregado por la selección de fútbol, en donde se indica que se deben crear tres clases, una con el nombre de “SeleccionFutbol” como clase padre (principal), luego la clase “Jugadores” como clase hija y finalmente la clase “Arqueros” como clase nieta de la clase principal.

En el diagrama de clases se muestra cuáles serán los atributos de cada clase, así como los métodos. El método `infoArqueto()` deberá mostrar como mensaje: "La alimentación del jugador es ..., ¿es titular?: ..., el número en la camiseta es: ..., lesiones conocidas: ..., nombre: ..., edad: ...".

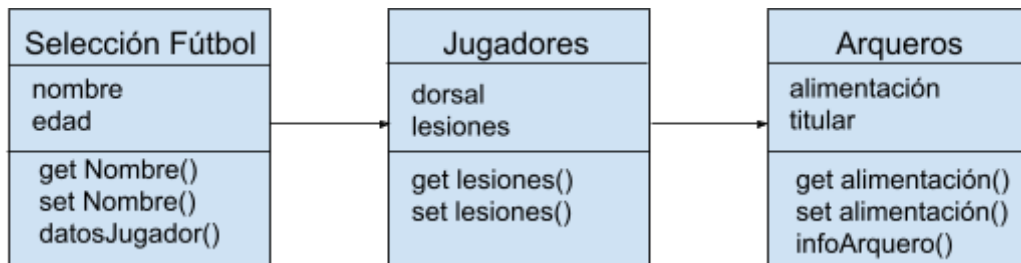


Imagen 15. Diagrama de herencia en tres niveles.

Fuente: Desafío Latam

## Ejercicio guiado: Integrando objetos, herencia y HTML

Crear un programa en ES6 implementando clases y herencia de clases, el cual recibirá la información desde un formulario para registrar el nombre y la raza de un perro cuando se haga clic sobre el botón de registrar.

Por lo que se deben crear dos clases, una clase padre con el nombre de Animal y otra clase hija con el nombre de Perro. Ahora bien, para mostrar la información del perro registrado, se debe hacer un clic sobre el botón de "ver", ubicado en el formulario, como se muestra a continuación:

```

<h1>Registra tu perrito</h1>
<label>Nombre</label>
<input id="nombre" type="text"/>
<label> Raza</label>
<input id="raza" type="text"/>
<button id="registrar">Registrar</button>
<button id="ver">Ver</button>
<div id="data"></div>
<script src="script.js"></script>
    
```

Por lo tanto, para realizar este ejercicio se debe seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos, un index.html y un script.js.
- **Paso 2:** En el index.html debes escribir la estructura básica de un documento HTML incluyendo el formulario indicado en el enunciado del ejercicio, como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>ES6 - Clases</title>
</head>
<body>
  <h1>Registra tu perrito</h1>
  <label>Nombre</label>
  <input id="nombre" type="text"/>
  <label> Raza</label>
  <input id="raza" type="text"/>
  <button id="registrar">Registrar</button>
  <div id="data"></div>
  <script src="script.js"></script>
</body>
</html>
```

- **Paso 3:** En el archivo script.js definir una clase llamada Animal, con un constructor que recibe el parámetro "nombre" y un método get para retornar el nombre. Entonces nuestro código queda de esta forma.

```
class Animal {
  constructor(nombre) {
    this._nombre = nombre;
  }
  get nombre() {
    return this._nombre;
  }
}
```

- **Paso 4:** Crear una clase llamada Perro, la cual hereda de Animal, esta tendrá como atributo "raza" en el constructor al igual que la palabra clave "super" para utilizar ese dato de la clase padre. Luego, se crearán dos métodos, uno con get para obtener la raza y otro con set para modificar la raza. Entonces nuestro código queda de esta forma.

```
class Perro extends Animal {  
  constructor(nombre, raza) {  
    super(nombre);  
    this._raza = raza;  
  }  
  get raza() {  
    return this._raza;  
  }  
  set raza(raza) {  
    this._raza = raza;  
  }  
};
```

- **Paso 5:** Captar los valores del HTML mediante DOM, es decir, primeramente los dos botones mediante los "id" de cada uno de ellos, agregando un escucha a cada botón para poder activar funciones individuales que ejecuten procedimientos separados.

```
let registrar = document.getElementById('registrar');  
registrar.addEventListener('click',observando);
```

- **Paso 6:** Definidos los escuchas y las funciones que se deben ejecutar, trabajamos ahora con ellas. La primera función denominada "registrando", será utilizada para registrar los elementos ingresados por el usuario, es decir, el nombre y la raza. Al igual que instanciar el objeto pasando como argumento al constructor el nombre y a través del setter, el valor correspondiente al atributo "raza".

