# UT04.3: Objetos en ES5

## Contenido

1. Introducción ..... 2
2. Detalle del modelo de objetos ..... 2
2.1.1. Ejemplo de implementación de herencia ..... 3
2.1.2. Constructores sin parámetros y prototipos ..... 3
2.1.3. Propiedades en prototipos ..... 5
2.1.4. Parámetros en constructores ..... 6
2.1.5. Identificadores secuenciales ..... 8
2.1.6. Herencia múltiple ..... 8
2.1.7. Cadena de prototipos ..... 10
3. Ámbito de los miembros de un objeto ..... 10
3.1. Campos privados ..... 10
3.2. Métodos privados ..... 11
3.3. Métodos públicos ..... 12
3.4. Miembros estáticos ..... 12
3.5. Ejercicio ámbito de objetos ..... 13
3.5.1. Solución ..... 13
3.6. Invocación de métodos de la clase padre ..... 16
3.6.1. Métodos en el prototipo ..... 16
3.6.2. Métodos con campos privados ..... 16
4. Objetos `Error` ..... 19
4.1. Lanzar error genérico ..... 19
4.2. Tipos de errores ..... 19
4.3. Errores personalizados ..... 20
4.4. Jerarquía de excepciones ..... 22
4.4.1. Ejercicio ..... 24
4.4.2. Solución ..... 24
5. Ejercicios productos ..... 25
5.1. Solución ..... 26
5.1.1. Excepciones ..... 26
5.1.2. Implementación de *Product* ..... 27
5.1.3. Implementación de *Laptop* ..... 30
5.1.4. Resto de constructores ..... 33

# 1. Introducción

Los objetos en JavaScript están implementados en base a prototipos. Aunque existen sintaxis más amigable en versiones superiores que veremos en documentos posteriores, la versión ES5 establece la estructura real de cómo construimos objetos en JavaScript. En este documento vamos a tratar el uso de objetos JavaScript utilizando código de la versión ES5.

# 2. Detalle del modelo de objetos

Vamos a revisar el **modelo de objetos** de la versión ES5 en base al documento de MDN https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Details_of_the_Object_Model en el que se hace una comparativa entre leguajes basados en clases vs. basados en prototipos.

JavaScript es un lenguaje basado en objetos en lugar de estar basado en clases, es decir, utiliza por debajo prototipos para implementar la jerarquía de objetos, aunque resulta poco evidente cómo se realiza.

Los **lenguajes basados en clases** como puede ser **Java**, utilizan la clase para definir las propiedades para caracterizar un conjunto de objeto, por ejemplo podríamos tener una clase *Employee* para representar los empleados de una empresa. Una **instancia** es uno de los miembros de la clase, por ejemplo, podríamos tener el empleado "Pablo" y asociarle un número de empleado y un salario entre otros.

Los **lenguajes basados en prototipos** no realizan esta distinción, ya que utilizan otro objeto prototipo que servirá de plantilla en la creación de nuevos objetos. Cualquier objeto puede servir de prototipo para otro objeto.

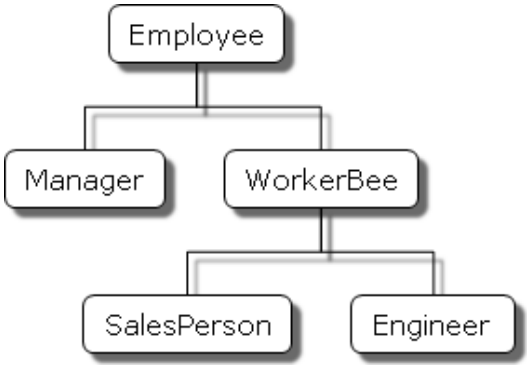
JavaScript utiliza una **función constructora o constructor** para crear objetos, pero no tiene una definición separada de ellos. La función se utiliza como conjunto inicial de propiedades y valores. Como hemos visto anteriormente, el constructor tiene asociado un objeto prototipo el cual es asignado a cada uno de los objetos creadas con el constructor.

En la siguiente tabla podemos ver una comparativa entre lenguajes basados en clases, y los lenguajes basados en prototipos.

| Basado en clases (Java) | Basado en prototipos (JavaScript) |
|---|---|
| La clase y la instancia son diferentes | Todos los objetos son instancias |
| Define una clase como plantilla y utiliza el constructor para instanciar objetos. | Define y crea un conjunto de objetos con funciones constructoras. |
| Se crea un objeto con el operador `new`. | Igual. |
| Se construye una jerarquía de objetos utilizando la definición de las clases para definir subclases de clases existentes. | Se construye una jerarquía de objetos mediante la asignación de un objeto como el prototipo asociado a un constructor. |
| Se heredan propiedades siguiendo la cadena de clases. | Se heredan propiedades siguiendo la cadena de prototipos. |
| No se pueden añadir propiedades dinámicamente en tiempo de ejecución. | Se pueden añadir y quitar propiedades dinámicamente a objetos específicos o a un conjunto de objetos. |

## 2.1.1. Ejemplo de implementación de herencia

Vamos a implementar un ejemplo de herencia basado en una estructura de objetos para gestionar los empleados de una empresa. La siguiente imagen muestra la estructura de objetos que vamos a tener, además, vamos a describir cada uno de los tipos de objeto.




- **Employee**: tiene las propiedades *name* (cuyo valor por defecto es un string vacío) y *dept* (cuyo valor por defecto es "general").
- **Manager**: basado en *Employee*. Añade la propiedad *reports* (cuyo valor por defecto es un array vacío, en la que se pretende almacenar un array de objetos *Employee* como su valor).
- **WorkerBee**: basado en *Employee*. Añade la propiedad *projects* (cuyo valor por defecto es un *array* vacío).
- **SalesPerson**: basado en *WorkerBee*. Añade la propiedad *quota* (cuyo valor por defecto es 100). También reemplaza la propiedad *dept* con el valor "sales".
- **Engineer**: se basa en *WorkerBee*. Añade la propiedad *machine* (cuyo valor por defecto es un *string* vacío) y también reemplaza la propiedad *dept* con el valor "engineering".

Vamos a realizar la implementación de esta estructura por pasos, aunque modificando algunos de los aspectos del documento original.

## 2.1.2. Constructores sin parámetros y prototipos

Pasamos a definir los **constructores sin parámetros y las propiedades utilizadas en sus prototipos para implementar la estructura**. Una idea que debemos seguir a la hora de implementar un objeto es que **cuantas más propiedades heredemos del prototipo**, más rápido ejecutaremos el constructor y menos memoria consumiremos en busca de mejorar la eficiencia de nuestro código.

Definimos el constructor para *Employee*. En lugar de añadir la propiedad *dept* en el cuerpo de la función, lo asociamos al prototipo para que todos los objetos derivados de *Employee* hereden la propiedad. También asociamos la propiedad *constructor* al prototipo con la función constructora.

```
//Constructor Employee
function Employee () {
  this.name = "";
}
Employee.prototype.constructor = Employee;
//Propiedad heredad por todos los objetos Employee, así como todos los ob
jetos que hereden de Employee.
Employee.prototype.dept="general";
```

Los tipos *Manager* y *WorkerBee* heredan de *Employee* asociando su prototipo con un objeto creado a partir del prototipo de *Employee*.

```
//Constructor Manager
function Manager () {
  this.reports = [];
}
//Manager hereda del objeto Employee sus propiedades
Manager.prototype = Object.create(Employee.prototype);
Manager.prototype.constructor = Manager;

//Constructor WorkerBee
function WorkerBee () {
  this.projects = [];
}
//WorkerBee hereda del objeto Employee sus propiedades
WorkerBee.prototype = Object.create(Employee.prototype);
WorkerBee.prototype.constructor = WorkerBee;
```

Las propiedades *reports* y *projects* han sido implementadas en el cuerpo del constructor, en lugar de ser heredadas, esto es debido a que como los *arrays* se tratan de referencias en memoria, todos los objetos creados a partir del constructor compartirían el *array*, en lugar de tener cada uno su propio *array*.

