UT04.2: Objetos y prototipos

Contenido

1. Introducción

En este apartado vamos a trabajar la creación de **objetos definidos por el usuario**. JavaScript ofrece tres formas de crear los objetos que cumplan con los requisitos de nuestra lógica de negocio:

- **Objetos literales**: Diseñados a partir de la notación de llaves. Ya los hemos estado trabajando en documentos anteriores.
- **Funciones constructoras**: Permite crear objetos a partir de una función constructora de forma similar a un lenguaje orientado a objetos como es Java, aunque basándose en el uso de **prototipos** como veremos a continuación.
- **Clases**: Son una mejora sintáctica sobre el uso de los prototipos, siendo más clara y sencilla de comprender, pero **no** introducen un nuevo mecanismo interno de creación y mantenimiento de objetos.

Algunos de los conceptos importantes que vamos a trabajar son:

- Igualdad entre objetos.
- Fusionado de propiedades.
- Inmutabilidad.

2. Crear objetos en JavaScript

2.1. Objetos dinámicos

A diferencia de otros lenguajes de programación orientados a objetos, JavaScript utiliza objetos **dinámicos**, es decir, podemos alterar la estructura de un objeto, sus propiedades y métodos, en tiempo de ejecución.

Podemos crear un objeto literal denominado *product* con dos propiedades, el número de serie de producto en *serial number* y su nombre en *name*. Una vez creado, las características del lenguaje nos permiten añadir nuevas propiedades al objeto, tan solo tenemos que indicar la variable de la instancia, y a través de la notación punto asigna un nuevo nombre de propiedad con el valor. En el ejemplo hemos creado *price* con el precio del producto.

De la misma forma podemos asociar un método al objeto asignando una función a una propiedad. Como sabemos, la palabra reservada `this` hace referencia al contexto donde se está ejecutando la función, en este caso es el propio objeto, permitiendo acceder al resto de sus propiedades.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
};
product.price = 750;
product.fullName = function () {
  return `${this.serial}: ${this.name}`;
};
$$result.log(product.price); // 750
$$result.log(product.fullName()); // 111-111-111: Portátil
```

2.2. Atajo para crear objetos literales

Tenemos una notación abreviada para crear objetos literales en la que los nombres de las variables son recogidas directamente de las variables utilizadas para asignar sus valores. La notación para los métodos podemos utilizar el nombre de la función el lugar de la palabra reservada `function`.

```
function createProduct(serial, name) {
  return {
    serial,
    name,
    fullName() {
      return `${this.serial}: ${this.name}`;
    },
  };
}
const product = createProduct('111-111-111', 'Portátil');
// Object {serial: 111-111-111, name: Portátil, fullName...}
$$result.log(product);
$$result.log(product.fullName()); // 111-111-111: Portátil
```

2.3. El objeto `Object`

En la referencia de JavaScript dispone de la clase de objeto `Object`. Esta clase dispone de una serie de métodos estáticos que podemos utilizar para manipular otros objetos. A lo largo de documentos vamos a utilizar algunos de sus principales métodos.

Un ejemplo son los métodos `keys()` y `value()` de la clase devuelven un array con los nombres de las propiedades y los valores de las propiedades enumerables del objeto respectivamente.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
  fullName() {
    return `${this.serial}: ${this.name}`;
  },
};
$$result.log(Object.keys(product)); // serial,name,price,fullName
// 111-111-111,Portátil,750,fullName() {...}
$$result.log(Object.values(product));
```

2.4. Igualdad entre objetos

En JavaScript disponemos de tres métodos para comparar valores. Los operadores de igualdad `==` y `===`, y el método estático `Object.is()`. Vamos a estudiar su efecto en el uso de objetos.

El operador `==` ya hemos estudiado que no es seguro, ya que compara valores realizando un casting automático entre ellos, es decir, nos permite comparar un *string* con un *number* y que su valor sea *true* entre otras comparaciones.

```
"12345" == 12345 // true
```

El método `Object.is()` es similar al operador `===` pero más verboso, salvo en algunos casos puntuales en el uso `NaN`.

Si trabajamos con tipos primitivos, una variable nos permite acceder a una posición de memoria donde se encuentra almacenado el valor. Al comparar dos variables, estamos comparando el contenido de sus posiciones de memoria, por lo tanto, con los tres métodos obtenemos que las variables son iguales.

```
let serial1 = "111-111-111";
let serial2 = "111-111-111";

serial1 == serial2; // true
serial1 === serial2; // true
Object.is(serial1, serial2); //true
```

Dirección	Contenido
222200000A	"111-111-111"
222200000B	"111-111-111"
222200000C	
222200000D	
222200000E	
222200000F	

Al trabajar con objetos, las variables acceden a una posición de memoria donde lo que se guarda es la posición de memoria donde está almacenado el contenido del objeto. Al comparar dos variables de tipo objeto, independientemente de su contenido, lo que estamos comparando son las referencias donde se guarda el objeto.

```
let product1 = {
  serial: "111-111-111",
  name: "Portátil"
};
let product2 = {
  serial: "111-111-111",
  name: "Portátil"
};

product1 == product2; // false
product1 === product2; // false
Object.is(product1, product2); //false
```