```
function registrando() {  
  let nombre = document.getElementById("nombre").value;  
  let raza = document.getElementById("raza").value;  
  var perrito = new Perro(nombre);  
  perrito = new Perro(nombre);  
  return perrito;  
}
```

- **Paso 7:** En la segunda función denominada “**observando**”. Se creará una variable para llamar la primera función creada y así tener acceso a la clase y sus métodos. Con esto se puede obtener el valor del nombre y la raza cargados anteriormente al objeto. Luego, se creará un nuevo elemento, en este caso podría ser un “<div>” para poder agregar el mensaje donde se muestre el nombre del perro y la raza. Finalmente, se limpiarán los campos (<inputs>) para dejarlos disponibles para otros datos.

```
function observando() {  
  const perroData = registrando();  
  const nombre = perroData.nombre;  
  const raza = perroData.raza;  
  const data = document.getElementById('data');  
  const p = document.createElement('p');  
  p.innerHTML = `🐶 Nombre: ${nombre} - Raza: ${raza}`;  
  data.appendChild(p);  
  document.getElementById('nombre').value = '';  
  document.getElementById('raza').value = '';  
}
```

- **Paso 8:** Al ejecutar el index.html en el navegador web, haciendo doble clic sobre el archivo, veremos lo siguiente:

## Registra tu perrito

Nombre  Raza

Imagen 16. Ejercicio Herencia Input.  
Fuente: Desafío Latam

- **Paso 9:** Ingrese los dos valores, por ejemplo: “Taty y Pastor Alemán”, luego pulsamos el botón de “**Registrar**” y posteriormente el botón de “**Ver**”, el resultado sería:

## Registra tu perrito

Nombre  Raza

🐶 Nombre: Taty - Raza: Pastor Alemán

Imagen 17. Ejercicio Herencia Resultado.  
Fuente: Desafío Latam



## Ejercicio propuesto (6)

Crear un programa con ES6 mediante clases y herencia, en donde existe una plataforma que maneja distintos tipos de evaluaciones, estas pueden ser un quiz de alternativas, un desafío o un proyecto. Para ello, el usuario utilizará un formulario donde seleccionará el tipo de evaluación, además indicará el nombre, apellido y duración de la evaluación. Todo se debe realizar siguiendo el siguiente diagrama de clases.

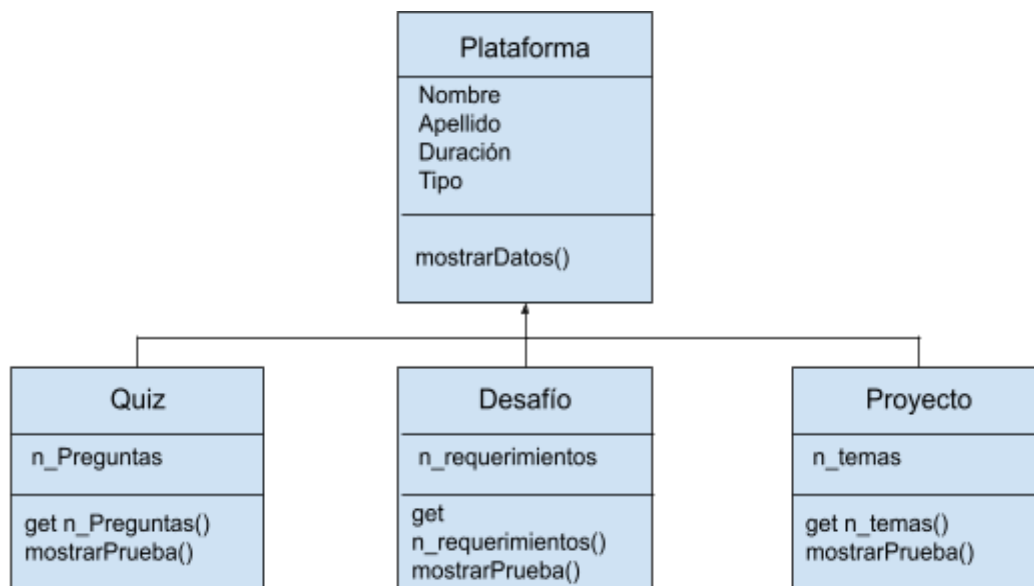


Imagen 18. Diagrama de clases ejercicio cadena prototipos.

Fuente: Desafío Latam

Los métodos de “mostrarPrueba”, deben mostrar todos los datos “tipo, duración”, además de la cantidad de preguntas, requerimientos o temas, todo depende de la selección del usuario. Mientras que el método “mostrarDatos”, sólo debe mostrar los datos del participante “nombre y apellido”. Por otra parte, el formulario a utilizar sería el siguiente:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Desafío</title>
</head>
<body>
  <h2>Tipo de Evaluación a Realizar</h2>
  <form>
    <label for="nombre">Ingrese el nombre del participante: <input
type="text" id="nombre"></label>
    <br>
    <label for="apellido">Ingrese el apellido del participante: <input
type="text" id="apellido"></label>
    <br>
    <label for="varios">Ingrese el número de preguntas, requerimientos
o temas: <input type="number" id="varios"></label>
    <br>
    <label for="duracion">Ingrese el duracion de la evaluación en
minutos: <input type="number" id="duracion"></label>
    <br>
    <label for="tipo">Seleccione el tipo de evaluación:
      <select name="tipo" id="tipo">
        <option value="quiz">Quiz</option>
        <option value="desafio">Desafio</option>
        <option value="proyecto">Proyecto</option>
      </select>
    </label>
    <br>
    <button>Enviar</button>
  </form>
  <div id="resultado"></div>
  <script src="./script.js"></script>
</body>
</html>
```