Para los tipos *SalesPerson* y *Engineer* heredan de *WorkerBee*. En cada uno de sus prototipos hemos añadido la propiedad *dept* para que sea heredada, y no tiene por qué cambiar a lo largo de la vida del objeto. De momento *quota* y *machine* los hemos mantenido en el cuerpo del constructor porque serán personalizada por cada usuario posteriormente.

```
//Constructor SalesPerson
function SalesPerson () {
  this.quota = 100;
}
//SalesPerson hereda del objeto WorkerBee sus propiedades
SalesPerson.prototype = Object.create(WorkerBee.prototype);
SalesPerson.prototype.constructor = SalesPerson;
SalesPerson.prototype.dept = "sales";

//Constructor Engineer
function Engineer () {
   this.machine = "";
}
//Engineer hereda del objeto WorkerBee sus propiedades
Engineer.prototype = Object.create(WorkerBee.prototype);
Engineer.prototype.constructor = Engineer;
Engineer.prototype.dept = "engineering";
```

Para finalizar, la función de testeo muestra las propiedades de dos objetos *Engineer* y *SalesPerson*.

```
function testExample() {
  $$result.clear();
  const engineer = new Engineer();
  // Engineer {machine: ,constructor: Engineer,dept: engineering,}
  $$result.log(engineer, 'Engineer');
  // SalesPerson {constructor: SalesPerson,dept: sales,quota: 100,}
  const salesPerson = new SalesPerson();
  $$result.log(salesPerson, 'SalesPerson');
}
```

### 2.1.3. Propiedades en prototipos

Queremos que añadir una propiedad *specialty* para todos los objetos. Los objetos *Engineer*, esta propiedad tiene que tener valor de "code" por defecto, mientras que el resto tiene que tener "none".

En *Employee* añadimos la propiedad en el prototipo con el valor "none" para que sea heredada por cada objeto.

```
function Employee () {
  this.name = "";
}
Employee.prototype.constructor = Employee;
//Propiedad heredad por todos los objetos Employee, así como todos los ob
jetos que hereden de Employee.
Employee.prototype.dept="general";
Employee.prototype.specialty="none";
```

En *Engineer* sobrescribimos la propiedad con el valor "code".

```
//Constructor Engineer
function Engineer () {
  this.machine = "";
}
//Engineer hereda del objeto WorkerBee sus propiedades
Engineer.prototype = Object.create(WorkerBee.prototype);
Engineer.prototype.constructor = Engineer;
Engineer.prototype.dept = "engineering";
Engineer.prototype.specialty="code";
```

Con la misma función de testeo anterior vemos como cada tipo de objeto hereda la propiedad correcta.

```
function testExample() {
  $$result.clear();
  const engineer = new Engineer();
  // Engineer {machine: ,constructor: Engineer,dept:
engineering,specialty: code,}
  $$result.log(engineer, 'Engineer');
  // SalesPerson {constructor: SalesPerson,dept: sales,quota:
100,specialty: none,}
  const salesPerson = new SalesPerson();
  $$result.log(salesPerson, 'SalesPerson');
}
```

### 2.1.4. Parámetros en constructores

Vamos a añadir **parámetros en la firma de los constructores** para personalizar la creación de objetos. En los parámetros utilizamos valores por defecto para evitar tener argumentos indefinidos.

```
//Constructor Employee
function Employee (name = "") {
  this.name = name;
}
Employee.prototype.constructor = Employee;
//Propiedad heredad por todos los objetos Employee, así como todos los ob
jetos que hereden de Employee.
Employee.prototype.dept="general";
Employee.prototype.specialty="none";
```

Rediseñamos *Manager* y *WorkerBee* para definir el parámetro *name*, el cual es utilizado para hacer la invocación al superconstructor *Employee*, y admitir un *array* para definir la propiedad local.

```
//Constructor Manager
function Manager (name, reps = []) {
  //Invocamos al superconstructor Employee con call y pasandole como cont
exto this y los argumentos que espera.
  Employee.call(this, name);
  this.reports = reps;
}
//Manager hereda del objeto Employee sus propiedades
Manager.prototype = Object.create(Employee.prototype);
Manager.prototype.constructor = Manager;

//Constructor WorkerBee
function WorkerBee (name, projs = []) {
  //Invocamos al superconstructor Employee con call y pasandole como cont
exto this y los argumentos que espera.
  Employee.call(this, name);
  this.projects = projs;
}
//WorkerBee hereda del objeto Employee sus propiedades
WorkerBee.prototype = Object.create(Employee.prototype);
WorkerBee.prototype.constructor = WorkerBee;
```

En *SalesPerson* y *Engineer* recibimos todos los argumentos para invocar a los superconstructores, y asignamos a las propiedades locales valores por defecto desde los parámetros.

```
//Constructor SalesPerson
function SalesPerson (name, projs, q = 100) {
  WorkerBee.call(this, name, projs);
  this.quota = q;
}
//SalesPerson hereda del objeto WorkerBee sus propiedades
SalesPerson.prototype = Object.create(WorkerBee.prototype);
SalesPerson.prototype.constructor = SalesPerson;
SalesPerson.prototype.dept = "sales";

//Constructor Engineer
function Engineer (name, projs, mach = "") {
  WorkerBee.call(this, name, projs);
  this.machine = mach;
}
//Engineer hereda del objeto WorkerBee sus propiedades
Engineer.prototype = Object.create(WorkerBee.prototype);
Engineer.prototype.constructor = Engineer;
Engineer.prototype.dept = "engineering";
Engineer.prototype.specialty="code";
```

Por último, la creamos varios objetos de prueba en la función de testeo.

```
function testExample() {
  $$result.clear();
  const engineer = new Engineer('engineer', ['project1', 'project2',
'project3'], 'Machine1');
  // Engineer {name: engineer,projects:
project1,project2,project3,machine: Machine1,constructor: Engineer,dept:
engineering,specialty: code,}
  $$result.log(engineer, 'Engineer');
  // SalesPerson {name: salesPerson,projects:
project4,project5,project6,quota: 2500,constructor: SalesPerson,dept:
sales,specialty: none,}
  const salesPerson = new SalesPerson('salesPerson', ['project4',
'project5', 'project6'], 2500);
  $$result.log(salesPerson, 'SalesPerson');
}
```

### 2.1.5. Identificadores secuenciales

Todo empleado debe disponer de una propiedad *id* que almacene su número de empleado. Intenta resolverlo antes de ver la solución.

Utilizamos una función IIFE para devolver la función constructora *Employee*, dentro de la estructura IIFE creamos una variable *counter* para generar los identificadores. Al devolver la función constructora la variable *counter* se mantendrá en el **closure** de la función.

```
// IIFE de la función constructora Employee.
const Employee = (function () {
  let counter = 0;

  // Constructor Employee
  function Employee(name = '') {
    this.name = name;
    if (name) {
      this.id = ++counter;
    }
  }
  return Employee;
}());
Employee.prototype.constructor = Employee;
// Propiedad heredad por todos los objetos Employee, así como todos los
objetos que hereden de Employee.
Employee.prototype.dept = 'general';
Employee.prototype.specialty = 'none';
```

### 2.1.6. Herencia múltiple

En principio, JavaScript no admite **herencia múltiple** ya que cada objeto solo dispone de una única propiedad *__proto__*, por lo que solo pueden heredar de un único prototipo de forma directa, y el operador `instanceof` solo atenderá a un único tipo de objeto.

Lo que si que podemos hacer es simular la herencia múltiple añadiendo propiedades de forma local a un objeto invocando un segundo constructor. Debemos recordar que el operador `new` crea un objeto vacío con tan solo una propiedad *__proto__* que apuntará al objeto de la propiedad *prototype* de la función. Todas las propiedades añadidas en el cuerpo de constructor son creadas localmente en el objeto. Podemos invocar un segundo constructor, pasando como contexto el objeto actual, para añadir el número de propiedades del segundo constructor. El único problema que tendríamos es que las propiedades heredadas del segundo objeto no estarían accesibles.

