UT04.6: Clases en JavaScript

1 Contenido

1. Introducción

Con la llegada de la versión ES6, JavaScript ha añadido una sintaxis diferente para crear objetos basada en clases. Esta nueva sintaxis está más relacionada con leguajes clásicos orientados a objetos como podría ser Java, pero en realidad si analizamos qué está ocurriendo veremos que las instancias tienen una propiedad *__proto__* que apunta a la propiedad *prototype* de la clase.

Las clases tiene la misma utilidad que una función constructora. Se trata de una plantilla para crear objetos y los métodos de la lógica de negocio, pero con una sintaxis más amigable.

2. Clase de objetos

La palabra reservada `class` nos permite crear una clase de objetos asociándola a un nombre. Lo habitual es tener una función `constructor` que actuará como constructor de la clase.

2.1. Definición de una clase

Vamos a crear una clase *Person* con tres propiedades, *firstName*, *lastName* y *age*. Instanciamos un objeto y mostramos su contenido y el de su prototipo que en primera instancia está vacío.

```
class Person {
  constructor (firstName, lastName, age){
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }
}
let p1 = new Person("John", "Doe", 23);
// Person {firstName: John,lastName: Doe,age: 23,}
console.log(p1);
// Person { }
console.log(p1.__proto__);
```

Aunque el cuerpo de una clase es muy similar a un objeto literal, realmente no funcionan igual al no soportar la definición de propiedades con pares nombre y valor.

El constructor **no es obligatorio** si no tiene ningún parámetro en definición.

Por último, debemos tener en cuenta que implícitamente, la declaración de una clase se hace bajo del modo estricto independientemente que no lo hayamos definido. Las **declaraciones de las clases no son elevadas** como ocurre con el resto de las funciones, por lo que no podemos instanciar un objeto de una clase antes de que sea declarado.

2.2. Getter and Setter

Podemos utilizar propiedades de acceso para generar *getter* y *setter* en nuestra clase. Añadimos una propiedad *fullName* a la clase.

```
get fullName(){
  return this.firstName + " " + this.lastName;
}
set fullName(fullName){
  let name = fullName.split(" ");
  this.firstName = name[0];
  this.lastName = name[1];
}
```

Para testear invocamos tanto el setter como el getter para modificar y acceder a la propiedad. Debemos recordad que las propiedades de acceso no son enumerables.

```
p1.fullName = "Joe Doe";
console.log(p1.fullName); //Joe Doe
```

Por defecto, las propiedades de acceso no son enumerables. Podemos hacer que sean enumerables definiéndola dentro del constructor a través de `Object.defineProperty()` y marcando la propiedad `enumerable` como `true`.

```
Object.defineProperty(this, 'fullName2', {
  enumerable: true,
  get() { return `${this.firstName} ${this.lastName}`; },
  set(fullName) {
    const name = fullName.split(' ');
    this.firstName = name[0];
    this.lastName = name[1];
  },
});
```

2.3. Creando métodos

Para añadir un método no necesitamos utilizar la palabra reservada `function`. El siguiente método *idAdult()* es creamos en el cuerpo de la clase y es añadido método en el prototipo del objeto.

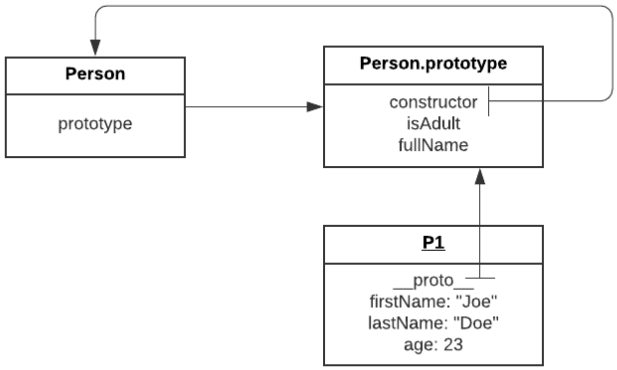
```
isAdult(){
  return this.age >= 18;
}
```

Invocamos el método con una instancia para testearlo.

```
console.log(p1.isAdult()); //true
```

2.4. Funcionamiento de las clases

Una clase realmente es una función. La declaración de la clase define una nueva variable *Person* y le asigna el valor de la función *constructor*. Su principal propósito es crear la cadena de prototipos. La clase *Person* tiene una propiedad *prototype* que apunta a un objeto, el cual tiene las propiedades *constructor*, el método *isAdult()* y la propiedad *fullName*.



La propiedad *constructor* apunta a la función *Person*. En el siguiente código podemos ver la comparativa que demuestra la relación de estos objetos, así como el uso del operador `instaceof` o el nombre del constructor.

```
console.log(Person.prototype.constructor === Person); //true
console.log(p1.__proto__ === Person.prototype); //true
console.log(p1 instanceof Person); //true
console.log(p1 instanceof Student); // false
console.log(p1.constructor.name); //Person
```

2.5. Herencia

Para implementar la herencia entre clases utilizamos la palabra reservada `extends`, y para invocar al superconstructor la palabra reservada `super`. Vamos a crear una subclase *Student* con una propiedad que contenga los cursos en los que se ha matriculado un alumno.

```
class Student extends Person {
  constructor (firstName, lastName, age){
    super(firstName, lastName, age);
    this._enrolledCourses = [];
  }
}
```

Modificamos el testeo para crear un objeto de tipo *Student*.

```
let s1 = new Student("John", "Doe", 23);
// Student {firstName: John,lastName: Doe,age: 23,_enrolledCourses: ,full
Name: John Doe,}
console.log(s1 instanceof Student); //true
console.log(s1 instanceof Person); //true
```

Al utilizar `super` en un constructor, debemos tener en cuenta las siguientes normas:

- Si la subclase no define ningún constructor, el intérprete **definirá uno automáticamente de forma implícita**. Este constructor recoge los argumentos recibidos e invoca el superconstructor con ellos utilizando `super`.
- La superclase debe ser inicializada antes que la subclase, por lo que no podemos utilizar la palabra reservada `this` antes de invocar a `super`.

2.6. Métodos en el prototipo

Los métodos definidos se encuentran dentro del prototipo de la clase. Vamos a crear un método *toString()* en *Person*.

```
toString() {
  return `${this.firstName} ${this.lastName} ${this.age}`;
}
```

Hacemos lo mismo para *Student,* pero invocando el método de la superclase utilizando la palabra reservada `super`.

```
toString(){
  return super.toString() + ` Cursos: ${this._enrolledCourses.length}`;
}
```

El ejemplo de testeo tenemos:

```
console.log(s1.toString()); //Joe Doe 23 Cursos: 0
```

2.7. Propiedades y métodos estáticos

Para declarar propiedades y métodos estáticos tenemos que hacerlo utilizando la palabra reservada `static`. Estas propiedades son añadidas a la función que representa la clase, y no a las instancias.

Vamos a crear en *Student* un método estático que permita instanciar un *Student* a partir de un objeto *Person*, y una propiedad con el curso actual.

```
static fromPerson(person){
  return new Student(person.firstName, person.lastName, person.age);
}
static course = 2021;
```

Para chequear el funcionamiento.

```
let s2 = Student.fromPerson(new Person("Jane","Smith",21));
console.log(s2.toString()); //Jane Smith 21 Cursos: 0
console.log(Student.course); //2021
```

2.8. Uso de `class` como expresión

Al igual que las funciones, las clases se pueden definir en una expresión de asignación a una variable o constante.

```
const Square = class {
  constructor(side) {
    this.side = side;
  }
  get area(){
    return this.side * this.side;
  }
}
let s = new Square(5);
console.log(s.area); //25
```

2.9. Propiedades en el cuerpo de la clase

Antes de continuar tenemos que tener en cuenta que este tipo de acción no es soportada por todos los navegadores, ya que es una modificación del 2020.