## Resumen

A lo largo de esta lectura cubrimos:

- Utilizar herencia para la reutilización de componentes en el contexto de la Programación Orientada a Objetos.
- Implementar herencias a partir de prototipos mediante la sintaxis de ES5 y ES6.
- Implementar la cadena de prototipos para lograr distintos niveles de herencia.
- Aplicar múltiples niveles de herencia que permitan reutilizar propiedades y métodos de clases superiores.
- Construir propiedades get y set en ES6 para acceder o modificar los atributos de una clase.
- Combinar objetos, herencia y HTML para realizar ejercicio práctico.

## Solución de los ejercicios propuestos

1. A partir del siguiente diagrama de clases, crea un programa en JavaScript que permite utilizar los atributos de una clase superior en las clases hijas, además de agregar atributos propios para cada una de las clases hijas, instanciando cada clase y mostrando el resultado.

```
class Producto {  
  constructor(nombre,sku,precio,marca) {  
    this.nombre = nombre;  
    this.sku = sku;  
    this.precio = precio;  
    this.marca = marca;  
  }  
}  
  
class Computadores extends Producto {  
  constructor(nombre, sku, precio, marca, procesador){  
    super (nombre, sku, precio, marca);  
    this.procesador = procesador;  
  }  
}  
  
class Linea_Blanca extends Producto {  
  constructor(nombre, sku, precio, marca, puertas){  
    super (nombre, sku, precio, marca);  
    this.puertas = puertas;  
  }  
}  
  
class Zapateria extends Producto {  
  constructor(nombre, sku, precio, marca, talla){  
    super (nombre, sku, precio, marca);  
    this.talla = talla;  
  }  
}  
  
let pc = new Computadores('Acer Nitro', 'DTG56D', '700000',  
  'Acer','AMD7');  
let nevera = new Linea_Blanca('Nevera FrioKing', '878GT3', '500000',  
  'LG','2');
```

```
let zapato = new Zapateria('Runner Team', '24NMGLK', '50000',  
  'Nike', '42');  
  
console.log(computador);  
console.log(nevera);  
console.log(zapato);
```

2. Transforma el siguiente código de ES5 a ES6 implementando herencia de clases.

```
class Curso {  
  constructor(titulo,turno,alumno) {  
    this.titulo = titulo;  
    this.turno = turno;  
    this.alumno = alumno;  
  }  
  inscribirAlumno(){  
    console.log(`El alumno: ${this.alumno}, se ha inscrito en el  
curso: ${this.titulo}, perteneciente al turno: ${this.turno}`);  
  }  
}  
  
class JavaScript extends Curso {  
  constructor(titulo,turno,alumno,nivel,tema) {  
    super (titulo,turno,alumno)  
    this.nivel = nivel;  
    this.tema = tema;  
  }  
  temaSolicitado() {  
    console.log(`El nivel y el tema solicitado fueron: ${this.nivel} y  
${this.tema} respectivamente`);  
  }  
}  
  
var alumno1 = new JavaScript('Programando con JavaScript', 'Nocturno',  
  'Juan', 'Basico', 'Introduccion');  
console.log(alumno1);  
alumno1.temaSolicitado();  
alumno1.inscribirAlumno();
```