Creamos un constructor *Hobbyist* que permite crear un hobby para los objetos *Engineer*. En el prototipo hemos añadido una propiedad *equipment* con el material que se necesita para ese deporte.

```
//Constructor para implementar herencia múltiple.
function Hobbyist (hobby = "scuba") {
  this.hobby = hobby;
}
//El problema de la herencia múltiple, es que al solo heredar de un proto
type, la siguiente propiedad no podrá ser heredada por los objetos Engine
er.
Hobbyist.prototype.equipment = ["mask", "fins", "regulator"];
```

Modificamos *Engineer* para invocar dos superconstructores, *WorkerBee* manteniendo la cadena de prototipos, y *Hobbyist* para añadir las nuevas propiedades. El objeto creado mantiene las propiedades locales de *Hobbyist*, pero no puede heredar *equipment* por no estar relacionado con este prototipo.

```
//Constructor Engineer
function Engineer (name, projs, mach = "", hobby) {
  WorkerBee.call(this, name, projs);
  //Al invocar un segundo constructor, estamos implementando herencia múl
tiple.
  Hobbyist.call(this, hobby);
  this.machine = mach;
}
```

Una solución sería asignar todas las propiedades del prototipo de forma local al objeto contexto mediante el método `Object.assign()` o recorriendo sus propiedades con un bucle.

```
//Constructor Engineer
function Engineer (name, projs, mach = "", hobby) {
  WorkerBee.call(this, name, projs);
  //Al invocar un segundo constructor, estamos implementando herencia múl
tiple.
  Hobbyist.call(this, hobby);
  Object.assign(this, Hobbyist.prototype);
  this.machine = mach;
}
```

### 2.1.7. Cadena de prototipos

En el siguiente ejemplo vemos la cadena de prototipos de un objeto *Enginieer*. El intérprete buscar una propiedad siguiendo la cadena de prototipos referenciados en la propiedad *__proto__*, hasta que encuentra un prototipo asignado a `null`. El prototipo de un `Object` es un objeto sin propiedad __proto__, por tanto, el operador no lo reconoce como tipo `Object`.

```
function testExample() {
  $$result.clear();
  const engineer = new Engineer('engineer', ['project1', 'project2',
'project3'], 'Machine1');
  $$result.log(engineer instanceof Engineer); // true
  $$result.log(engineer.__proto__ == Engineer.prototype); // true
  $$result.log(engineer instanceof WorkerBee); // true
  $$result.log(engineer.__proto__.__proto__ == WorkerBee.prototype); //
true
  $$result.log(engineer instanceof Employee); // true
  $$result.log(engineer.__proto__.__proto__.__proto__ ==
Employee.prototype); // true
  $$result.log(engineer instanceof Object); // true
  $$result.log(engineer.__proto__.__proto__.__proto__.__proto__ ==
Object.prototype); // true
  // El prototipo de un Object es un objeto sin propiedad __proto__, por
tanto el operador no lo reconoce como tipo Object.
  $$result.log(engineer.__proto__.__proto__.__proto__.__proto__
instanceof Object); // false
  $$result.log(engineer.__proto__.__proto__.__proto__.__proto__.__proto__
== null); // true
}
```

## 3. Ámbito de los miembros de un objeto

### 3.1. Campos privados

Hasta ahora todas las propiedades y métodos de los objetos con los que hemos tratado tienen **ámbito público**, es decir, desde fuera del objeto podemos acceder a los métodos o modificar los valores de las propiedades sin atender a la lógica de negocio de nuestra aplicación. Las propiedades de acceso permiten relacionar funciones con propiedades, pero no podemos vincularlas simultáneamente con las propiedades de datos.

Para crear **campos privados** en un objeto que sigan la lógica de negocio de nuestra aplicación, debemos utilizar dos recursos conjuntamente, los *closure* y las *propiedades de acceso*.

Partimos del constructor *Employee* del punto anterior, junto con el *closure* para generar el identificador del número de empleado. Esta propiedad debe ser inmutable a lo largo de la vida del objeto. Esta característica la podemos implementar modificando el atributo *writable* del *descriptor de la propiedad*, o implementando un campo privado como veremos a continuación.

Los campos privados se implementan mediante variables, definimos *_name* y *_id* una por cada propiedad. El nombre comienza con un guion bajo por convenio, no porque sea necesario. A continuación, definimos propiedades de acceso que permitan alterar el contenido de estas variables.

```
const Employee = (function () {
  let counter = 0;

  // Constructor Employee
  function Employee(name = '') {
    // Campos privados
    let _name = name;
    let _id;
    if (name) {
      _id = ++counter;
    }

    // Propiedades de acceso a los atributos privados
    Object.defineProperty(this, 'id', {
      get() {
        return _id;
      },
    });
    Object.defineProperty(this, 'name', {
      get() {
        return _name;
      },
      set(value = '') {
        _name = value;
      },
    });

    return Employee;
}());
Employee.prototype.constructor = Employee;
// Métodos públicos
Employee.prototype.toString = function () {
  return `(${this.username}): ${this.name} ${this.id}`;
};
```

La limitación que tenemos con este enfoque es que estamos obligados a mantener las propiedades que necesiten acceso directo a los campos privados, dentro del cuerpo del constructor. Al tratarse de un *closure* estas propiedades no pueden ser heredadas desde el prototipo, con la consecuente pérdida de rendimiento.

**Tenemos que decidir entre el encapsulamiento de los campos privados, y el rendimiento del objeto**.

## 3.2. Métodos privados

Los métodos privados son implementados mediante **funciones internas** en el cuerpo del constructor. Estos métodos solamente son accesibles desde el cuerpo del constructor, y nunca podrán utilizarse desde los métodos asignados desde el prototipo.

Vamos a crear una propiedad de dato *username* de solo lectura cuyo valor sea calculado a partir de la concatenación del nombre del empleado y su identificador, utilizando una función interna. La función *getUserName()* implementa el método privado, aunque realmente no es un método, ya que no tiene acceso al contexto del objeto. Puede acceder a los campos privados, pero no a las propiedades.

```
// Método interno
function getUserName(){
  return _name + _id;
}
Object.defineProperty(this, 'username', {
  get:function(){
    return getUserName();
  }
});
```

## 3.3. Métodos públicos

Como hemos comentado, los métodos asignados al prototipo no pueden acceder a los campos privados, ya que están fuera del *closure*. Todos estos métodos tienen carácter público, pero tienen acceso al contexto del objeto, por lo que se pueden implementar invocando el resto de las propiedades públicas del objeto. Por eficiencia, siempre deberíamos utilizar este tipo de métodos, ya que son heredados por todos los objetos construidos en base al constructor, mejorando el rendimiento en velocidad y memoria.

El método público más típico es *toString()*. Vamos a implementarlo para reducir a una única cadena de texto el objeto y lo añadimos a través del prototipo del constructor.

```
// Métodos públicos
Employee.prototype.toString = function (){
  return "(" + this.username + "): " + this.name + " " + this.id;
}
```

## 3.4. Miembros estáticos

Los miembros estáticos son métodos y propiedades que no necesitan de una instancia para poder ejecutarse, es decir, no están acogidos a un determinado contexto para implementar su función. Para crear un miembro estático debemos asociarlo como propiedad en la función constructora.

Vamos a crear un generador de números enteros aleatorios como función estática de la función constructora *Employee*.

```
// Método estático
Employee.getRandomNumber = function (level = 10) {
  return Math.floor(Math.random() * level);
};
```

## 3.5. Ejercicio ámbito de objetos

Realiza la implementación de los objetos *Manager*, *WorkerBee*, *SalerPerson* y *Engineer* manteniendo sus propiedades como campos internos. Haz que las propiedades de tipo *array* denominadas *reports* y *projects* de *Manager* y de *Engineer* respectivamente, solo se pueda modificar el contenido desde fuera del objeto, no el *array* completo.

En este ejercicio debes ser consciente de qué propiedades estás haciendo privadas, y cuáles son públicas. Como norma, deja públicas las propiedades que no tienen que pasar ningún tipo de validación para ser añadidas la objeto. En caso de necesitar algún tipo de control por parte de la lógica de negocio, deberás hacerlas privadas, y accesibles mediante una propiedad de acceso.

Debemos pensar cómo tenemos que implementar las propiedades *dept* y *specialty* utilizadas en los pasos anteriores, e implementadas como herencia de prototipos.