Podemos inicializar propiedades fuera del constructor, para ello basta con declarar la propiedad sin hacer uso de la palabra reservada `this`. Inicializar explícitamente todas las propiedades con un valor por defecto siempre es una buena práctica.

Las siguientes dos clases son ejemplos análogos de clases. En la versión *V2* vemos como las propiedades son declaradas directamente en el cuerpo de la clase, y les asignamos un valor.

```
class BufferV1 {
  constructor() {
      this.size = 0;
      this.capacity = 4096;
  }
}
class BufferV2 {
  size = 0;
  capacity = 4096;
}
let b1 = new BufferV1(); //4096
let b2 = new BufferV2(); //4096
console.log(b1.capacity);
console.log(b2.capacity);
```

3. Campos privados para las clases

Vamos a ver los mecanismos que disponemos para crear propiedades privadas con las clases. Para ello vamos a implementar una clase *Countdown* que nos sirva de contador, con una función de *callback* cuando el contador haya llegado a 0, y la iremos rediseñando en función de las diferentes alternativas.

3.1. Nombrado por convenio

No es realmente una protección, ya que lo único que hacemos es añadir un prefijo con el guion bajo a cada una de las propiedades internas para distinguirlas. Esto no es realmente ninguna protección, es una forma de indicar hacía fuera que esa propiedad es usada internamente.

Este mecanismo es útil si no queremos sobrecarga los objetos, pero tenemos que ser conscientes de que estamos habilitando una puerta para que se salten toda nuestra lógica de negocio.

Como vemos en el código, realmente seguimos teniendo acceso a las propiedades internas del objeto.

```
class Countdown {
  constructor(counter, action) {
    this._counter = counter;
    this._action = action;
  }
  dec() {
    this._counter--;
    if (this._counter === 0) {
      this._action();
    }
  }
}
const c = new Countdown(3, () => $$result.log('El contador ha parado.'));
c.dec();
c.dec();
c.dec(); //El contador ha parado
console.log(c._counter);//0
```

3.2. Uso de `WeakMap`

Los objetos `WeakMap` mantienen los valores asociados a una clave de tipo objeto, siempre y cuando el objeto siga instanciado y no haya pasado por el recolector de basura. Podemos declarar un contenedor global de tipo `WeakMap` para cada propiedad privada que necesitemos. En lugar de trabajar con propiedades, trabajaríamos con el mapa, aportando el grado de seguridad que necesitamos, ya que sin la instancia del objeto no podríamos recoger el valor del mapa.

```
const _counter = new WeakMap();
const _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

let c = new Countdown(3,()=>console.log("El contador ha parado."));
c.dec();
c.dec();
c.dec(); //El contador ha parado
console.log(c._counter);//undefined
```

Las ventajas es que tenemos una protección real, pero los inconvenientes es que se hace muy tedioso trabajar con este tipo de objetos. Si tuviéramos muchos campos privados podría ser inmanejable.

3.3. Uso de `Symbol`

Basado en el ejemplo anterior, podemos añadir al *registro de symbols* una entrada por cada campo privado que necesitemos. Cada *symbol* es único, con lo que no tendremos problemas en el nombrado de propiedades.

```
const _counter = Symbol('counter');
const _action = Symbol('action');
class Countdown {
    constructor(counter, action) {
        this[_counter] = counter;
        this[_action] = action;
    }
    dec() {
        if (this[_counter] < 1) return;
        this[_counter]--;
        if (this[_counter] === 0) {
            this[_action]();
        }
    }
}
```

```
let c = new Countdown(3,()=>console.log("El contador ha parado."));
c.dec();
c.dec();
c.dec(); //El contador ha parado
console.log(c._counter);//undefined
```

Los objetos `Symbol` en principio están ocultos al resto del mundo, pero podemos utilizar el API de reflexión para acceder a su contenido. En el siguiente código vemos cómo obtenemos todas las claves del objeto *Countdown* incluido los `Symbol`, con lo que podemos recuperar el valor que deseemos.

```
console.log(Reflect.ownKeys(c)); // [ Symbol(counter), Symbol(action) ]
console.log(c[Reflect.ownKeys(c)[0]]);//0
```

3.4. Uso del carácter

Este mecanismo es experimental en la versión del 2020, por lo que todavía no está soportado en todos los navegadores. No debemos utilizar funcionalidades experimentales hasta que no haya una amplia aceptación en las diferentes versiones de los navegadores.

Como la primera versión del prefijo con el guion bajo, se trata de añadir como prefijo el carácter # a cada uno de los campos privados del objeto, declarándolos en el cuerpo de la clase, para que sean tratados de manera especial por el intérprete para hacerlo inaccesible, invisible e inmutable.

En el siguiente código hacemos uso del carácter #. Debemos notar que antes de poder hacer uso de los campos privados, tenemos haberlos declarado, por lo que se sitúan al comienzo del cuerpo de la clase.

```
class Countdown {
  #counter = 0;
  #action = null;
  constructor(counter, action) {
      this.#counter = counter;
      this.#action = action;
  }
  dec() {
      if (this.#counter < 1) return;
      this.#counter--;
      if (this.#counter === 0) {
        this.#action();
      }
  }
}
let c = new Countdown(3,()=>console.log("El contador ha parado."));
c.dec();
c.dec();
c.dec(); //El contador ha parado
```

Podemos declarar propiedades *getter* y *setter* para los campos privados, pero deben ser definidas en el cuerpo de la clase. La ventaja respecto a la versión ES5 es que estas propiedades son heredadas desde el prototipo de la clase.

```
get counter(){
  return this.#counter;
}
```

4. Ampliación de subclases

Vamos a ver algunos ejemplos de uso de subclases.

4.1. Extendiendo objetos estándar

Podemos crear clases que extiendan la funcionalidad de los objetos estándar de JavaScript. El siguiente ejemplo hemos creado una clase que amplía la funcionalidad de los tipos de objetos array, implementando dos propiedades para obtener el primer y el último elemento de un array.

La nueva clase creada, además de las nuevas propiedades, hereda todas las funcionalidades de los objetos `Array`.

```
class EZArray extends Array {
  get first() { return this[0]; }
  get last() { return this[this.length-1]; }
}
let a = new EZArray();
console.log(a instanceof Array); //true
a.push("a","b","c","d");
console.log(a.pop()); //d
console.log(a.first); //a
console.log(a.last); //c
console.log(EZArray.isArray(a)); //true
```

En el código anterior debemos destacar que además de los métodos y propiedades del prototipo, las propiedades y métodos estáticos también son heredados gracias a `extends`, por eso podemos hacer uso del método *EZArray.isArray()*, lo que versiones previas a ES6 no podíamos realizar.

4.2. Mezcla de clases

JavaScript solamente admite la *herencia simple* entre objetos de forma estándar. Al igual que hicimos en la versión ES5 con la invocación de múltiples constructores, vamos a saltarnos esta limitación utilizando la técnica denominada **mezcla de clases** (mixin classes), la cual está basada en convertir clases en factorías para las subclases.