Dirección	Contenido
222200000A	333300000A
222200000B	444400000A
222200000C	
222200000D	
222200000E	
222200000F	

Si queremos comparar dos objetos diferentes, podemos tener que comparar su contenido, y no solamente las variables en sí, ya que apuntarán a posiciones de memoria diferentes.

```
product1.serial == product2.serial; // true
product1.name == product2.name; // true
```

2.5. Fusionar objetos

El método `assign()` permite copiar todas las propiedades enumerables de un objeto a otro objeto. Hemos creado un objeto *product1* con una serie de propiedades, y un objeto *product2* vacío. Al ejecutar el método sobre *product2* y *product1* estamos copiando las propiedades de *product1* en *product2*. Aunque su contenido sea el mismo, ambas referencias apuntan a posiciones de memoria diferentes, por lo tanto, al compararlas obtenemos *false*.

```
const product1 = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
  fullName() {
    return `${this.serial}: ${this.name}`;
  },
};

const product2 = { };
Object.assign(product2, product1);
// Object {serial: 111-111-111, name: Portátil, price: 750, fullName:
fullName() { ... },}
$$result.log(product2);
$$result.log(product1 === product2); // false
```

Este método nos da la posibilidad de fusionar dos objetos. En el siguiente ejemplo vemos como *product* pasa a tener las propiedades del objeto *price*.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
};
const price = {
  price: 750,
  currency: 'euro',
};

Object.assign(product, price);
// Object {serial: 111-111-111, name: Portátil, price: 12, currency:
euro,}
$$result.log(product);
```

Vamos a rediseñar el ejemplo anterior creando una función que devuelva dos objetos fusionados utilizando el método `Object.assign()`.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
};
const price = {
  price: 750,
  currency: 'euro',
};
function mergePrice(product, price) {
  return Object.assign(product, price);
}
const fullProduct = mergePrice(product, price);
// Object {serial: 111-111-111, name: Portátil, price: 12, currency:
euro,}
$$result.log(fullProduct);
```

El problema que tenemos es que el primer argumento que pasamos al método `Object.assign()` queda también modificado. Los objetos son referencias, y por tanto cualquier modificación realizada en una función permanece al finalizar la ejecución. En el ejemplo *product* mantiene los cambios.

```
// Object {serial: 111-111-111,name: Portátil,}
$$result.log(product);
const fullProduct = mergePrice(product, price);
// Object {serial: 111-111-111,name: Portátil,price: 750,currency: euro,}
$$result.log(product);
```

La solución pasa por invocar `Object.assign()` utilizando como primer argumento un objeto vacío, ya que el método hace uso del objeto `arguments`.

```
function mergePrice(product, price) {
  return Object.assign({}, product, price);
}
```

2.6. Función constructora

Trabajar con objetos literales es una solución cuando no necesitamos muchas instancias similares del objeto. Una **función constructora** nos permite crear objetos cuando es invocada utilizando el operador `new()`. Por convenio, la primera letra del nombre de la función estará en mayúsculas.

En este ejemplo hemos creado una función constructora *Product* que recibe tres argumentos para crear las propiedades, las cuales acceden al contexto `this` para hacerlo. El objeto creado es devuelto de forma implícita al invocar la función con el operador `new()`.

```
function Product(serial, name, price) {
  this.serial = serial;
  this.name = name;
  this.price = price;
  this.fullName = function () {
    return `${this.serial}: ${this.name}`;
  };
}
const product1 = new Product('111-111-111', 'Portátil', 750);
// Object {serial: 111-111-111,name: Portátil,price: 750,currency:
euro,fullname...}
$$result.log(product1);
const product2 = new Product('222-222-222', 'Tablet', 500);
// Object {serial: 222-222-222,name: Tablet,price: 500,currency:
euro,fullname...}
$$result.log(product2);
```

Es importante saber qué es lo que está ejecutando por debajo el intérprete. Al invocar la función el operador `new()` crea un objeto vacío `{ }`, el cual es utilizado como contexto. Al asignar un nombre de propiedad con `this`, esta propiedad se está asignado a este contexto. Por último, el objeto del contexto es devuelto de forma implícita.

El constructor puede ser invocado todas las veces que sea necesario.

```
let product = new Product("111-111-111", "Portátil", 750);
let product = new Product("222-222-222", "Tablet", 500);
```

2.7. Método `Object.create()`

Los objetos en JavaScript son creados en base a **prototipos** como veremos más adelante. Un prototipo es objeto que sirve de plantilla para crear otro objeto. El método estático `Object.create()` permite crear un objeto nuevo, utilizando como plantilla un previo creado anteriormente.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
};
const product2 = Object.create(product);
// Object {serial: 111-111-111, name: Portátil, price: 750}
$$result.log(product2);
```

El primer argumento del método tenemos el **prototipo**. Opcionalmente, el segundo argumento admite un objeto literal con **descriptores de propiedades** para ser también añadidas al objeto. Las propiedades de un objeto también tienen sus propias propiedades como veremos en el siguiente punto, y quedan definidas dentro de un objeto literal. En este ejemplo hemos rediseñado el ejemplo anterior para hacer uso de descriptores de propiedades. El objeto `Object.protype` es un objeto vacío utilizado como prototipo.