### 3.5.1. Solución

El constructor *Employee* lo mantenemos igual que en el punto anterior. Hemos añadido las dos propiedades *dept* y *specialty* al prototipo del constructor para que pueda ser heredado, el resultado es que no podemos aplicar una validación a través de una propiedad de acceso. De ser necesario, deberíamos mover estas propiedades al cuerpo de la función.

```
// Propiedades públicas heredadas
Employee.prototype.dept = 'general';
Employee.prototype.specialty = 'none';
```

Implementamos *Manager* y *WorkerBee* con el campo privado para *reports* y *projects* respectivamente. La invocación al superconstructor se hace de la forma usual, pero ahora en lugar de copiar propiedades, accedemos mediante *closure*.

```
//Constructor Manager
function Manager (name, reps = []) {
  //Invocamos al superconstructor Employee con call y pasandole como cont
exto this y los argumentos que espera.
  Employee.call(this, name);

  let _reports = reps;
  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'reports', {
    get:function(){
      return _reports;
    }
  });
}
//Manager hereda del objeto Employee sus propiedades
Manager.prototype = Object.create(Employee.prototype);
Manager.prototype.constructor = Manager;
```

```
//Constructor WorkerBee
function WorkerBee (name, projs = []) {
  //Invocamos al superconstructor Employee con call y pasandole como cont
exto this y los argumentos que espera.
  Employee.call(this, name);

  let _projects = projs;
  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'projects', {
    get:function(){
      return _projects;
    }
  });
}
//WorkerBee hereda del objeto Employee sus propiedades
WorkerBee.prototype = Object.create(Employee.prototype);
WorkerBee.prototype.constructor = WorkerBee;
```

Los constructores *SalesPerson* y *Engineer* siguen la misma línea.

```
//Constructor SalesPerson
function SalesPerson (name, projs, q = 100) {
  WorkerBee.call(this, name, projs);

  let _quota = q;
  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'quota', {
    get:function(){
      return _quota;
    }
  });
}
//SalesPerson hereda del objeto WorkerBee sus propiedades
SalesPerson.prototype = Object.create(WorkerBee.prototype);
SalesPerson.prototype.constructor = SalesPerson;
SalesPerson.prototype.dept = "sales";
```

```
//Constructor Engineer
function Engineer (name, projs, mach = "", hobby) {
  WorkerBee.call(this, name, projs);

  let _machine = mach;
  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'machine', {
    get:function(){
      return _machine;
    }
  });
}
//Engineer hereda del objeto WorkerBee sus propiedades
Engineer.prototype = Object.create(WorkerBee.prototype);
Engineer.prototype.constructor = Engineer;
Engineer.prototype.dept = "engineering";
Engineer.prototype.specialty="code";
```

Por último, en la función de testeo vemos como se heredan todas las propiedades con las llamadas de superconstructores y con la herencia de prototipos.

```
function testExample() {
  $$result.clear();
  const employee = new Employee('anon');
  const manager = new Manager('manager', ['project1', 'project2',
'project3']);
  const engineer = new Engineer('engineer', ['project4', 'project5',
'project6'], 'Machine1');
  $$result.logBold(manager.id); // 2
  $$result.log(manager.name); // manager
  $$result.log(manager.reports); // project1,project2,project3
  $$result.log(manager.dept); // general
  $$result.log(manager.specialty); // none
  engineer.name = 'Pablo';
  $$result.logBold(engineer.id); // 3
  $$result.log(engineer.name); // Pablo
  $$result.log(engineer.projects); // project4,project5,project6
  $$result.log(engineer.dept); // engineering
  $$result.log(engineer.specialty); // code
}
```

## 3.6. Invocación de métodos de la clase padre

### 3.6.1. Métodos en el prototipo

En *Employee* tenemos un método *toString()* heredado desde el prototipo. A partir del ejercicio anterior con toda la jerarquía de clases completada, en cada uno de los prototipos de los constructores vamos a añadir un método *toString()*, pero invocando la implementación de la clase padre, con lo que no necesitaremos hacer una implementación desde cero, solamente añadiremos las propiedades que necesitamos de cada tipo de objeto.

La llamadas a un método de la clase padre la tenemos que hacer invocando al método utilizando el método `call()` pasando como contexto el objeto actual que estamos ejecutando. Es independiente dónde reside el método, en el prototipo o en la función constructora. El siguiente código implementa el método *toString()*.

```
Manager.prototype.toString = function (){
  return Employee.prototype.toString.call(this) + " Reports: " + this.rep
orts;
}
WorkerBee.prototype.toString = function (){
  return Employee.prototype.toString.call(this) + " Projects: " + this.pr
ojects;
}
SalesPerson.prototype.toString = function (){
  return WorkerBee.prototype.toString.call(this) + " Quota: " + this.quot
a;
}
Engineer.prototype.toString = function (){
  return WorkerBee.prototype.toString.call(this) + " Machine: " + this.ma
chine;
}
```

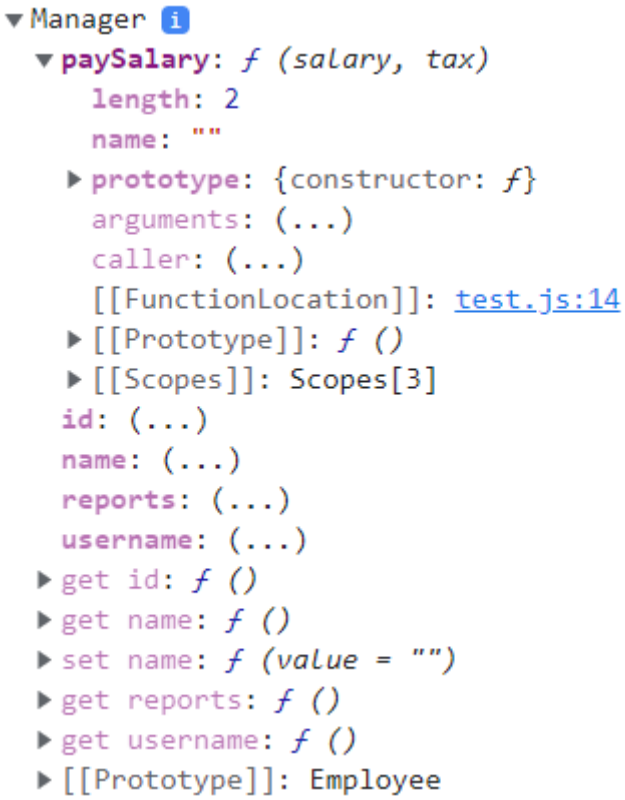
### 3.6.2. Métodos con campos privados

Queremos almacenar en un array cada una de las pagas que recibe un empleado. Este array estará protegido en un campo privado, y estará accesible mediante un *closure* de los métodos que residen en el constructor de *Employee*. También hemos creado un método *paySalary()* con dos parámetros con el salario y el porcentaje de impuestos que debe pagar el empleado, el cual almacenará el valor final de la nómina en el array.

```
let paychecks = [];

this.paySalary = function(salary, tax){
    paychecks.push(salary - (salary * tax / 100));
}
```

Cuando instanciamos un objeto *Manager* o *WokerBee* podemos acceder al método *paySalary()* porque es añadido directamente en el los objetos al invocar el constructor *Employee()*, pero no reside en el prototipo.




El problema que se nos presenta es que si queremos sobrescribir el método *paySalary()* dentro de *Manager* o *WorkerBee*, los constructores de las clases hijas, perdemos la referencia de *paySalary()* que tenemos heredada de *Employee* en la llamada al superconstructor, ya que ambos métodos utilizan la misma referencia.

La solución pasa por guardar la referencia en al método de la clase padre en un campo privado para que no se pierda, definir el método en la clase hija, y utilizar la referencia guardada para invocar al método de la clase hija porque ahora ya pueden coexistir en el mismo objeto.

```
let paySalaryEmployee = this.paySalary;
// Guardamos la referencia a paySalary heredado de Employee.
this.paySalary = function(salary, tax, bonus){ // El nuevo método
paySalary invoca al del padre.
    paySalaryEmployee.call(this, salary + bonus, tax);
}
```

Ahora vamos a crear un nuevo método que devuelva el total de las nóminas de un empleado en el constructor *Employee*. Este método puede acceder al campo privado *paychecks* por estar creado dentro del constructor.

```
this.getSalaries = function(){
    return paychecks.reduce((salary, total) => total + salary, 0);
}
```

Para el caso de los *Manager* o *WorkerBee* queremos cambiar el comportamiento del método. Sobrescribimos el método para recibir un argumento con la cantidad de impuestos anuales del empleado, que será restado al total de las nóminas.

```
this.getSalaries = function(annualTax = 0){
    let total = paychecks.reduce((salary, total) => total + salary, 0);
    return total - annualTax;
}
```

El método parece en principio bien implementado, pero al invocarlo generará una excepción porque el array *paychecks* no está definido. La razón es porque los métodos del constructor *Manager* o *WorkerBee* no tiene acceso campo privado del array porque no forma parte del cerramiento de los métodos.