El siguiente código muestra dos factorías de subclases. Las funciones arrow devuelven una clase que extienden a una superclase denominada *Sup*.

```
const S1 = (Sup) => class extends Sup { /*...*/ };
const S2 = (Sup) => class extends Sup { /*...*/ };
```

Podemos crear una clase que extienda de ambas factorías creando una herencia múltiple de clases en base a ellas. La clase *C* es una clase que extiende de *S2*, que a su vez extiende de *S1*, que ha sido creada extendiendo la clase *Object*.

```
class C extends S2(S1(Object)) {
  /*...*/
}
```

Veamos un ejemplo de implementación de la herencia múltiple. Creamos una clase *Hobbyist* como primera clase factoría de subclase. Tenemos una propiedad *hobby* y los métodos para mantener los materiales necesarios para practicarlo. Hemos utilizado campos privados.

```
const Hobbyist = (Sup) => class extends Sup {
  #materials = [];
  #hobby;
  set hobby(value){
    this.#hobby = value;
  }
  get hobby(){
    return this.#hobby;
  }
  addMaterial(material){
    this.#materials.push(material);
    return this;
  }
  materials(){
    return `${this.hobby}: ${this.#materials.toString()}`;
  }
}
```

La segunda clase factoría es la clase *Person*. En esta clase estamos utilizando un constructor, el cual no puede definir propiedades locales con `this` hasta que no sea invocado el superconstructor, de ahí que utilicemos `super()` antes de la definición de las propiedades locales.

```
const Person = (Sup) => class extends Sup {
  constructor (firstName, lastName, age){
    super();
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }
}
```

Por último, creamos la clase con herencia múltiple. La clase *Student* invoca al superconstructor de *Person* y añade la propiedad local *course*, así como un método *toString()* para transformar el objeto en un *string*.

```
class Student extends Person(Hobbyist(Object)){
  constructor (firstName, lastName, age, course){
    super(firstName, lastName, age);
    this.course = course;
  }
  toString(){
    return this.firstName + " " + this.course + " " + this.hobby;
  }
}
```

Testeamos la herencia a través de un objeto *Student* y la invocación de sus métodos.

```
let s1 = new Student("John","Doe",23,"2020-2021");
s1.hobby = "scuba";
s1.addMaterial("mask").addMaterial("fins").addMaterial("regulator");
console.log(s1.toString()); // John 2020-2021 scuba
console.log(s1.materials()); // scuba: mask,fins,regulator
```

4.3. Composición de clases

Podemos crear una nueva clase a partir de la composición de otras clases. Según la teoría de programación orientada a objetos, deberíamos favorecer la composición sobre la herencia de objetos.

En este ejemplo, en lugar de crear una subclase que herede de `Set`, vamos a crear una clase *Histogram* que defina una API similar a `Set`, pero implementando sus métodos delegando a un objeto `Map` interno.

Esta clase realiza un seguimiento de cuántas veces se ha agregado un valor. El método *add()* añade una entrada al histograma, o le suma una unidad si ya existe. El método *count()* devuelve el número de veces que hemos agregado una entrada en el histograma. El siguiente código realiza la implementación, así como su testeo.

```
class Histogram {
  // Constructor con propiedad de tipo Map.
  constructor() { this.map = new Map(); }

  // Devuelve el número de ocurrencias de una clave.
  count(key) { return this.map.get(key) || 0; }

  // Indica si la clave está en el histograma a partir del número de entr
adas.
  has(key) { return this.count(key) > 0; }

  // Nos devuelve el número de entradas del histogragama.
  get size() { return this.map.size; }

  // Incrementa en 1 una clave del histograma.
  add(key) { this.map.set(key, this.count(key) + 1); return this; }

  // Decrementa una clave del histograma.
  delete(key) {
      let count = this.count(key);
      if (count === 1) {
          this.map.delete(key);
      } else if (count > 1) {
          this.map.set(key, count - 1);
      }
      return this;
  }
```

```
  // Iteradores del histograma.
  keys() { return this.map.keys(); }
  values() { return this.map.values(); }
  entries() { return this.map.entries(); }
}

let h = new Histogram();
h.add("Tablet").add("Tablet").add("Tablet").add("Tablet"); // Tablet: 4
h.add("Laptop").add("Laptop").add("Laptop"); // Laptop: 3
h.add("Camera"); // Camera: 1
h.add("SmartPhone").add("SmartPhone"); // SmartPhone: 2
for (let [k,v] of h.entries()){
  console.log(k + ": " + v);
}
```

4.4. Clases abstractas y propiedad `new.target`

En la versión ES6, disponemos de la propiedad `new.target` utilizada para comprobar si un constructor ha sido invocado con el operador `new`. La propiedad devuelve una referencia al constructor sobre el que se ha invocado el operador `new`. Podemos utilizar esta propiedad para chequear si la función invocadora es la clase abstracta, en ese caso lanzaríamos una excepción.

Con la propiedad `new.target` reducimos la complejidad de la declaración de clase abstracta, ya que no será necesario implementar el cerramiento con el cerrojo entre las clases.

La clase *AbstractSet* la definimos como abstracta, por esa razón, en el constructor realizamos el chequeo de `new.target` comparando con la clase *AbstractSet* para lanzar la excepción. Esta clase solo dispone de un único método que es abstracto para que se implemente en la subclase.

```
class AbstractSet {
  constructor(){
    if(new.target === AbstractSet) throw new Error("AbstractSet is an
abstract class.");
  }
  // Método abstracto sin implementación.
  has(x) { throw new Error("Abstract method"); }
}
```

Creamos una clase concreta *NotSet* en el cuerpo de la función anónima. Ya no es necesario el uso de cerrojos. Esta clase verifica si un elemento se encuentra o no en el conjunto inicial. En esta clase debemos implementar el método *has()*.

```
// Clase concreta que extiende AbstractSet.
class NotSet extends AbstractSet {
  constructor(set) {
    super();
    this.set = set;
  }
  // Implementación del método heredado.
  has(x) { return !this.set.has(x); }
  // Sobrescribimos toString()
  toString() { return `{ x| ${this.set.toString()}`; }
}
```

Añadimos una segunda clase concreta *RangeSet* que verifica si un elemento está entre los límites de un rango.

```
// Clase concreta que extiende AbstractSet.
class RangeSet extends AbstractSet {
  constructor(from, to) {
    super();
    this.from = from;
    this.to = to;
  }
  has(x) { return x >= this.from && x <= this.to; }
  toString() { return `{ x | ${this.from} ≤ x ≤ ${this.to} }`; }
}
```

Testeamos las clases creadas.

```
let range = new RangeSet(1,5);
console.log(range.has(3)); //true
console.log(range.has(-3)); //false
try {
  let abstract = new AbstractSet();
} catch(error){
  // Error: AbstractSet is an abstract class.
  console.log(error.toString());
}
```

5. Ejercicios de clases ES6

Implementación de la clase

Implementa una clase *Range* que reciba como argumentos *from* y *to* para definir un rango. Implementa un método *includes()* que dado un entero devuelva un booleano informando si pertenece o no al rango.

Generar un método *toString()* que transforme el objeto en un *string* representativo de sus propiedades.

Herencia de clase

Crea una clase *Span* que inicialice el rango a partir del punto inicial y la longitud de dicho rango.

Método estático

Añade un método estático *parse()* a *Range* que reciba un *string* con el formato "(x…y)" y devuelva un objeto *Range* con el rango definido.