```
const product = Object.create(Object.prototype, {
  serial: {
    value: '111-111-111', enumerable: true, writable: true, configurable:
true,
  },
  name: {
    value: 'Portátil', enumerable: true, writable: true, configurable:
true,
  },
  price: {
    value: 750, enumerable: true, writable: true, configurable: true,
  },
});
// Object {serial: 111-111-111, name: Portátil, price: 750}
$$result.log(product);
```

3. Propiedades

3.1. Notación corchete

Una propiedad es un par nombre y valor que permite personalizar el contenido del objeto con el que estamos trabajando. Como estamos acostumbrados, para acceder a una propiedad utilizamos a la notación punto, pero existe una notación corchete que nos permite obtener el valor de una propiedad invocando su nombre a través de un *string* entre corchetes. Esta notación, aunque no es común, es útil cuando el nombre de una propiedad reside en una variable, o cuando queremos acceder a un objeto remoto en formato JSON.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
};
for (const propertyName in product) {
  $$result.log(`${propertyName}: ${product[propertyName]}`);
}
```

3.2. Descriptor de propiedad

El descriptor de una propiedad es un objeto literal que define cómo podemos utilizar una propiedad de un objeto. Como hemos descrito con el método `Object.create()`, el descriptor de propiedad está compuesto por 4 atributos:

- *value*
- *enumarable*
- *writable*
- *configurable*

Siendo *value* el valor que estamos asignando a una propiedad.

El método `Object.getOwnPropertyDescriptor()` nos devuelve el descriptor de una propiedad para un objeto dado.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
};
// Object {value: 111-111-111,writable: true,enumerable:
true,configurable: true,}
$$result.log(Object.getOwnPropertyDescriptor(product, 'serial'));
```

Veamos el significado del resto de propiedades del descriptor.

3.2.1. Atributo *writable*

El atributo *writable* nos indica si podemos modificar el valor de la propiedad, es decir, nos permite hacer que una propiedad sea de solo lectura. El método estático `defineProperty()` nos permite configurar una propiedad para un objeto.

Si configuramos una propiedad de solo lectura, al intentar modificar su valor el intérprete genera una excepción, la cual hemos capturado en un bloque `try`.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
};
Object.defineProperty(product, 'serial', { writable: false });
try {
  product.serial = '222-222-222';
} catch (error) {
  // TypeError: Cannot assign to read only property 'serial' of object
'#'
  $$result.log(error.message);
}
```

Si la propiedad que hacemos de solo lectura es un objeto, lo que no podemos modificar es el objeto, pero si su contenido.

```
const product = {
  serial: '111-111-111',
  name: {
    brand: 'HP',
    model: 'EliteBook',
  },
  price: 750,
};
Object.defineProperty(product, 'name', { writable: false });
product.name.model = 'Pavilion';
$$result.log(product.name.model); // Pavilion
```

Podemos **congelar** un objeto completo para que ninguna de sus propiedades pueda ser modificadas utilizando el método `Object.freeze()`.

```
const product = {
  serial: '111-111-111',
  name: {
    brand: 'HP',
    model: 'EliteBook',
  },
  price: 750,
};
Object.freeze(product.name);
try {
  product.name.model = 'Pavilion';
} catch (error) {
  // TypeError: Cannot assign to read only property 'model' of object '#'
  $$result.log(error.message);
}
```

3.2.2. Atributo *enumerable*

Nos indica si la propiedad forma parte de la lista de propiedades del objeto sobre las que podemos iterar. Que no podamos iterar sobre una propiedad no significa que no podamos acceder a ella de forma explícita. Tampoco la propiedad será enumerada en serializaciones.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
};
Object.defineProperty(product, 'name', { enumerable: false });
// serial: 111-111-111 price: 750
for (const propertyName in product) {
  $$result.log(`${propertyName}: ${product[propertyName]}`);
}
$$result.log(product.name); // Portátil
$$result.log(Object.keys(product)); // serial,price
// {"serial":"111-111-111","price":750}
$$result.log(JSON.stringify(product));
```

3.2.3. Atributo *configurable*

Nos previene para que una propiedad pueda ser modificada posteriormente, tanto a través de la propiedad *enumable* como borrándola utilizando el método `delete()`.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,

};
Object.defineProperty(product, 'name', { configurable: false });
try {
  Object.defineProperty(product, 'name', { enumerable: false });
} catch (error) {
  // TypeError: Cannot redefine property: name
  $$result.log(error.message);
}
try {
  delete (product.name);
} catch (error) {
  // TypeError: Cannot delete property 'name' of #
  $$result.log(error.message);
}
```

3.3. Getters y Setters

Además de las propiedades de valor con las que hemos trabajado hasta ahora, JavaScript dispone de **propiedades de acceso**. Los **Getters** y **Setters** son propiedades enlazadas con una función para obtener o modificar un valor del objeto, pudiendo ser el valor de otra propiedad o una combinación de varias propiedades.