3. Una empresa constructora de edificios, necesita un sistema que identifique el tipo de cliente (pequeño, mediano o grande) para poder estimar la cantidad de personal que necesite para la ejecución de la obra civil. Por ello, en este preciso momento requieren que el sistema muestre un mensaje indicando el nombre y tipo de cliente, además que se muestre el valor del código del cliente ingresado. Así mismo, la empresa solicita que el código se realice mediante la sintaxis de ES6. Por otra parte, la empresa facilitó un diagrama de clases para seguir al momento de desarrollar el código en JavaScript, este se debe ejecutar utilizando Node.

```
class Constructora {
  constructor(nombreCliente, tipoCliente){
    this.nombreCliente = nombreCliente;
    this.tipoCliente = tipoCliente;
  }
  mostrarDatos(){
    console.log(`El nombre del cliente es: ${this.nombreCliente} y el
tipo es: ${this.tipoCliente}`);
  }
}

class Cliente extends Constructora {
  constructor(nombreCliente, tipoCliente, codigoCliente){
    super (nombreCliente, tipoCliente);
    this.codigoCliente = codigoCliente;
  }
  mostrarCodigo(){
    console.log(`El código del cliente es: ${this.codigoCliente}`);
  }
}

let cliente = new Cliente('Inmobiliaria XYZ', 'Medio', '546FFGHTF5');
cliente.mostrarCodigo();
cliente.mostrarDatos();
```

4. Realizar un programa para una tienda de instrumentos musicales, la cual necesita ordenar los instrumentos musicales que posee. La tienda solicita crear el programa para ordenar por el tipo de instrumento (cuerda o viento) por los momentos. Por lo que requiere mostrar desde la clase superior “Tienda” la dirección del local comercial, mientras que en la clase de “Instrumento” se debe mostrar en un método el mensaje con el número de serie del instrumento. Para lograr esto, se debe crear una instancia para cada tipo de instrumento pasando como parámetros todos los datos requeridos por las clases superiores. Como se muestra en el diagrama de clases a continuación:

```
class Tienda {
  constructor(direccion) {
    this.direccion = direccion;
  }
  datosTienda(){
    console.log(`La dirección del local es: ${this.direccion}`);
  }
}

class Instrumento extends Tienda {
  constructor(direccion, serial) {
    super(direccion)
    this.serial = serial;
  }
  datosInstrumento(){
    console.log(`El serial del instrumento es es: ${this.serial}`);
  }
}

class Cuerdas extends Instrumento {
  constructor(direccion, serial, numeroCuerdas,) {
    super(direccion, serial)
    this.numeroCuerdas = numeroCuerdas;
  }
}

class Viento extends Instrumento {
  constructor(direccion, serial, material_cosntruc,) {
    super(direccion, serial)
    this.material_cosntruc = material_cosntruc;
  }
}

let guitarra = new Cuerdas('Santiago de Chile', 'B00HQWRTTU', 6);
let trompeta = new Viento('Santiago de Chile', 'XERF45TGFD', 'Metal');
guitarra.datosTienda();
guitarra.datosInstrumento();
trompeta.datosTienda();
trompeta.datosInstrumento();
```

5. Una selección de fútbol requiere un programa para agrupar a sus jugadores, entrenadores, personal médico, administrativo y de mantenimiento. Por lo que requiere que se construya un programa en ES6 mediante el uso de clases y herencia, que una vez ingresado los datos, muestre cual es la información del jugador (alimentación, si es titular, el número de dorsal, si tiene algún tipo de lesión, el nombre y la edad). Además, mostrar solamente los datos del jugador (nombre y edad). Esto se debe hacer mediante un diagrama de clases entregado por la selección de fútbol, en donde se indica que se deben crear tres clases, una con el nombre de "SeleccionFutbol" como clase padre (principal), luego la clase "Jugadores" como clase hija y finalmente la clase "Arqueros" como clase nieta de la clase principal.

```
class seleccionFutbol {
  constructor(nombre, edad) {
    this._nombre = nombre;
    this.edad = edad;
  }
  get nombre(){
    return this._nombre;
  }
  set nombre(nuevo_nombre){
    this._nombre = nuevo_nombre;
  }
  datosJugador(){
    console.log(`El nombre del jugador es: ${this.nombre}, y la edad
es: ${this.edad}`);
  }
}

class Jugadores extends seleccionFutbol {
  constructor(nombre, edad, dorsal, lesiones) {
    super(nombre, edad)
    this.dorsal = dorsal;
    this._lesiones = lesiones;
  }
  get lesiones(){
    return this._lesiones
  }
  set lesiones(nuevas_lesiones){
    this._lesiones = nuevas_lesiones;
  }
}

class Arqueros extends Jugadores {
```



```
    constructor(nombre, edad, dorsal, lesiones, alimentacion, titular)
    {
        super(nombre, edad, dorsal, lesiones)
        this._alimentacion = alimentacion;
        this.titular = titular;
    }
    get alimentacion(){
        return this._alimentacion = alimentacion;
    }
    set alimentacion(n_alimentacion){
        this._alimentacion = n_alimentacion;
    }
    infoArquero(){
        console.log(`La alimentación del jugador es ${this._alimentacion},
        ¿es titular?: ${this.titular}, el número en la camiseta es:
        ${this.dorsal}, lesiones conocidas: ${this.lesiones}, nombre:
        ${this.nombre}, edad: ${this.edad}`);
    }
}

let arquero = new Arqueros('Juan', '35', 3, 'Rotura de meniscos
interno', 'Mayor a 5000 calorías', 'No');
arquero.datosJugador();
arquero.infoArquero();
```