Solución

```
class Range {
  constructor(from, to) {
    // Límites del rango
    this.from = from;
    this.to = to;
  }
  // Devuelve si un valos está en el rango.
  includes(x) { return this.from <= x && x <= this.to; }
  // Representación en string
  toString() { return `(${this.from}...${this.to})`; }
  // Método estático para crear Range
  static parse(s) {
    let matches = s.match(/^\((\d+)\.\.\.(\d+)\)$/);
    if (!matches) {
        throw new TypeError(`Cannot parse Range from "${s}".`)
    }
    // Instanciamos el Range en base a los paréntesis de la expresión reg
ular
    return new Range(parseInt(matches[1]), parseInt(matches[2]));
  }
}

class Span extends Range {
  constructor(start, length) {
    if (length >= 0) {
        super(start, start + length);
    } else {
        super(start + length, start);
    }
  }
}
```

```
let r1 = new Range(1,3);   // Create a Range object
console.log(r1.includes(2)); //true
console.log(r1.toString()); //(1...3)
let s = new Span(2,5);
console.log(s.toString()); //(2...7)
let r2 = Range.parse("(3...8)");
console.log(r2.toString()); //(3...8)
```

6. Protocolos de Iteración

La versión ES6 añade una serie de protocolos para que puedan ser implementados por cualquier tipo de objeto, lo que permite que puedan trabajar de forma estandarizada.

6.1. Protocolo iterable

Este protocolo permite **definir el comportamiento de iteración de los objetos**, es decir, la forma en la que vamos a recuperar la información que contiene para poder ser secuenciada. Un ejemplo es el uso de una sentencia `for..of` para recorrer un `Array`, el cual tiene un comportamiento de iteración por defecto, al igual que ocurre con `Map` o `Set`, sin embargo, con `Object` no tenemos este comportamiento por defecto, por lo que este comportamiento lo debemos implementar nosotros utilizando el protocolo.

Para transformar un objeto como iterable debe implementar el método `@@iterator`, lo que implica que el objeto dispone o hereda una propiedad `Symbol.iterator` que contiene una función que devuelve el objeto iterador. Más adelante veremos cómo hacer un objeto iterable personalizado.

6.2. Protocolo iterador

Permite **producir la secuencia de valores de un objeto que es iterable**. Un objeto iterador debe implementar el método `next()` el cuál devuelve un objeto literal con dos propiedades:

- `value`: contiene el siguiente valor de la secuencia de valores del objeto iterable con el que estamos trabajando.
- `done`: es un booleano que devuelve `false` si no hemos terminado la secuencia y `true` si la hemos finalizado.

En el siguiente ejemplo vemos cómo podemos obtener un iterador para un array a partir de la propiedad `Symbol.iterator`.

```
let arr = [1,'a',3,'d',5];
let arrIterator = arr[Symbol.iterator]();
console.log(arrIterator.next()); // { value: 1, done: false }
console.log(arrIterator.next()); // { value: 'a', done: false }
console.log(arrIterator.next()); // { value: 3, done: false }
console.log(arrIterator.next()); // { value: 'd', done: false }
console.log(arrIterator.next()); // { value: 5, done: false }
console.log(arrIterator.next()); // { value: undefined, done: true }
```

A partir del mismo array podemos iterar también con una sentencia `for..of`, siempre y cuando reiniciemos el iterador.

```
arrIterator = arr[Symbol.iterator]();
for (let value of arrIterator){
    console.log(value);
}
```

En realidad, la sentencia `for..of` se encarga de iterar mientras la propiedad `done` sea `false`. El siguiente ejemplo es equiparable al que acabamos de implementar, donde nosotros buscamos la condición de salida y vamos recogiendo el siguiente valor de la secuencia con `next()`.

```
arrIterator = arr[Symbol.iterator]();
let item = arrIterator.next();
while (!item.done){
    console.log(item.value);
    item = arrIterator.next();
}
```

Muchos objetos que implementan los protocolos tienen métodos específicos para obtener el iterador. Por ejemplo, el método `entries()` de un array.

```
for (let value of arr.entries()){
    console.log(value);
}
```

Otros ejemplos de iteración con otros tipos de objetos lo mostramos en el siguiente ejemplo.

```
let mapExample = new Map();
mapExample.set(1,'a');
mapExample.set(2,'b');
mapExample.set(3,'c');
let mapIterator = mapExample[Symbol.iterator]();
for (let value of mapIterator){
    console.log(value);
}
let str = "Quijote";
let strIterator = str[Symbol.iterator]();
for (let value of strIterator){
    console.log(value);
}
```

Por último, algunos objetos *iterador* pueden ser a su vez *iterables*. En el siguiente ejemplo vemos como obtenemos un iterador del array del ejemplo, y vemos como tiene implementada la propiedad `Symbol.iterator`.

```
let iterator = arr.entries();
console.log(iterator.toString()); // [object Array Iterator]
console.log(iterator === iterator[Symbol.iterator]()); // true
```

6.3. Personalizando iterables

La estructura de un iterador en ES6 es la misma que para uno en ES5, pero en lugar de devolver directamente el iterador, debemos empaquetarlo en un objetor iterable, es decir, un objeto con la propiedad `Symbol.iterator`. Esta propiedad es una función que devolverá el objeto

iterador con la implementación del método `next()` que es obligatorio. Esta estructura tiene la siguiente forma:

```
const iterable = {
// El objeto iterable contiene una propiedad Symbol.iterator
  [Symbol.iterator]() { // Symbol.iterator es una función que devuelve el
iterador
      const iterator = { // El objeto iterador implementa next()
obligatoriamente
          next() {
            if (!condition) {
              return {value: 'value', done: false}
            } else {
              return {value: undefined, done: true}
            }
          }
      };
      return iterator;
  }
};
```

Veamos un ejemplo implementado. La función *myIterable* devuelve un objeto iterable para implementar un contador desde un número inicial y hasta un límite en el que finalizaría la cuenta. La función devuelve el objeto iterable sombre el que podemos conseguir seguir la cuenta del contador.

```
function myIterable(start, finish) {
  // Iniciamos variables del closure
  let index = start;
  let count = 0;

  return { // Devolvemos objeto iterable con la propiedad Symbo.iterator
    [Symbol.iterator]() { // Propiedad con la función que devuelve el
iterador
      return { // Devolvemos objeto iterador con método next()
        next() {
          let result;
          // No hemos terminado de iterar
          if (index < finish) {
            // value le asignamos el índice, done es false porque no
hemos termiando.
            result = { value: index, done: false };
            index++;
            count++;
          } else { // Condición de salida
            // done es true, value le asignamos la cuenta.
            result = { value: count, done: true }
          }
          return result;
        }
      }
    }
  }
}
```

La ventaja que tenemos con ES6 es que se trata de una estructura estandarizada. Un bucle `for..of` puede recibir un objeto iterable para extraer su iterador y recuperar la secuencia de valores del objeto iterable.

```
let iterable = myIterable(2, 5);
for (let value of iterable){
  console.log(value);
}
```

Una estructura habitual es hacer que un iterable sea también un iterador. Para hacer esto, el objeto iterable debe implementar el método `next()`, además de la propiedad `Symbol.iterator`. La función de esta propiedad debe devolver la referencia `this` como objeto iterador.

```
const iterable = {
    [Symbol.iterator]() { // La propiedad devuelve una referecia al
propio objeto como iterador
        return this;
    },
    next() { // El objeto iterable implementa next(), por lo que es un
iterador
        if (!condition) {
            return {value: 'value', done: false};
        } else {
            return {done: true};
        }
    },
};
```