A veces es deseable permitir acceso a una propiedad que retorna un valor calculado dinámicamente en base al valor de otras propiedades, aunque no implica que tengamos que utilizar un método explícitamente. Es lo que denominamos una propiedad *Getter*.

Un *Setter* se usa para ejecutar una función para cambiar el valor de una propiedad, implementando por tanto regla de nuestra lógica de negocio.

Como limitación tenemos que una propiedad *Getter* no admite argumentos, tampoco tiene sentido que lo hiciese, y una propiedad *Setter* solo puede recibir un argumento con el valor a modificar. Una propiedad debe ser de valor o de acceso, pero no puede definirse simultáneamente de ambas.

Aunque se pueden declarar individualmente, el ejemplo muestra una propiedad *fullName* que va a recuperar el valor de las propiedades *brand* y *model*. En base de una cadena de entrada, también es capaz de desestructurarla para recuperar ambas propiedades.

```
const product = {
  serial: '111-111-111',
  name: {
    brand: 'HP',
    model: 'EliteBook',
  },
  price: 750,
};
Object.defineProperty(product, 'fullName', {
  get() {
    return `${this.name.brand} ${this.name.model}`;
  },
  set(value) {
    const name = value.split(' ');
    this.name.brand = name[0];
    this.name.model = name[1];
  },
});
product.fullName = 'HP Pavilion';
$$result.log(product.fullName); // HP Pavilion
$$result.log(product.name.brand); // HP
$$result.log(product.name.model); // Pavilion
```

Podemos embeber las propiedades de acceso en un objeto literal. En el siguiente ejemplo tenemos un objeto que almacenará los números de serie de los últimos productos vendidos, pero solamente muestra el último que se ha vendido en su propiedad *lastest*.

```
const soldProducts = {
  soldProducts: [],
  get lastest() {
    return this.soldProducts[this.soldProducts.length - 1];
  },
  set lastest(product) {
    this.soldProducts.push(product);
  },
};
soldProducts.lastest = '111-111-111';
soldProducts.lastest = '222-222-222';
$$result.log(soldProducts.lastest); // 222-222-222
```

4. Prototipos e interfaces

4.1. ¿Qué es un prototipo?

Un prototipo es una **plantilla en base a la cual vamos a crear nuevos objetos**. Se trata de una instancia de objeto ubicado en memoria. Cuando una función es creada, por debajo el intérprete también crea una propiedad prototipo para la función, asignándole una instancia de un objeto, que por defecto es un objeto vacío.

```
const myFunction = function () { };
$$result.logBold('Prototipo de una función');
$$result.log(myFunction.prototype); // myFunction { }
```

Un objeto literal no tiene prototipo.

```
const product = {
  serial: '111-111-111',
  name: 'Portátil',
  price: 750,
};
$$result.log(product.prototype); // undefined
```

Pero si tienen una **propiedad __proto__** que apunta a la instancia del objeto en base a la cual se ha creado el objeto. Esta propiedad es creada para cada objeto, y en principio se trata de un objeto también vacío.

```
$$result.log(product.__proto__); // Object { }
```

Si un objeto es creado con una función constructora, tiene una propiedad proto que apunta al mismo objeto en memoria que el prototipo de la función.

4.2. Prototipo de función vs prototipo de objeto

Los prototipos de una función y de un objeto funcionan de forma diferente.

Podemos definir el **prototipo de una función** como la **instancia** de un objeto que será convertida en el prototipo para todos los objetos creados utilizando esta función como constructor.

Por el contrario, el **prototipo de un objeto** es la instancia del objeto del que hereda nuestro objeto.

Veamos cómo funcionan ambos en código. Partimos de una función constructora y mostramos la propiedad *prototype* de la función. Vemos que se trata de un objeto vacío.

```
function Product(serial, name) {
  this.serial = serial;
  this.name = name;
}
$$result.log(Product.prototype); // Product { }
```

Creamos una instancia con el constructor y comprobamos la propiedad *proto*. Podemos comprobar que también se trata de un objeto vacío, y si comparamos ambas propiedades vemos que son iguales. La instancia ha sido creada en base al **prototipo de la función**.

```
const product = new Product('111-111-111', 'Portátil');
$$result.log(product.__proto__); // Product { }
$$result.log(Product.prototype === product.__proto__); // true
```

Si mostramos las propiedades de un objeto instanciado veremos todas las propiedades que han sido añadidas desde el constructor. Añadimos una propiedad *price* al objeto prototipo de la función. Todas las instancias creadas con la función constructora heredarán esta propiedad.

```
$$result.log(product); // Product {serial: 111-111-111, name: Portátil}
Product.prototype.price = 750;
$$result.log(product); // Product {serial: 111-111-111,name:
Portátil,price: 750}
```