La solución para que, en una jerarquía de objetos, los objetos de las clases hijas tengan acceso a los campos privados de la clase padre, siendo este campo privado un array o un objeto, pasa por tener el campo privado en cada uno de los constructores de la jerarquía, y que todos los campos tengan asignada la misma referencia al mismo objeto. Esta referencia única tiene que pasarse desde los constructores de las clases hijas a los constructores de la clase padre.

En los constructores de *Manager y WorkerBee* definimos el campo privado antes de la invocación del constructor de *Employee*. En la llamada al constructor pasamos la instancia creada.

```
let paychecks = []; // Campo privado para nóminas

//Invocamos al superconstructor Employee con call y pasandole como
contexto this y los argumentos que espera.
// Invocamos al superconstructor pasando el campo privado.
Employee.call(this, name, paychecks);
```

En el constructor de *Employee* declaramos el parámetro con un valor por defecto para en el caso de que se utilice un objeto *Employee*.

```
function Employee (name = "", checks = []) {
// Redefinimios el constructor para el array de las nóminas.
```

Y por último asignamos el valor de *checks* en el campo privado.

```
let paychecks = checks;
// Asignamos el array compartido entre constructores.
```

# 4. Objetos `Error`

Los objetos `Error` tienen dos cometidos principales:

- Son lanzados cuando ocurre un error en tiempo de ejecución.
- Funcionan como base para crear excepciones personalizadas por el usuario.

Las propiedades de una instancia de un objeto Error son:

- *name*: con el nombre del error.
- *message*: con el mensaje del error.

Podemos encontrar más propiedades en función del navegador utilizado, pero no están estandarizadas. Un ejemplo es *stack* con la pila de llamadas de funciones.

En cuanto a los métodos tenemos:

- *toString()*: reduce el objeto a un *string*.

## 4.1. Lanzar error genérico

Como ya hemos visto, mediante un bloque `try…catch` podemos controlar la gestión de errores. La estructura habitual para gestionar errores es en el bloque `try` invocar una función que lanza un error, y el bloque `catch` se encarga de capturar el error para que no se reproduzca al usuario. En lugar de utilizar un *string* podemos utilizar un objeto `Error`.

Como vemos en el ejemplo, al instanciar un objeto `Error` podemos definir la propiedad `message` en la invocación del constructor.

```
function genericError() {
  throw new Error('Throw generic error');
}
try {
  $$result.log('Before error'); // Before error
  genericError();
  $$result.log('After error');
} catch (error) {
  $$result.log(error.name); // Error
  $$result.log(error.message); // Throw generic error
}
```

## 4.2. Tipos de errores

Existen siete constructores de errores nativos en JavaScript.

- `EvalError`: Representa un error respecto a la función global `eval()`.
- `InternalError`: Instancia un error interno en el motor de JavaScript.
- `RangeError`: Este tipo de error ocurre cuando una variable excede del rango válido de valores que puede utilizar.
- `ReferenceError`: Crea una instancia que representa un error que ocurre cuando se quita la referencia a una referencia no válida.
- `SyntaxError`: Se utilizan para errores de sintaxis.
- `TypeError`: Representa un error con un tipo inválido de variable o parámetro.
- `URIError`: Instancia un error en el codificado o decodificado de una URI.

Para manejar errores específicos necesitamos hacer uso del operador `instanceof` para chequear de qué tipo de error se trata, ya que tan solo se admite un único bloque `catch`. Podemos realizar un bloque `if…else` anidado.

```
function referenceError() {
  throw new ReferenceError('Error in a reference');
}
function typeError() {
  throw new TypeError('Error in a type');
}
try {
  referenceError();
} catch (error) {
  if (error instanceof ReferenceError) {
    // ReferenceError: Error in a reference
    $$result.log(`${error.name}: ${error.message}`);
  } else if (error instanceof TypeError) {
    $$result.log(`${error.name}: ${error.message}`);
  }
}
try {
  typeError();
} catch (error) {
  if (error instanceof ReferenceError) {
    $$result.log(`${error.name}: ${error.message}`);
  } else if (error instanceof TypeError) {
    // TypeError: Error in a type
    $$result.log(`${error.name}: ${error.message}`);
  }
}
```

## 4.3. Errores personalizados

Podemos implementar nuestros propios objetos de error extendiendo el objeto `Error` a través de herencia. Una primera implementación básica sería la siguiente, en la que estamos utilizando la herencia por prototipos.

```
function MyError(message = 'Default Message') {
  this.name = 'MyError';
  this.message = message;
}
MyError.prototype = Object.create(Error.prototype);
MyError.prototype.constructor = MyError;

try {
  throw new MyError();
} catch (error) {
  $$result.log(error.name); // 'MyError'
  $$result.log(error.message); // 'Default Message'
}

try {
  throw new MyError('custom message');
} catch (error) {
  $$result.log(error instanceof MyError); // true
  $$result.log(error instanceof Error); // true
  $$result.log(error.name); // 'MyError'
  $$result.log(error.message); // 'custom message'
  $$result.log(error.stack); // undefined
}
```

Uno de los problemas de esta aproximación es que no tenemos control absoluto sobre el error, como hemos visto, no tenemos acceso a la pila de llamadas.

En el siguiente ejemplo estamos creando nuestro error personalizado. En el cuerpo del constructor creamos una instancia de tipo `Error`, añadimos nuestras propiedades personalizadas, y le fijamos como prototipo el prototipo del contexto. Por último, añadimos la pila de llamadas con el método `Error.captureStackTrace()` asociando a la instancia la función constructora. Es necesario realiza un retorno explícito de la instancia.

```
function MyError(myproperty = 'unknown', message = 'Default Message',
fileName, lineNumber) {
  const instance = new Error(message, fileName, lineNumber);
  instance.myproperty = myproperty;
  instance.name = 'MyError';
  Object.setPrototypeOf(instance, Object.getPrototypeOf(this));
  if (Error.captureStackTrace) {
    Error.captureStackTrace(instance, MyError);
  }
  return instance;
}
MyError.prototype = Object.create(Error.prototype, {
  constructor: {
    value: MyError,
    enumerable: false,
    writable: true,
    configurable: true,
  },
});

try {
  throw new MyError('myprop', 'custom message');
} catch (error) {
  $$result.log(error instanceof MyError); // true
  $$result.log(error instanceof Error); // true
  $$result.log(error.myproperty); // myprop
  $$result.log(error.name); // MyError
  $$result.log(error.message); // custom message
  $$result.log(error.stack); // ...
}
```

## 4.4. Jerarquía de excepciones

Como acabamos de ver, crear excepciones personalizadas puede ser tedioso. Vamos a crear un tipo de objeto *BaseException* donde realicemos la integración con la clase `Error`, así facilitamos la creación y mantenimiento de nuestra propia jerarquía de excepciones. Creamos el constructor.

```
function BaseException(message = "Default Message",
  fileName, lineNumber) {
  let instance = new Error(message, fileName, lineNumber);
  instance.name = "MyError";
  Object.setPrototypeOf(instance, Object.getPrototypeOf(this));
  if (Error.captureStackTrace) {
    Error.captureStackTrace(instance, BaseException);
  }
  return instance;
}
BaseException.prototype = Object.create(Error.prototype, {
  constructor: {
    value: BaseException,
    enumerable: false,
    writable: true,
    configurable: true
  }
});
```

Si fuese necesario, podríamos ampliar la clase anterior con las propiedades personalizadas que necesitásemos, que se pudieran utilizar en cualquier clase que herede de *BaseException*.