Vamos a reimplementar la función *myIterable* para que devuelva un objeto iterable e iterador.

```
function myIterable(start, finish) {
  // Iniciamos variables del closure
  let index = start;
  let count = 0;

  return { // Devolvemos objeto iterable con la propiedad Symbo.iterator
    [Symbol.iterator]() { // La propiedad devuelve una referecia al
propio objeto como iterador
      return this;
    },
    next() {
      let result;
      // No hemos terminado de iterar
      if (index < finish) {
        // value le asignamos el índice, done es false porque no hemos
termiando.
        result = { value: index, done: false };
        index++;
        count++;
      } else { // Condición de salida
        // done es true, value le asignamos la cuenta.
        result = { value: count, done: true }
      }
      return result;
    }
  }
}
```

Al tratarse de un objeto iterador e iterable, podemos utilizar el iterador en un bucle `for..of`.

```
let iterable = myIterable(2, 5);
const iterator = iterable[Symbol.iterator]();
console.log(iterator === iterable); // true
console.log('Iteración en for..of');
for (let value of iterator){
  console.log(value);
}
```

Al ser el objeto iterable e iterador podemos romper la estructura de un bucle, pudiendo volver a iterar más tarde en el mismo punto donde lo habíamos dejado.

```
let iterable2 = myIterator(2, 5);
for (let value of iterable2){
  console.log(value);  // 2
  break;
}
console.log('Cotinuamos en un nuevo bucle');
for (let value of iterable2){
  console.log(value);  // 3, 4
}
```

6.4. Métodos opcionales en un iterador

Un iterador tiene dos métodos opcionales:

- `return()`: Si un iterador finaliza prematuramente dentro de un bucle, se invocará este método para limpiar ese estado final. Un ejemplo es si estuviéramos procesando las líneas de un fichero a través de un iterador, querríamos cerrar el fichero ante una ruptura del bucle prematura. Este caso, aunque es visual para la funcionalidad, es poco representativo para ejecutar en el navegador por no trabajar normalmente con ficheros.
- `trow()`: utilizado en generadores.

Sobre el último ejemplo de la función *myIterable* vamos a añadir el método return() que reinicialice el contador en el caso de romper el bucle por algún motivo.

```
return() {
  console.log('Reiniciamos el contador en return');
  index = start;
  return this;
}
```

Ahora cuando rompemos el bucle, el iterador vuelve a comenzar desde el principio.

```
let iterable = myIterator(2, 5);
for (let value of iterable){
  console.log(value); // 2
  break;
}
for (let value of iterable){
  console.log(value);  // 2, 3, 4
}
```

Volver a destacar que no todos los iteradores implementan este método.

6.5. Ejemplo de implementación. Clase List

Actualizamos el ejemplo de la versión ES5 para la clase *List* que habíamos implementado. En primer lugar, creamos la clase en la versión ES6. La clase tiene un campo privado con el array que mantiene los elementos, no necesita constructor y tiene dos métodos públicos para añadir y borrar elementos de una instancia.

```
class List {
  //Campo privado
  #list = [];

  // Método para añadir un nuevo objeto en la lista
  add (obj) {
    this.#list.push(obj);
    return this.#list.length;
  }

  // Método para borrar un objeto de la lista.
  remove (position) {
    if (position >= this.#list.length || position < 0)
      throw new Error("Intento de borrado fuera del límite de la
lista.");

    return this.#list.splice(position, 1);
  }
}
```

Pasamos a implementar la propiedad de acceso *elements* que devolverá un objeto iterable. Como podemos ver en el código, la propiedad guarda en una variable local denominada *array* la referencia al campo privado para que pueda estar accesible en el closure. La propiedad devuelve el objeto iterable ya que implementa la propiedad `Symbol.iterator` que nos permite construir el objeto iterador. En esta función inicializamos el índice por el que recorreremos el array, variable *nextIndex* y que estará accesible en el closure, e implementamos el método `next()` devolviendo la posición del array al que apunte el índice. Notar como hacemos el postincremento en el índice para que esté actualizado en la siguiente invocación.

```
get elements(){
  // referencia para habilitar el closure en el objeto
  let array = this.#list;

  return { // Objeto iterable
    [Symbol.iterator]() { // Propiedad con la función que devuelve el
iterador
      let nextIndex = 0; // Inicialización del índice para cada iterador
      return {
        next: function () {
          return nextIndex < array.length ?
            { value: array[nextIndex++], done: false } :
            { done: true };
        }
      }
    }
  }
}
```

Instanciamos un objeto *List* y le añadimos elementos en forma de objeto literal.

```
let list = new List();
//Añadimos elementos a la lista
list.add({ property1: "obj1", property2: 1 });
list.add({ property1: "obj2", property2: 2 });
list.add({ property1: "obj3", property2: 3 });
```

Obtenemos el iterable de la lista, y de él podemos recuperar un iterador para utilizarlo en un bucle `while`.

```
let iterable = list.elements; // Obtenemos el iterable de la lista.
const iterator = iterable[Symbol.iterator](); // Obtenemos iterador del
iterable
let obj = iterator.next();
while (!obj.done) { // Iteramos con while
  let element = obj.value;
  console.log("Objeto: " + element.property1 + " " + element.property2);
  obj = iterator.next();
}
```

Del objeto iterable también lo podemos utilizar para recorrer la lista con un bucle `for..of`.

```
for (let element of iterable){ // Iteramos sobre el iterable
  console.log("Objeto: " + element.property1 + " " + element.property2);
}
```

Un detalle importante es que cada vez que obtenemos el objeto iterador, la variable índice se inicializa a 0, por lo que los nuevos iteradores comienzan a recuperar elementos desde el principio. Si hubiéramos declarado el índice en la propiedad de acceso *elements*, en el primer iterador solo funcionaría una única vez, lo que nos obligaría a instanciar un nuevo objeto iterable.

7. Generadores

El uso de los iteradores puede resultar tedioso en ciertas partes de nuestro código, además de hacerlo poco legible. La solución pasa por el uso de generadores.

7.1. Definición

Los generadores son funciones que podemos pausar en un momento determinado, y retomarlas más adelante en el mismo punto donde las habíamos dejado. En cada pausa podremos recibir valores de la función, así como pasarle valores a la función.

La sintaxis para crear un generador es incluir un carácter * después de la palabra reservada `function`. La función podría ser también anónima. La función *datesGenerator* declara una función generadora. En el cuerpo vemos que genera por la consola un mensaje con la fecha actual.

```
function* datesGenerator(){
  console.log(Date.now());
}
```

Si invocamos la función el resultado no es el que a priori estamos esperando, ya que **el cuerpo de la función no se ejecuta**.

```
datesGenerator(); // El cuerpo de la función no se ejecuta
```

La función generadora devuelve un objeto iterador e iterable para que podamos iterar sobre ella, por tanto, para obtener el resultado invocamos el método `next()` sobre el iterador.

```
let it = datesGenerator(); // Obtenemos un iterador e iterable
// Ejecuta el cuerpo del generador mostrando fecha por consola
it.next(); // 1640085688698
```

En la definición del generador hablábamos de pausar la ejecución de la función para retomarla a partir del punto donde lo habíamos dejado. La palabra reservada `yield` detiene la ejecución de la función, para que en la próxima invocación de `next()` retomemos justo en el punto donde lo habíamos dejado.