La propiedad *prototype* de la función y la propiedad *proto* del objeto apuntan a la misma instancia que acabamos de modificar.

```
$$result.log(Product.prototype);
// Product {price: 750,}
$$result.log(product.__proto__);
$$result.log(Product.prototype === product.__proto__); // true
```

Una instancia de un objeto, además de las propiedades añadidas desde el constructor, también tiene acceso a las propiedades de su prototipo almacenado en la propiedad *proto*, al ser el mismo objeto que al que apunta la propiedad *prototype* de la función, cualquier cambio en el objeto se refleja en todas las instancias que hayan sido creadas.

4.3. Cadena de prototipos

Como acabamos de ver podemos tener instancias de objeto con propiedades creadas en la propia instancia o heredadas del prototipo a través de la propiedad *proto*. En el siguiente ejemplo, tenemos las propiedades *serial* y *name* que están en la propia instancia. Debemos tener en cuenta cómo funciona el operador new, primero crea un objeto vacío, para pasar a invocar a la función constructora. Todas las propiedades de la función constructora pasan a ser creadas en la instancia.

En el objeto vacío inicial, el operador `new` ha creado una propiedad *proto*, la cual apunta a la propiedad *prototype* de la función. Cualquier cambio en el prototipo de la función hace que todas las instancias hereden dicho cambio, en este ejemplo la propiedad *price*.

```
function Product(serial, name) {
  this.serial = serial;
  this.name = name;
}
Product.prototype.price = 750;
const product1 = new Product('111-111-111', 'Portátil');
const product2 = new Product('222-222-222', 'Tablet');
$$result.log(product1.price); // 750
$$result.log(product2.price); // 750
```

Vamos a asignar en la propiedad *price* en una de las instancias que hemos creado. Si comprobamos el valor de esta propiedad vemos que *product1* ha modificado su valor, pero *producto2* continua con el valor heredado.

```
product1.price = 1000;
$$result.log(product1.price); // 1000
$$result.log(product2.price); // 750
```

Lo que ha pasado es que en la instancia de *product1* hemos creado una nueva propiedad *price* con el valor asignado. Al mostrarla, el intérprete primero comprueba si la propiedad existe en la instancia, como la encuentra la muestra. Con la instancia *product2* como la propiedad no existe en la instancia, el intérprete busca en el objeto de la propiedad *proto* si existe la propiedad, mostrando el valor heredado del prototipo.

La instancia *product1* todavía mantiene el valor en el prototipo ya que no ha sido modificado, pero debemos acceder a él de forma explícita.

```
$$result.log(product1.__proto__.price); // 750
```

El concepto por el que una instancia tiene enlazada un prototipo en su propiedad *proto* es denominado **cadena de prototipos**, a través de la cual, una instancia puede heredar propiedades de los prototipos de la función de la que ha sido creada. Se denomina cadena porque la instancia a la que apunta la propiedad *proto* tiene también su propia propiedad *proto*, por lo que el intérprete, sino encuentra la propiedad en la instancia, ni en la propiedad *proto*, iría a buscarlo en la propiedad *proto* de la instancia de nuestra propiedad *proto*, generando una cadena de prototipos. Es lo que denominamos **herencia de prototipos**.

Para determinar si una propiedad está ubicada en la instancia o es heredada desde un prototipo, disponemos del método `hasOwnProperty()` heredado de la clase `Object`.

```
$$result.log(product1.hasOwnProperty('price')); // true
$$result.log(product2.hasOwnProperty('price')); // false
```

4.4. Representación gráfica de prototipos

En base al ejemplo anterior, vamos a ver gráficamente cómo funcionan los prototipos.

Al crear una función, automáticamente también se crea una propiedad *prototype* que apunta a un objeto vacío.

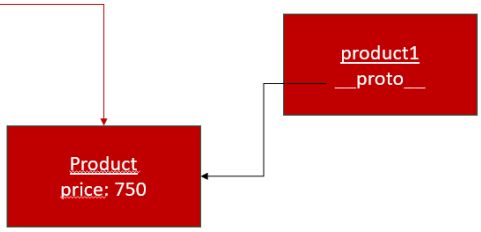
```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
```

A través de la propiedad, podemos modificar el contenido del objeto, por ejemplo añadiendo una propiedad *price*.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;
```

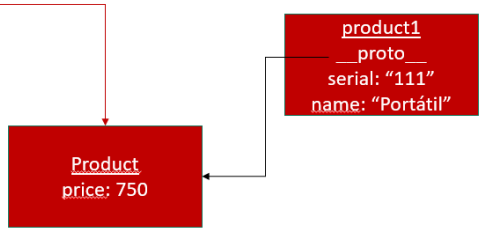
Al instanciar un nuevo objeto con la función constructora, el operador `new` crea una instancia de un objeto con una única propiedad *__proto__* que apuntará a la propiedad *prototype* de la función.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;

let product1 = new Product("111",
"Portátil");
```