Creamos una excepción personalizada en base a la anterior *InvalidAccessConstructorException*. Las ventajas de crear excepciones personalizadas son, en primer lugar, que los mensajes de las excepciones quedan estandarizados al estar definidos en el cuerpo del constructor. En segundo lugar, nos permite utilizar el operador `intanceof` para identificar qué tipo de excepción estamos capturando en un bloque `catch`.

```
//Excepción acceso inválido a constructor
function InvalidAccessConstructorException() {
  let instance = BaseException.call(this, "Constructor can't be called as
 a function.");
  instance.name = "InvalidAccessConstructorException";
  return instance;
}
InvalidAccessConstructorException.prototype = Object.create(BaseException
.prototype);
InvalidAccessConstructorException.prototype.constructor = InvalidAccessCo
nstructorException;
```

Veamos un ejemplo de cómo podríamos utilizarlo. En principio, el modo estricto impide que podamos invocar un constructor sin el operador `new`, pero si no lo activamos, deberíamos verificar que se pueda invocar sin el operador utilizando una excepción. El siguiente constructor lanzaría una excepción en caso de no ser invocado con el operador `new`. Como vemos, utilizamos el operador `instanceof` para saber si el contexto tiene como prototipo el de la función constructora.

```
try {
  const testObject = TestObject();
} catch (error) {
  // InvalidAccessConstructorException: Constructor can’t be called as a
function.
  $$result.log(error instanceof BaseException); // true
  $$result.log(error instanceof InvalidAccessConstructorException); //
true
  $$result.log(error.toString());
  $$result.log(error.stack);
}
```

### 4.4.1. Ejercicio

Crea un tipo de excepción para especificar que un argumento tiene un valor vacío. Guarda el nombre del argumento como propiedad en el objeto de la excepción.

De la misma forma crea un segundo tipo de excepción para especificar que un argumento ha recibido un valor inválido. Almacena en el objeto de la excepción el nombre del parámetro y el valor asociado como propiedades.

### 4.4.2. Solución

```
//Excepción personalizada para indicar valores vacios.
function EmptyValueException(param) {
  let instance = BaseException.call(this, "Error: The parameter " + param
 + " can't be empty.");
  instance.name = "EmptyValueException";
  instance.param = param;
  return instance;
}
EmptyValueException.prototype = Object.create(BaseException.prototype);
EmptyValueException.prototype.constructor = EmptyValueException;

//Excepción de valor inválido
function InvalidValueException(param, value) {
  let instance = BaseException.call(this, "Error: The paramenter " + para
m + " has an invalid value. (" + param + ": " + value + ")");
  instance.name = "InvalidValueException";
  instance.param = param;
  instance.param = value;
  return instance;
}
InvalidValueException.prototype = Object.create(BaseException.prototype);
InvalidValueException.prototype.constructor = InvalidValueException;
```

En cuanto al testeo tendríamos:

```
try {
  throw new EmptyValueException('price');
} catch (error) {
  // EmptyValueException: Error: The parameter price can't be empty.
  $$result.log(error.toString());
}
try {
  throw new InvalidValueException('price', 'aaa');
} catch (error) {
  // InvalidValueException: Error: The paramenter price has an invalid
value. (price: aaa)
  $$result.log(error.toString());
}
```

## 5. Ejercicios productos

Tenemos que implementar la estructura de clases necesaria para mantener nuestra aplicación **MiTienda**. Esta aplicación la iremos evolucionando a lo largo de las unidades.

La estructura debe constar de las clases para mantener productos de tipo *Laptop*, *Camera*, *SmartPhone* y *Tablet*. Cada producto debe recoger la siguiente información de forma obligatoria:

- Número de serie
- Marca
- Modelo
- Precio
- Porcentaje de impuesto sobre el precio.

Además, debemos poder añadir una descripción al producto de manera opcional.

Como propiedades calculadas necesitamos saber:

- El precio del producto sin impuestos.
- El total de impuestos pagado por el producto.
- Además, mantendremos una constante con el porcentaje de IVA aplicado por defecto.

Por último, deberemos tener un método *toString* para reducir el objeto a un *string*.

En función del tipo de producto debemos tener las siguientes propiedades.

- *Laptop*:
  - Procesador.
  - Memoria. En formato número indicando si son GB o TB.
  - Disco duro: HDD o SDD
  - Tamaños del disco duro: En formato número indicando si son GB o TB.
  - Sistema operativo.
- *Camera*:
  - Tipo: Digital o Reflex.
  - Resolución: Numérico seguido de la unidad MP.
  - Tamaño: Numérico en pulgadas
- *SmartPhone*:
  - Memoria. En formato número indicando si son GB o TB.
  - Almacenamiento: En formato número indicando si son GB o TB.
  - Tamaño: Numérico en pulgadas.

    - Resolución: Ancho por alto.
    - Sistema operativo.
- *Tablet*:
    - Memoria. En formato número indicando si son GB o TB.
    - Almacenamiento: En formato número indicando si son GB o TB.
    - Tamaño: Numérico en pulgadas.
    - Resolución: Ancho por alto. 1024x1024
    - Sistema operativo.

Debemos pensar qué propiedades son públicas, y cuáles necesitan ser mantenidas mediante campos internos por ser necesario su validación antes de ser añadidas al objeto.

Debemos implementar una estructura de clases para representar las excepciones producidas en el código. Podemos reutilizar las excepciones del punto anterior, o diseñar nuevas excepciones en función de nuestras necesidades.

Por último, debemos testear cada una de las clases creadas para verificar que todos los métodos implementados funcionan correctamente.

Dentro del directorio *js* vamos a crear un directorio *mitienda* para albergar los ficheros que vamos a utilizar en el proyecto.

## 5.1. Solución

Lo explicamos por pasos.

### 5.1.1. Excepciones

Reutilizamos las excepciones creadas en el punto 4. Creamos un fichero específico dentro de *mitienda* para ellas, **exceptions.js** y añadimos los constructores creados.

```
function BaseException(message = "Default Message", fileName, lineNumber)
 {
  let instance = new Error(message, fileName, lineNumber);
  instance.name = "MyError";
  Object.setPrototypeOf(instance, Object.getPrototypeOf(this));
  if (Error.captureStackTrace) {
    Error.captureStackTrace(instance, BaseException);
  }
  return instance;
}
BaseException.prototype = Object.create(Error.prototype, {
  constructor: {
    value: BaseException,
    enumerable: false,
    writable: true,
    configurable: true
  }
});
```

```
//Excepción acceso inválido a constructor
function InvalidAccessConstructorException() {
  let instance = BaseException.call(this, "Constructor can’t be called as
 a function.");
  instance.name = "InvalidAccessConstructorException";
  return instance;
}
InvalidAccessConstructorException.prototype = Object.create(BaseException
.prototype);
InvalidAccessConstructorException.prototype.constructor = InvalidAccessCo
nstructorException;

//Excepción personalizada para indicar valores vacios.
function EmptyValueException(param) {
  let instance = BaseException.call(this, "Error: The parameter " + param
 + " can't be empty.");
  instance.name = "EmptyValueException";
  instance.param = param;
  return instance;
}
EmptyValueException.prototype = Object.create(BaseException.prototype);
EmptyValueException.prototype.constructor = EmptyValueException;

//Excepción de valor inválido
function InvalidValueException(param, value) {
  let instance = BaseException.call(this, "Error: The paramenter " + para
m + " has an invalid value. (" + param + ": " + value + ")");
  instance.name = "InvalidValueException";
  instance.param = param;
  instance.param = value;
  return instance;
}
InvalidValueException.prototype = Object.create(BaseException.prototype);
InvalidValueException.prototype.constructor = InvalidValueException;
```

### 5.1.2. Implementación de *Product*

Implementamos los constructores que vamos a utilizar para resolver el ejercicio. Creamos el directorio *enteties* y dentro albergamos **products.js**.