Modificamos nuestro generador para detener la ejecución del código. En la primera pasada el generador se detiene en `yield`, en la segunda pasada continuamos a partir del punto donde lo habíamos dejado.

```
function* datesGenerator(){
  console.log('Primera pasada');
  console.log(Date.now()); // 1640089550695
  yield;
  console.log('Segunda pasada');
  console.log(Date.now()); // 1640089550695
}

let it = datesGenerator(); // Obtenemos un iterador e iterable
it.next(); // Primera pasada
it.next(); // Segunda pasada
```

Los generadores pueden también declararse como métodos de objetos literales o detro una clase.

```
const obj = {
    * generatorMethod() {
        ...
    }
};
const genObj = obj.generatorMethod();

class MyClass {
    * generatorMethod() {
        ...
    }
}
const myInst = new MyClass();
const genObj = myInst.generatorMethod();
```

7.2. Generador con función de iterador

Un generador puede actuar en una aplicación **produciendo datos** y también **consumiendo datos**. Si el generador tiene el papel de productor actuará implementando los interfaces `Iterable` e `Iterator` que hemos visto en la sección anterior.

Vamos a transformar la función *datesGenerator* para que reciba como argumento el número de fechas que queremos generar. La función generadora va a "producir" un objeto `Date` con la fecha actual y lo va a retornar utilizando `yield`. Este objeto lo vamos a "consumir" a través de iterador obtenido del generador con su método `next()`.

Como observamos en el código de *datesGenerator*, hemos definido una función con un bucle `for`, el cuál se va a detener en cada iteración devolviendo un objeto `Date` con la sentencia `yield`. El bucle se queda pausado hasta que no se produzca la siguiente invocación de `next()` en el iterador.

```
function* datesGenerator(num){
  for (let i = 0; i < num; i++){
    yield Date.now();
  }
}
```

Para consumir los objetos `Date` obtenemos un iterador en base a la función generadora, y con su método `next()` obtenemos el objeto empaquetado en un objeto literal con el formato que dicta el protocolo, propiedad `value` con el valor y propiedad `done` para saber si hemos terminado. En el ejemplo vemos como iteramos sobre el generador.

```
console.log("Obtenemos iterable");
let it = datesGenerator(5); // Obtenemos un iterador e iterable
console.log ("Obtenemos fecha");
console.log (it.next()); // {value: 1641984904090, done: false}
console.log ("Obtenemos fecha");
console.log (it.next()); // {value: 1641984904090, done: false}
console.log ("Obtenemos fecha");
console.log (it.next()); // {value: 1641984904090, done: false}
console.log ("Obtenemos fecha");
```

```
console.log (it.next()); // {value: 1641984904090, done: false}
console.log ("Obtenemos fecha");
console.log (it.next()); // {value: 1641984904090, done: false}
console.log ("Hemos finalizado");
console.log (it.next()); // {value: undefined, done: true}
```

De manera opcional, podemos utilizar el valor final del generador si lo consideramos necesario. Vamos a modificar el generador para devolver la cantidad total de objetos `Date` generados, para ello debemos devolver este valor con una sentencia `return`, lo hacemos justo después de finalizar el bucle del generador.

```
return num;
```

Si volvemos iterar sobre el generador, el último valor obtenido será el siguiente.

```
console.log ("Hemos finalizado");
console.log (it.next()); // {value: 5, done: true}
```

Nota importante: Solamente podemos utilizar la sentencia yield estáticamente, por lo que **no podremos utilizarla dentro de una función de callback**, ya que obtendremos un error de sintaxis. El siguiente código no se puede utilizar.

```
function* genFunc() {
  ['a', 'b'].forEach(x => yield x); // Error de sintaxis
}
```

7.3. Iterando sobre un generador

Iterar sobre un generador es exactamente igual que hacerlo con un iterador. Siguiendo con la función generadora de fechas, vamos a iterar utilizando diferentes métodos.

Una forma habitual es hacerlo con un bucle `for..of`. El inconveniente con este tipo de iteración es que perdemos el valor del último retorno, ya que es consumido internamente por el bucle.

```
let it = datesGenerator(5); // Obtenemos un iterador e iterable
for (const d of it) {
  console.log (d); // 1641984904090
}
console.log ("Perdemos el valor final");
console.log (it.next()); // {value: undefined, done: true}
```

En caso de implementar el último valor podríamos utilizar un bucle `while`.

```
let it2 = datesGenerator(5);
let elem = it2.next();
while (!elem.done){
  console.log (elem.value); // 1641984904090
  elem = it2.next();
}
console.log (elem); // {value: 5, done: true}
```

Otro ejemplo es utilizar el operador de propagación.

```
const arr = [...datesGenerator(3)];
console.log (arr); // [1641984904090, 1641984904090, 1641984904090]
```

7.4. Comunicación con el generador

La sentencia `yield` nos permite comunicarnos de forma bidireccional con el generador. Además de devolver los valores sobre los que estamos iterando, también permite recibir argumentos desde el método `next()`.

Vamos a transformar nuestro generador *datesGenerator* para que, en cada pausa, el generador reciba como argumento la cantidad de milisegundos en la que vamos incrementar la fecha actual. Como vemos en el código, la sentencia yield se asigna a una variable amount, la cuál será utilizada para incrementar la fecha actual. Utilizamos una expresión cortocircuito para asignar un valor por defecto al argumento recibido.

```
function* datesGenerator(num){
  for (let i = 0, amount = 0; i < num; i++){
    amount = (yield Date.now() + amount) || 0;
  }
  return num;
}
```

En cada invocación del método `next()` pasamos como argumento la cantidad de milisegundo a incrementar, que será un múltiplo de 10 en cada iteración.

```
let it = datesGenerator(5); // Obtenemos un iterador e iterable
let elem = it.next();
let i = 0;
while (!elem.done){
  console.log (elem.value); // Fecha incrementada por un múltiplo de 10
  elem = it.next(++i * 10);
}
console.log (elem);
```

7.5. Lanzar excepciones al generador

Podemos pasar una excepción al generador para que detenga su ejecución. El método `throw()` del iterador nos permite lanzar una excepción lo que obliga al generador a finalizar su ejecución.

En el generador *datesGenerator* anidamos el bucle en un bloque `try-catch` que nos permita capturar una excepción, en este caso solamente mostramos el mensaje de la excepción en la consola.

```
function* datesGenerator(num){
  try {
    for (let i = 0, amount = 0; i < num; i++){
      amount = yield Date.now() + amount || 0;
    }
  } catch (error){
    console.log(error.message);
  }
  console.log('Antes de finalizar el generador.');
  return num;
}
```

Al iterar sobre el generador añadimos una condición, cuando el contador alcanza el valor "2" en lugar de seguir iterando lanzaremos una excepción al generador con el método `throw()`, esta excepción es capturada en el bloque `catch` del generador, finalizando su ejecución, ya que no atenderá la sentencia `return` que dispone. El bucle finaliza, pero el generador devuelve un objeto con la propiedad `done` con valor `true`, pero si asignar ningún valor en la propiedad `value`.

```
while (!elem.done){
  console.log(elem.value); // Fecha actual incrementada por un múltiplo
de 10
  if (i === 2)
    it.throw(new Error('Error: Alcanzado el límite de 2.')); // Error:
Alcanzado el límite de 2.
  elem = it.next(++i * 10);
}
console.log(elem); // {value: undefined, done: true}
```