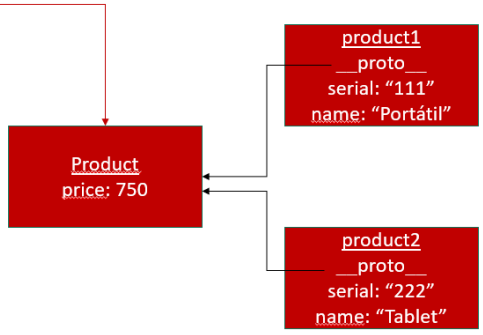
Posteriormente la función es invocada utilizando como contexto la instancia que acaba de crear `new`, es decir, `this` hace referencia a esa instancia vacía. Según se ejecuta la función se irán añadiendo nuevas propiedades a la instancia. Finalmente, la instancia es retornada implícitamente con todas las propiedades creadas en el cuerpo de la función, y la propiedad *__proto__* apuntando a la propiedad *prototype* de la función.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;

let product1 = new Product("111",
"Portátil");
```



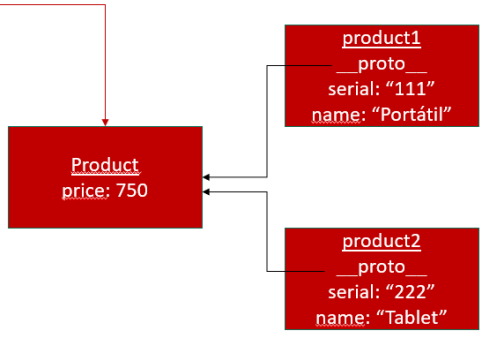
Creamos una segunda instancia. El procedimiento sigue siendo el mismo. Como propiedades de instancia se asignan las propiedades creadas en el cuerpo de la función, y como propiedades heredadas tenemos las que se encuentra en el objeto al que apunta la propiedad *__proto__*.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;

let product1 = new Product("111",
"Portátil");
let product2 = new Product("222",
"Tablet");
```



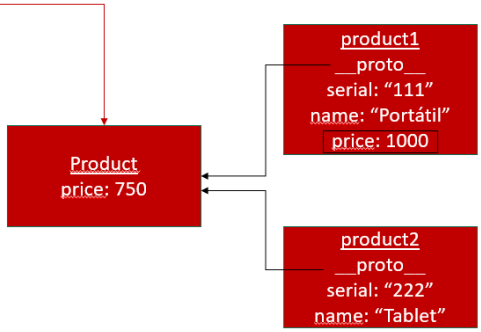
Para acceder a una propiedad seguimos la *cadena de prototipos*. Primero buscamos en las propiedades que realmente tiene la instancia del objeto, si no se encuentra en la instancia, el intérprete busca en el objeto al que apunta la propiedad *__proto__*.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;

let product1 = new Product("111",
"Portátil");
let product2 = new Product("222",
"Tablet");
console.log(product1.price); // 750
```



Al hacer una asignación explicita sobre una propiedad de la instancia, está propiedad es creada dentro del objeto de la propia instancia. Las propiedades de la instancia sobrescriben las del prototipo, ya que el intérprete cuando necesita acceder a una propiedad determinada primero busca dentro de la instancia, si la encuentra, no tiene que seguir buscando.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;

let product1 = new Product("111",
"Portátil");
let product2 = new Product("222",
"Tablet");
product1.price = 1000;
```



4.5. Cambiando prototipo de funciones

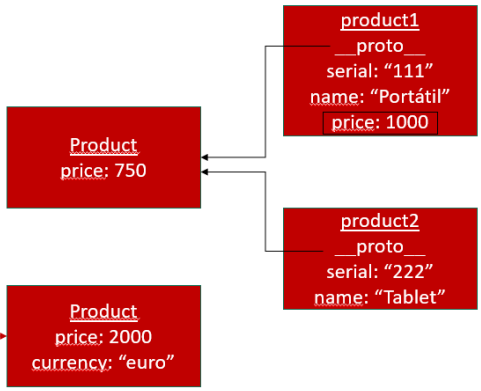
Podría darse el caso de que necesitemos cambiar el objeto al que apunta la propiedad *prototype* de una función, habiendo instanciado ya objetos con la función. En ese caso, los objetos instanciados no cambiar de prototipo, su propiedad __proto__ sigue apuntando al antiguo prototipo. Como vemos en el ejemplo, *product1* y *product2* se han instanciado con un prototipo determinado. Aunque cambiemos el prototipo de la función, estas instancias mantienen el valor de la propiedad *price*.