6. Crear un programa con ES6 mediante clases y herencia, en donde existe una plataforma que maneja distintos tipos de evaluaciones, estas pueden ser un quiz de alternativas, un desafío o un proyecto. Para ello, el usuario utilizará un formulario donde seleccionará el tipo de evaluación, además indicará el nombre, apellido y duración de la evaluación. Todo se debe realizar siguiendo el siguiente diagrama de clases.

```
let formulario = document.querySelector('form');

let evaluacion = (event) =>{
    event.preventDefault();

    let nombre = document.getElementById('nombre').value;
    let apellido = document.getElementById('apellido').value;
    let duracion = document.getElementById('duracion').value;
    let varios = document.getElementById('varios').value;
    let tipo = document.getElementById('tipo').value;
    let resultado = document.getElementById('resultado');
```

```
        if (tipo == 'quiz') {
            let quiz = new Quiz(nombre,apellido,duracion,tipo,varios);
            resultado.innerHTML = `${quiz.mostrarDatos()}. <br>
${quiz.mostrarPrueba()}`
        } else if (tipo == 'desafio') {
            let desafio = new Desafio(nombre,apellido,duracion,tipo,varios);
            resultado.innerHTML = `${desafio.mostrarDatos()}. <br>
${desafio.mostrarPrueba()}`
        } else {
            let proyecto = new Proyecto(nombre,apellido,duracion,tipo,varios);
            resultado.innerHTML = `${proyecto.mostrarDatos()}. <br>
${proyecto.mostrarPrueba()}`
        }
    }

formulario.addEventListener('submit',evaluacion);

class Plataforma {
    constructor(nombre,apellido,durancion, tipo) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.durancion = durancion;
        this.tipo = tipo;
    }

    mostrarDatos(){
        return (`Los datos del participante son: ${this.nombre}
${this.apellido}.`);
    }
}

class Quiz extends Plataforma {
    constructor(nombre,apellido, durancion, tipo, n_preguntas) {
        super(nombre,apellido,durancion, tipo);
        this._n_preguntas = n_preguntas;
    }

    get n_preguntas(){
        return this._n_preguntas;
    }

    mostrarPrueba(){
        return (`Los datos de la prueba son: Tipo ${this.tipo}, con una
```

```
duración de ${this.durancion} minutos y con ${this._n_preguntas}
pregunta(s)`);
    }
}

class Desafio extends Plataforma {
    constructor(nombre,apellido, durancion, tipo, n_requerimientos) {
        super(nombre,apellido,durancion, tipo);
        this._n_requerimientos = n_requerimientos;
    }

    get n_requerimientos(){
        return this._n_requerimientos;
    }

    mostrarPrueba(){
        return (`Los datos de la prueba son: Tipo ${this.tipo}, con una
duración de ${this.durancion} minutos y con ${this._n_requerimientos}
requerimiento(s)`);
    }
}

class Proyecto extends Plataforma {
    constructor(nombre,apellido, durancion, tipo, n_temas) {
        super(nombre,apellido,durancion, tipo);
        this._n_temas = n_temas;
    }

    get n_temas(){
        return this._n_temas;
    }

    mostrarPrueba(){
        return (`Los datos de la prueba son: Tipo ${this.tipo}, con una
duración de ${this.durancion} minutos y con ${this._n_temas} tema(s)`);
    }
}
```