7.6. Ejemplo de implementacion. Clase List

Repetimos el ejemplo con la clase *List*, pero vamos a recorrer los objetos almacenados utilizando un generador. Los generadores no pueden ser asignados a propiedades de acceso, por lo que debemos transformar la propiedad en un método. En el siguiente código vemos cómo crear un método en una clase que sea un generador.

```
* elements(){
  for (const obj of this.#list){
    yield obj;
  }
}
```

Para recorrer una lista instanciada accedemos al método para que nos devuelva un iterador.

```
let list = new List();
list.add({ property1: "obj1", property2: 1 });
list.add({ property1: "obj2", property2: 2 });
list.add({ property1: "obj3", property2: 3 });

// Iteramos sobre la lista.
for (let element of list.elements()){ // Iteramos sobre list
  console.log("Objeto: " + element.property1 + " " + element.property2);
}
```

Otra posible solución es hacer que *List* sea iterable directamente, para ello podemos implementar la función, o en este caso el método `[Symbol.iterator]()` para que actúe como un generador. El siguiente código muestra cómo hacer *List* iterable mediante un generador.

```
* [Symbol.iterator]() {
  for (const obj of this.#list){
    yield obj;
  }
}
```

Ahora podemos utilizar directamente una instancia de *List* como objeto iterable en un bucle para recuperar su contenido.

```
for (let element of list){ // Iteramos sobre el iterable
  console.log("Objeto: " + element.property1 + " " + element.property2);
}
```

7.7. Generador recursivo

Podemos estar en la circunstancia de que un generador, además de sus propios objetos, necesite generar objetos de otro generador, es decir, disponemos de un generador principal o externo, y otro secundario o interno. Al ejecutar el generador externo queremos generar también los objetos del interno.

La siguiente función generador permite obtener dos datos ya que tenemos dos sentencias `yield`.

```
function* internalGenerator() {
  yield 'internal: 1';
  yield 'internal: 2';
}
```

Creamos un segundo generador que, además de generar datos propios, debe generar los datos del generador anterior. En lugar de iterar sobre el generador interno la sentencia `yield*` nos permite recuperar sus datos de forma recursiva sin necesidad de utilizar un bucle.

```
function* externalGenerator() {
    yield 'external: a';
    yield* internalGenerator();
    yield 'external: b';
}
```

Al iterar sobre este segundo generador obtenemos los objetos generados por ambos.

```
for (let element of externalGenerator()){
  console.log(element); // external: a,  internal: 1,  internal:
2,  external: b
}
```

8. Ejemplo de implementación. Carrito de la compra con objetos ES6

Vamos a rediseñar el carrito de la compra para que haga uso de objetos basados en la versión ES6. Para guardar la compatibilidad con la versión anterior queremos implementar los objetos mediante el uso de campos privados. Como método de encapsulado de campos privados vamos a utilizar el método experimental de **carácter #**. No tenemos que olvidar que, al ser experimental, este método no está implementado en todos los navegadores, por lo que **no sería una solución para una aplicación actual**. En caso de necesitar encapsulación en una aplicación actual, o bien deberíamos utilizar la versión ES5, o implementar los métodos con `WeakMap` o con `Symbol`.

Rediseñamos los objetos por pasos.

8.1. Excepciones genéricas

Para crear una clase que extienda de `Error` debemos:

1. Llamar al constructor de `Error` en base a los parámetros que define. Básicamente necesitaríamos el mensaje para definir la excepción. Lo definimos con valor por defecto vacío.
2. Asignar la propiedad *name* con el nombre de la clase. Es el mismo caso que con ES5.
3. Comprobamos si la clase `Error` dispone de método *captureStackTrace()* para mantener el seguimiento adecuado de la pila de llamadas.

```
class BaseException extends Error {
  constructor (message = "", fileName, lineNumber){
    super(message, fileName, lineNumber);
    this.name = "BaseException";
    if (Error.captureStackTrace) {
      Error.captureStackTrace(this, BaseException)
    }
  }
}
```

El resto de las excepciones genéricas pueden heredar de *BaseException*. En este caso personalizamos un mensaje para la excepción, y guardamos las propiedades que específicas de cada excepción como el parámetro, el valor o el nombre de clase.

```
//Excepción acceso inválido a constructor
class InvalidAccessConstructorException extends BaseException {
  constructor (fileName, lineNumber){
    super("Constructor can’t be called as a function.", fileName, lineNum
ber);
    this.name = "InvalidAccessConstructorException";
  }
}
```

```
//Excepción personalizada para indicar valores vacios.
class EmptyValueException extends BaseException {
  constructor (param, fileName, lineNumber){
    super("Error: The parameter " + param + " can't be empty.", fileName,
 lineNumber);
    this.param = param;
    this.name = "EmptyValueException";
  }
}

//Excepción de valor inválido
class InvalidValueException extends BaseException {
  constructor (param, value, fileName, lineNumber){
    super(`Error: The paramenter ${param} has an invalid value. (${param}
: ${value})`, fileName, lineNumber);
    this.param = param;
    this.name = "EmptyValueException";
  }
}

//Excepción personalizada para clases abstractas.
class AbstractClassException extends BaseException {
  constructor (className, fileName, lineNumber){
    super(`Error: The class  ${className} is abstract.`, fileName, lineNu
mber);
    this.className = className;
    this.name = "AbstractClassException";
  }
}
```

8.2. Clase *Product*

La clase *Product* queremos mantener que sea **abstracta**, utilizamos para ello la propiedad `new.target`. Empezaremos declarando la clase y acto seguido los campos privados que vamos a utilizar.

En el constructor, la primera acción es verificar si hemos invocado con el operador `new` mediante la propiedad `new.target`, para continuar verificando la clase abstracta comparando con la referencia *Product*, impidiendo con ello instanciar objetos *Product*. El resto de los pasos son los habituales, validación de argumentos y asignación a campos privados.

```
class Product{
  //Campos privados
  #serial;
  #brand;
  #model;
  #price;
  #taxPercentage;
  constructor (serial, brand, model, price, taxPercentage = Product.IVA){
    //La función se invoca con el operador new
    if (!new.target) throw new InvalidAccessConstructorException(); //
Verificación operador new
    if (new.target === Product) throw new
AbstractClassException("Product"); // Chequeo clase abstracta

    //Validación de parámetros obligatorios
    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",
price);
    if (!taxPercentage || taxPercentage < 0) throw new
InvalidValueException("taxPercentage", taxPercentage);

    //Definición de atributos privados del objeto
    this.#serial = serial;
    this.#brand = brand;
    this.#model = model;
    this.#price = price;
    this.#taxPercentage = taxPercentage;
  }

}
```

Añadimos los *getter* y *setter* a las propiedades una vez finalizado el constructor. Cada propiedad debe ser validada antes de modificar su valor. Todas las propiedades creadas son añadidas al prototipo.

```
Object.defineProperty(this, 'serial', {
  enumerable: true,
  get() {
    return this.#serial;
  },
  set(value) {
    if (!value) throw new EmptyValueException('serial');
    this.#serial = value;
  },
});
```

```
Object.defineProperty(this, 'brand', {
  enumerable: true,
  get() {
    return this.#brand;
  },
  set(value) {
    if (!value) throw new EmptyValueException('brand');
    this.#brand = value;
  },
});
Object.defineProperty(this, 'model', {
  enumerable: true,
  get() {
    return this.#model;
  },
  set(value) {
    if (!value) throw new EmptyValueException('model');
    this.#model = value;
  },
});
Object.defineProperty(this, 'price', {
  enumerable: true,
  get() {
    return this.#price;
  },
  set(value) {
    value = Number.parseFloat(value);
    if (Number.isNaN(value) && value > 0) throw new
InvalidValueException('price', value);
    this.#price = value;
  },
});
Object.defineProperty(this, 'taxPercentage', {
  enumerable: true,
  get () {
    return this.#taxPercentage;
  },
  set (value = Product.IVA) {
    if (!value || value < 0) throw new
InvalidValueException('taxPercentage', value);
    this.#taxPercentage = value;
  },
});
```