Al instanciar un nuevo objeto *product3* después del cambio del prototipo, este objeto si que su propiedad *__proto__* apuntará al nuevo prototipo, por eso *price* muestra un valor de 2000.

```
function Product(serial, name) {
  this.serial = serial;
  this.name = name;
}
Product.prototype.price = 750;
const product1 = new Product('111-111-111', 'Portátil');
const product2 = new Product('222-222-222', 'Tablet');

Product.prototype = { price: 2000, currency: 'euro' };
const product3 = new Product('333-333-333', 'Smartphone');
// Object {price: 2000, currency: euro,}
$$result.logBold('Nuevo prototipo');
$$result.log(Product.prototype);
$$result.logBold('Antiguas instancias no cambian');
$$result.log(product1.price); // 750
$$result.log(product2.price); // 750
$$result.logBold('Nuevas instancias apuntan al nuevo prototipo');
$$result.log(product3.price); // 2000
```

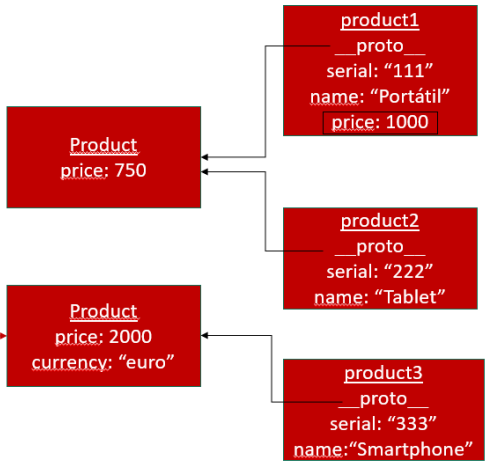
En la siguiente imagen podemos ver gráficamente como queda la situación con el cambio de prototipo. Las instancias *product1* y *product2* siguen apuntando al antiguo prototipo.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;

let product1 = new Product("111",
"Portátil");
let product2 = new Product("222",
"Tablet");
product1.price = 1000;
Product.prototype =
{price: 2000, currency: "euro"}
```



Al instanciar *product3*, su propiedad *__proto__* pasa a apuntar a al objeto de la propiedad *prototype* de la función, por lo que tiene acceso a la actualización de la propiedad *price*.

```
function Product(serial, name){
   this.serial = serial;
   this.name = name;
   prototype
}
Product.prototype.price = 750;

let product1 = new Product("111",
"Portátil");
let product2 = new Product("222",
"Tablet");
product1.price = 1000;
Product.prototype =
{price: 2000, currency: "euro"}
let product3 = new Product("333",
"Smartphone");
```



4.6. Herencia entre objetos

La herencia entre objetos en JavaScript está basada en la cadena de prototipos. El intérprete sigue la cadena de prototipos en búsqueda de una propiedad, hasta que se encuentra un valor de `null`. En el siguiente ejemplo vemos representada esta cadena de prototipos. La propiedad *__proto__* de la instancia apunta a la propiedad *prototype* de la función. A su vez, esta instancia también tiene una propiedad *__proto__* que en este caso apunta a un objeto `Object`. Por último, el objeto `Object` tiene una propiedad *__proto__*, pero en esta ocasión apunta a `null`, con lo que la cadena de prototipos finaliza.

```
function Product(serial, name) {
  this.serial = serial;
  this.name = name;
}
Product.prototype.price = 750;
const product1 = new Product('111-111-111', 'Portátil');

// Product {price: 750,}
$$result.log(product1.__proto__);
// Object { }
$$result.log(product1.__proto__.__proto__);
// null
$$result.log(product1.__proto__.__proto__.__proto__);
```

4.6.1. Tipo de objeto

La propiedad *__proto__* determina el **tipo de objeto** con el que estamos trabajando. Podemos utilizar el operador `instanceof` para saber de qué tipo es una instancia. Este operador compara una instancia con una función. En el ejemplo estamos comparando *product1* con *Product* y nos devuelve *true*.

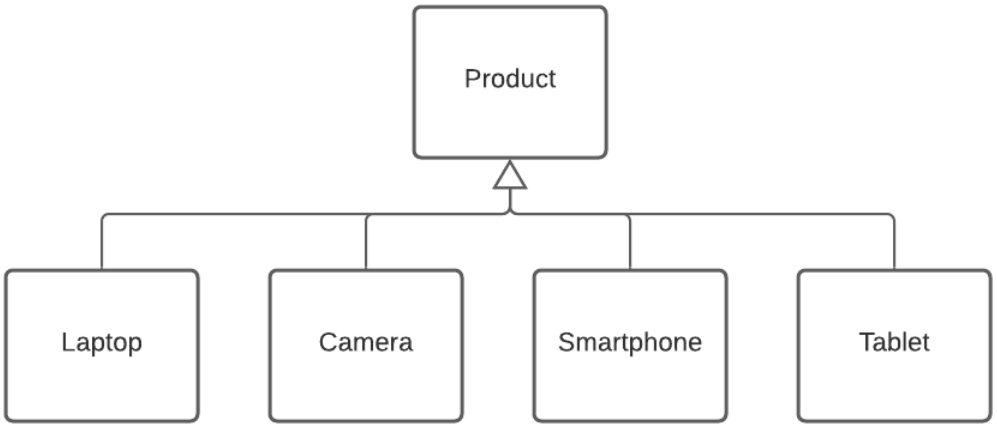
La instancia también hereda de un objeto `Object`, ya que la propiedad __proto__ del prototipo de la instancia es un `Object`, por tanto, el operador `instanceof` devuelve *true* al compararla con `Object`.

Por último, qué es lo que realmente está comparando el operador `instaceof`. El operador recibe como primer operando una instancia, de la cual sigue su cadena de prototipos comparándolos con el prototipo de la función que se pasa como segundo operando. Como vemos en el ejemplo, comparamos la propiedad *__proto__* de *product1* con la propiedad *prototype* de la función *Product* que es la misma comparación que realiza `instanceof` de forma abreviada.