Los puntos que hemos tenido en cuenta son los siguientes:

1. Definición de la función. Como parámetros debe disponer de los parámetros obligatorios del objeto. El parámetro *taxPercentage* le asignamos el tipo de IVA por defecto, el cual será una constante estática de la función constructora.

```
function Product(serial, brand, model, price,
taxPercentage = Product.IVA){
```

2. Cuerpo de la función. Seguimos siempre los mismos pasos:

a. Verificación de si la invocación se ha hecho con el operador `new`.

```
  if (!(this instanceof Product))
    throw new InvalidAccessConstructorException();
```

b. Validación de argumentos recibidos. Si encontramos algún argumento con valor no válido debemos lanzar una excepción. Podemos verificar si los valores son numéricos, si son negativos, si están en un rango o si son vacíos. Cualquier validación que se nos pueda ocurrir.

```
  if (!serial) throw new EmptyValueException("serial");
  if (!brand) throw new EmptyValueException("brand");
  if (!model) throw new EmptyValueException("model");
  price = Number.parseFloat(price);
  if (!price || price <= 0) throw new InvalidValueException("price", pric
e);
  if (!taxPercentage || taxPercentage < 0) throw new InvalidValueExceptio
n("taxPercentage", taxPercentage);
```

c. Definición de atributos o campos privados. Tenemos que definir las variables que almenarán los valores asociados a las propiedades de acceso.

```
  let _serial = serial;
  let _brand = brand;
  let _model = model;
  let _price = price;
  let _taxPercentage = taxPercentage;
```

d. Definimos las propiedades de acceso. En principio, cada *setter* debe tener las mismas validaciones que el constructor. Cada función de las propiedades de acceso habilita el acceso externo a los campos privados por tratarse de un *closure*.

```
  Object.defineProperty(this, 'serial', {
    get:function(){
      return _serial;
    },
    set:function(value){
      if (!value) throw new EmptyValueException("serial");
      _serialNumber = value;
    }
  });

  Object.defineProperty(this, 'brand', {
    get:function(){
      return _brand;
    },
    set:function(value){
      if (!value) throw new EmptyValueException("brand");
      _brand = value;
    }
  });
```

```
  Object.defineProperty(this, 'model', {
    get:function(){
      return _model;
    },
    set:function(value){
      if (!value) throw new EmptyValueException("model");
      _model = value;
    }
  });

  Object.defineProperty(this, 'price', {
    get:function(){
      return _price;
    },
    set:function(value){
      value = Number.parseFloat(value);
      if (Number.isNaN(value) && value > 0) throw new InvalidValueExcepti
on("price", value);
      _price = value;
    }
  });

  Object.defineProperty(this, 'taxPercentage', {
    get:function(){
      return _taxPercentage;
    },
    set:function(value = Product.IVA){
      if (!value || value < 0) throw new InvalidValueException("taxPercen
tage", value);
      _taxPercentage = value;

    }
  });
}
```

3. Propiedades del prototipo.
    a. Definimos el objeto prototipo y el constructor.

```
Product.prototype = {};
Product.prototype.constructor = Product;
```

    b. Propiedades del públicas. No todas las propiedades deben ser privadas. Las que no tengan validación asociada podemos implementarlas mediante propiedades de valor. Un ejemplo es *description* la cuál no tiene ninguna restricción que cumplir, por lo que la asociamos desde el prototipo.

```
Object.defineProperty(Product.prototype, 'description', {
  enunmerable: true,
  writable: true,
  configurable: false
});
```

c. Propiedades calculadas. Son propiedades que devuelven un valor dinámico en base a otras propiedades del objeto. Son propiedades de acceso solamente con *getter*. Son similares a los métodos.

```
Object.defineProperty(Product.prototype, 'priceWithoutTaxes', {
  get:function(){
    return this.price - (this.price * this.taxPercentage / 100);
  }
});
Object.defineProperty(Product.prototype, 'tax', {
  get:function(){
    return this.price * this.taxPercentage / 100;
  }
});
```

d. Métodos públicos. Realizan un calculo dinámico en base a las propiedades públicas del objeto, pero a diferencia de las propiedades, pueden definir parámetros de entrada que personalizan su ejecución.

```
Product.prototype.toString = function(){
  return "Serial: " + this.serial + " Brand: " + this.brand + " Model: "
+ this.model + " Price: " + this.price + "€ Tax: " + this.taxPercentage +
 "%";
}
```

4. Propiedades estáticas. Están asociadas al constructor y no al prototipo porque no necesitan una instancia para poder ejecutarse.

```
Object.defineProperty(Product, 'IVA', {
    value:21,
    writable:false,
    enumerable:true,
    configurable:false
});
```

### 5.1.3. Implementación de *Laptop*

La definición de este constructor debe recoger tanto los parámetros definidos en *Product* como los que necesite *Laptop* para crear sus propiedades. Debemos tener mucha precaución a la hora de definir un constructor ya que, si el constructor requiere muchos parámetros obligatorios, podría resultar muy tedioso utilizar este tipo de constructor. Lo ideal es tener valores por defecto asociados a ciertos parámetros, para que tengamos flexibilidad a la hora de invocar el constructor, es decir, que podamos invocar el constructor con todos los parámetros que necesitemos, pero si no disponemos de todos los valores necesarios en ese momento, podamos también crear el objeto.

En la definición del constructor hemos seguido este principio, hemos añadido en primer lugar los parámetros que se utilizarán para invocar al superconstructor, para que en segundo termino definamos los parámetros utilizados en el cuerpo del constructor, todos ellos definidos con valores por defecto que nos permitan invocar el constructor sin necesidad asignarles un valor.

```
function Laptop(serial, brand, model, price, taxPercentage = Product.IVA,
 processor = "unkonwn", memory = "0GB", hd = "-", size = "0GB"){
```

Una vez comprobado que se ha invocado el constructor con el operador `new`, debemos invocar al superconstructor con el método `call()` y pasando como contexto el objeto que estamos construyendo.

```
  //La función se invoca con el operador new
  if (!(this instanceof Laptop))
    throw new InvalidAccessConstructorException();
  //Llamada al superconstructor.
  Product.call(this,serial,brand,model,price,taxPercentage);
```

Posteriormente validamos los argumentos recibidos en función de las necesidades del constructor y siguiendo la lógica de negocio de nuestra aplicación. En este ejemplo hacemos uso de expresiones regulares para validad los *string* recibidos.

```
  //Validación de argumentos
  if (!processor) throw new EmptyValueException("processor");
  if (!/^((\d+GB)|(\d+TB))$/.test(memory)) throw new InvalidValueExceptio
n("memory",memory);
  if (!/^((HDD)|(SDD)|(-
))$/.test(hd)) throw new InvalidValueException("hd",hd);
  if (!/^((\d+GB)|(\d+TB))$/.test(size)) throw new InvalidValueException(
"size",size);
```

Debemos declarar las variables que contendrán los campos privados y hacemos la asignación con los argumentos recibidos ya validados.

```
  //Atributos privados
  let _processor = processor;
  let _memory = memory;
  let _hd = hd;
  let _size = size;
```

El último paso es definir las propiedades de acceso a los campos privados, teniendo en cuenta que los *setter* debe realizar la misma validación de los argumentos del constructor.

```
  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'processor', {
    get:function(){
      return _processor;
    },
    set:function(value){
      if (!value) throw new EmptyValueException("processor");
      _processor = value;
    }
  });
```

```
  Object.defineProperty(this, 'memory', {
    get:function(){
      return _memory;
    },
    set:function(value){
      if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new InvalidValueExcep
tion("memory",value);
      _memory = value;
    }
  });

  Object.defineProperty(this, 'hd', {
    get:function(){
      return _hd;
    },
    set:function(value){
      if (!/^((HDD)|(SDD))$/.test(value)) throw new InvalidValueException
("hd",value);
      _hd = value;
    }
  });

  Object.defineProperty(this, 'size', {
    get:function(){
      return _size;
    },
    set:function(value){
      if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new InvalidValueExcep
tion("size",value);
      _size = value;
    }
  });
}
```

Fuera del constructor tenemos que declarar la herencia con *Product*. Como siempre lo hacemos asociando el prototipo de *Laptop* con un objeto que tenga como prototipo el prototipo de *Product*, y asignando la propiedad `constructor` con la función *Laptop*.

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

Tenemos que declarar las posibles propiedades públicas con valores por defecto asociándolas al prototipo. En este ejemplo hemos utilizado *system* por no considerar que necesiten validación o que cumplan con algún requisito de la lógica de negocio. Si necesitásemos que esta propiedad fuera asignada en el constructor, deberíamos definirla en el cuerpo del constructor.

```
Laptop.prototype.system = "Unknown"; // Propiedad pública
```

Por último, tenemos métodos públicos que hacen uso de las propiedades públicas y que no requieran acceso directo a los campos privados. El método *toString()* cumple estos requisitos y hacen llamadas a método *toString()* de *Product*.

```
Laptop.prototype.toString = function(){
  return Product.prototype.toString.call(this) + " System: " + this.syste
m + " Processor: " + this.processor +
    " Memoria: " + this.memory + " HD: " + this.hd +
    " Size: " + this.size;
}
```

### 5.1.4. Resto de constructores

El resto de los constructores siguen la misma estructura que *Laptop*. Los únicos cambios estarían basados en la declaración de propiedades específicas de cada tipo de objeto como es lógico, y sus validaciones.

```
//Definimos la subclase Camera
function Camera(serial, brand, model, price, taxPercentage = Product.IVA,
 type = "-", resolution = 0, size = 0){
  //La función se invoca con el operador new
  if (!(this instanceof Camera))
    throw new InvalidAccessConstructorException();
  //Llamada al superconstructor.
  Product.call(this,serial,brand,model,price,taxPercentage);

  //Validación de argumentos
  resolution = Number.parseFloat(resolution);
  size = Number.parseFloat(size);
  if (!/^((Digital)|(Reflex)|(-
))$/.test(type)) throw new InvalidValueException("type",type);
  if (Number.isNaN(resolution) || resolution < 0) throw new InvalidValueE
xception("resolution",resolution);
  if (Number.isNaN(size) || size < 0) throw new InvalidValueException("si
ze",size);

  //Atributos privados
  let _type = type;
  let _resolution = resolution;
  let _size = size;
```

```
  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'type', {
    get:function(){
      return _type;
    },
    set:function(value){
      if (!/^((Digital)|(Reflex)|(-
))$/.test(value)) throw new InvalidValueException("type",value);
      _type = value;
    }
  });

  Object.defineProperty(this, 'resolution', {
    get:function(){
      return _resolution;
    },
    set:function(value){
      value = Number.parseFloat(value);
      if (Number.isNaN(value) || value < 0) throw new InvalidValueExcepti
on("resolution",value);
      _resolution = value;
    }
  });

  Object.defineProperty(this, 'size', {
    get:function(){
      return _size;
    },
    set:function(value){
      value = Number.parseFloat(value);
      if (Number.isNaN(value) || value < 0) throw new InvalidValueExcepti
on("size",value);
      _size = value;
    }
  });

}
Camera.prototype = Object.create(Product.prototype); //Heredamos de Produ
ct
Camera.prototype.constructor = Camera;
Camera.prototype.toString = function(){
  return Product.prototype.toString.call(this) +
    " Tipo: " + this.type + " Resolución: " + this.resolution + "MP Size:
 " + this.size + "''";
}

//Definimos la subclase Smartphone
```

```
function Smartphone(serial, brand, model, price, taxPercentage = Product.
IVA, memory = "0GB", storage = "0GB", resolution = "0x0", size = 0){
  //La función se invoca con el operador new
  if (!(this instanceof Smartphone))
    throw new InvalidAccessConstructorException();
  //Llamada al superconstructor.
  Product.call(this,serial,brand,model,price,taxPercentage);

  //Validación de argumentos
  if (!/^((\d+GB)|(\d+TB))$/.test(memory)) throw new InvalidValueExceptio
n("memory",memory);
  if (!/^((\d+GB)|(\d+TB))$/.test(storage)) throw new InvalidValueExcepti
on("storage",storage);
  size = Number.parseFloat(size);
  if (Number.isNaN(size) || size < 0) throw new InvalidValueException("si
ze",size);
  if (!/^(\d+x\d+)$/.test(resolution)) throw new InvalidValueException("r
esolution",resolution);

  //Atributos privados
  let _memory = memory;
  let _storage = storage;
  let _resolution = resolution;
  let _size = size;

  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'memory', {
    get:function(){
      return _memory;
    },
    set:function(value){
      if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new InvalidValueExcep
tion("memory",value);
      _memory = value;
    }
  });

  Object.defineProperty(this, 'resolution', {
    get:function(){
      return _resolution;
    },
    set:function(value){
      if (!/^(\d+x\d+)$/.test(value)) throw new InvalidValueException("re
solution",value);
      _resolution = value;
    }
  });
```

```
  Object.defineProperty(this, 'storage', {
    get:function(){
      return _storage;
    },
    set:function(value){
      if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new InvalidValueExcep
tion("storage",value);
      _storage = value;
    }
  });

  Object.defineProperty(this, 'size', {
    get:function(){
      return _size;
    },
    set:function(value){
      value = Number.parseFloat(value);
      if (Number.isNaN(value) || value < 0) throw new InvalidValueExcepti
on("size",value);
      _size = value;
    }
  });
}
Smartphone.prototype = Object.create(Product.prototype); //Heredamos de P
roduct
Smartphone.prototype.constructor = Smartphone;
Smartphone.prototype.system = "Unknown"; // Propiedad pública
Smartphone.prototype.toString = function(){
  return Product.prototype.toString.call(this) + " System: " + this.syste
m +
    " Memoria: " + this.memory + " Almacenamiento: " + this.storage + " R
esolución: " + this.resolution + " Size: " + this.size + "''";
}
```

```
//Definimos la subclase Tablet
function Tablet(serial, brand, model, price, taxPercentage = Product.IVA,
 memory = "0GB", storage = "0GB", resolution = "0x0", size = 0){
  //La función se invoca con el operador new
  if (!(this instanceof Tablet))
    throw new InvalidAccessConstructorException();
  //Llamada al superconstructor.
  Product.call(this,serial,brand,model,price,taxPercentage);

  //Validación de argumentos
  if (!/^((\d+GB)|(\d+TB))$/.test(memory)) throw new InvalidValueExceptio
n("memory",memory);
  if (!/^((\d+GB)|(\d+TB))$/.test(storage)) throw new InvalidValueExcepti
on("storage",storage);
  size = Number.parseFloat(size);
  if (Number.isNaN(size) || size < 0) throw new InvalidValueException("si
ze",size);
  if (!/^(\d+x\d+)$/.test(resolution)) throw new InvalidValueException("r
esolution",resolution);

  //Atributos privados
  let _memory = memory;
  let _storage = storage;
  let _resolution = resolution;
  let _size = size;

  //Propiedades de acceso a los atributos privados
  Object.defineProperty(this, 'memory', {
    get:function(){
      return _memory;
    },
    set:function(value){
      if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new InvalidValueExcep
tion("memory",value);
      _memory = value;
    }
  });
```

```
  Object.defineProperty(this, 'resolution', {
    get:function(){
      return _resolution;
    },
    set:function(value){
      if (!/^(\d+x\d+)$/.test(value)) throw new InvalidValueException("re
solution",value);
      _resolution = value;
    }
  });

  Object.defineProperty(this, 'storage', {
    get:function(){
      return _storage;
    },
    set:function(value){
      if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new InvalidValueExcep
tion("storage",value);
      _storage = value;
    }
  });

  Object.defineProperty(this, 'size', {
    get:function(){
      return _size;
    },
    set:function(value){
      value = Number.parseFloat(value);
      if (Number.isNaN(value) || value < 0) throw new InvalidValueExcepti
on("size",value);
      _size = value;
    }
  });
}
Tablet.prototype = Object.create(Product.prototype); //Heredamos de Produ
ct
Tablet.prototype.constructor = Tablet;
Tablet.prototype.system = "Unknown"; // Propiedad pública
Tablet.prototype.toString = function(){
  return Product.prototype.toString.call(this) + " System: " + this.syste
m +
    " Memoria: " + this.memory + " Almacenamiento: " + this.storage + " R
esolución: " + this.resolution + " Size: " + this.size + "''";
}
```

Por último, importamos los ficheros el *index.html*.

```
<script src="./js/mitienda/exceptions.js"></script>
<script src="./js/mitienda/entities/products.js"></script>
```