```
$$result.log(product1 instanceof Product); // true
$$result.log(product1 instanceof Object); // true
$$result.log(product1.__proto__ === Product.prototype); // true
```

5. Implementación de la herencia

Hasta ahora estamos trabajando con instancias de objetos *Product*, pero en nuestra tienda tenemos cuatro tipos de productos, *Laptop*, *Camera*, *SmartPhone* y *Tablet*. Todos los tipos de productos comparten propiedades como son un número de serie, el precio, la marca y el modelo, pero al ser productos diferentes, cada uno tiene sus propias características. El siguiente diagrama muestra la estructura UML de herencia de clases.



Creamos la función constructora de la clase *Product* con las propiedades destacadas del enunciado. Además, añadimos un método *toString()*.

```
function Product(serial, brand, model, price) {
  this.serial = serial;
  this.brand = brand;
  this.model = model;
  this.price = price;
  this.toString = function () {
    return `(${this.serial}) ${this.brand} ${this.model}`;
  };
}
```

Creamos una clase *Laptop* con las propiedades del tipo de microprocesador, memoria, y tamaño de disco duro en gigas.

```
function Laptop(micro, memory, hd){
  this.micro = micro;
  this.memory = memory;
  this.hd = hd;
}
```

Lógicamente necesitamos una instancia que incluya las propiedades de ambas clases, por lo que *Laptop* debe heredar de *Product*. Para ello debemos hacer que la propiedad *prototype* de *Laptop* apunte a un objeto cuyo prototipo sea el prototipo de *Product*. Esto lo hacemos mediante el método `Object.create()` pasando como argumento *Product.prototype*.

```
Laptop.prototype = Object.create(Product.prototype);
```

Tenemos añadir una propiedad al recién creado prototipo de la función *Laptop* indicando quien es la función constructora.

```
Laptop.prototype.constructor = Laptop;
```

Instanciamos un objeto *Laptop* y verificamos la cadena de prototipos para comprobar que estamos realizando la herencia.

```
// (111-111-111) HP EliteBook
const product = new Product('111-111-111', 'HP', 'EliteBook', 1200);
$$result.log(product.toString());
const laptop = new Laptop('111-111-111', 'HP', 'EliteBook', 1200, 'i7',
16, 1024);
$$result.log(laptop.toString());
```

Si nos fijamos, la instancia todavía no tiene añadidas las propiedades de la clase *Product*, ya que es necesario invocar al constructor. Modificamos la clase *Laptop* para llamar al constructor, pero tenemos que utilizar como contexto el objeto que estamos creando en este momento. Hacemos una invocación del constructo mediante el método `call()`, pasándole como argumento el contexto del objeto actual `this` y los argumentos que necesite el constructor para instanciar el nuevo objeto. Es necesario añadir a la lista de parámetros de *Laptop* los que tenemos en *Product*.

```
function Laptop(serial, brand, model, price, micro, memory, hd){
  Product.call (this, serial, brand, model, price);
  this.micro = micro;
  this.memory = memory;
  this.hd = hd;
}
Laptop.prototype = Object.create(Product.prototype);
Laptop.prototype.constructor = Laptop;
let laptop = new Laptop (
"111-111-111", "HP", "EliteBook", 1200, "i7", 16, 1024);
```

El objeto ha quedado definido correctamente con las propiedades de ambas clases. Pasamos a sobrescribir el método *toString()*.

```
this.toString = function(){
  return "(" + this.serial + ") " + this.brand + " " + this.model +
    ". " + this.micro + " " + this.memory + "GB" + " " + this.hd + "GB."
}
```

Podemos resumir los pasos para implementar la herencia entre objetos en JavaScript a:

1. Asignar al prototipo de la subclase un objeto cuyo prototipo sea el prototipo de la superclase.
   ```
   Laptop.prototype = Object.create(Product.prototype);
   ```
2. Al prototipo de la subclase le añadimos un la propiedad *constructor* que apuntara a la función constructora de la subclase.
   ```
   Laptop.prototype.constructor = Laptop;
   ```
3. Por último, como primera acción en la función constructora de la subclase, debemos invocar el constructor de la superclase mediante el método `call()` utilizando el contexto actual.
   ```
   Product.call (this, serial, brand, model, price);
   ```