Creamos una propiedad estática para el valor del IVA, la cual es asignada como propiedad del constructor.

```
static get IVA() {
  return 21;
}
```

El resto de los métodos públicos son añadidos en el prototipo dentro de la clase.

```
get priceWithoutTaxes() {
  return this.price - (this.price * this.taxPercentage / 100);
}

get tax() {
  return this.price * this.taxPercentage / 100;
}

toString() {
  return `Serial: ${this.serial} Brand: ${this.brand} Model:
${this.model} Price: ${this.price}€ Tax: ${this.taxPercentage}%`;
}
```

Para finalizar, tenemos que declarar las propiedades públicas del objeto que no necesiten validación, las cuales asignamos al prototipo.

```
Object.defineProperty(Product.prototype, "description", {enumerable: true
, writable: true});
```

8.3. Clase *Laptop*

Creamos la primera clase que extienda de *Product*. La estructura es similar a la anterior, definimos primero los campos privados específicos de la clase. En el constructor hacemos la validación de la propiedad `new.target` para asegurarnos que hemos invocado el constructor con el operador `new` y llamamos al superconstructor.

El resto de pasos son los mismos, validamos argumentos y asignamos a los campos privados.

```
class Laptop extends Product {
  // Atributos privados
  #processor;
  #memory;
  #hd;
  #size;
  #system;
  constructor(serial, brand, model, price, taxPercentage = Product.IVA,
processor = 'unkonwn', memory = '0GB', hd = '-', size = '0GB', system =
'unkonwn') {
    // La función se invoca con el operador new
    if (!new.target) throw new InvalidAccessConstructorException();
    // Llamada al superconstructor.
    super(serial, brand, model, price, taxPercentage);

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

    // Atributos privados
    this.#processor = processor;
    this.#memory = memory;
    this.#hd = hd;
    this.#size = size;
    this.#system = system;

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

```
    Object.defineProperty(this, 'memory', {
      enumerable: true,
      get() {
        return this.#memory;
      },
      set(value) {
        if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new
InvalidValueException('memory', value);
        this.#memory = value;
      },
    });

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

    Object.defineProperty(this, 'size', {
      enumerable: true,
      get() {
        return this.#size;
      },
      set(value) {
        if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new
InvalidValueException('size', value);
        this.#size = value;
      },
    });

    Object.defineProperty(this, 'system', {
      enumerable: true,
      get() {
        return this.#system;
      },
      set(value) {
        if (!value) throw new EmptyValueException('system');
        this.#system = value;
      },
    });
  }
```

```
  // Métodos públicos
  toString() {
    return `${super.toString()} System: ${this.system} Processor:
${this.processor
    } Memoria: ${this.memory} HD: ${this.hd} Size: ${this.size}`;
  }
}
```

8.4. Resto de subclases

El resto de las subclases siguen la estructura de *Laptop*.

```
// Definimos la subclase Camera
class Camera extends Product {
  // Atributos privados
  #type;
  #resolution;
  #size;
  constructor(serial, brand, model, price, taxPercentage = Product.IVA,
type = '-', resolution = 0, size = 0) {
    // La función se invoca con el operador new
    if (!new.target) throw new InvalidAccessConstructorException();
    // Llamada al superconstructor.
    super(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
InvalidValueException('resolution', resolution);
    if (Number.isNaN(size) || size < 0) throw new
InvalidValueException('size', size);

    // Atributos privados
    this.#type = type;
    this.#resolution = resolution;
    this.#size = size;
```

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

    Object.defineProperty(this, 'resolution', {
      enumerable: true,
      get() {
        return this.#resolution;
      },
      set(value) {
        value = Number.parseFloat(value);
        if (Number.isNaN(value) || value < 0) throw new
InvalidValueException('resolution', value);
        this.#resolution = value;
      },
    });

    Object.defineProperty(this, 'size', {
      enumerable: true,
      get() {
        return this.#size;
      },
      set(value) {
        value = Number.parseFloat(value);
        if (Number.isNaN(value) || value < 0) throw new
InvalidValueException('size', value);
        this.#size = value;
      },
    });
  }

  // Métodos públicos
  toString() {
    return `${super.toString()
    } Tipo: ${this.type} Resolución: ${this.resolution}MP Size:
${this.size}''`;
  }
}
```

```
// Definimos la subclase Smartphone
class Smartphone extends Product {
  // Atributos privados
  #memory;
  #storage;
  #resolution;
  #size;
  constructor(serial, brand, model, price, taxPercentage = Product.IVA,
memory = '0GB', storage = '0GB', resolution = '0x0', size = 0) {
    // La función se invoca con el operador new
    if (!new.target) throw new InvalidAccessConstructorException();
    // Llamada al superconstructor.
    super(serial, brand, model, price, taxPercentage);

    // Validación de argumentos
    if (!/^((\d+GB)|(\d+TB))$/.test(memory)) throw new
InvalidValueException('memory', memory);
    if (!/^((\d+GB)|(\d+TB))$/.test(storage)) throw new
InvalidValueException('storage', storage);
    size = Number.parseFloat(size);
    if (Number.isNaN(size) || size < 0) throw new
InvalidValueException('size', size);
    if (!/^(\d+x\d+)$/.test(resolution)) throw new
InvalidValueException('resolution', resolution);

    // Atributos privados
    this.#memory = memory;
    this.#storage = storage;
    this.#resolution = resolution;
    this.#size = size;

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

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

    Object.defineProperty(this, 'storage', {
      enumerable: true,
      get() {
        return this.#storage;
      },
      set(value) {
        if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new
InvalidValueException('storage', value);
        this.#storage = value;
      },
    });

    Object.defineProperty(this, 'size', {
      enumerable: true,
      get() {
        return this.#size;
      },
      set(value) {
        value = Number.parseFloat(value);
        if (Number.isNaN(value) || value < 0) throw new
InvalidValueException('size', value);
        this.#size = value;
      },
    });
  }

  // Métodos públicos
  toString() {
    return `${super.toString()} System: ${this.system
    } Memoria: ${this.memory} Almacenamiento: ${this.storage} Resolución:
${this.resolution} Size: ${this.size}''`;
  }
}
```

```
// Definimos la subclase Tablet
class Tablet extends Product {
  // Atributos privados
  #memory;
  #storage;
  #resolution;
  #size;
  constructor(serial, brand, model, price, taxPercentage = Product.IVA,
memory = '0GB', storage = '0GB', resolution = '0x0', size = 0) {
    // La función se invoca con el operador new
    if (!new.target) throw new InvalidAccessConstructorException();
    // Llamada al superconstructor.
    super(serial, brand, model, price, taxPercentage);

    // Validación de argumentos
    if (!/^((\d+GB)|(\d+TB))$/.test(memory)) throw new
InvalidValueException('memory', memory);
    if (!/^((\d+GB)|(\d+TB))$/.test(storage)) throw new
InvalidValueException('storage', storage);
    size = Number.parseFloat(size);
    if (Number.isNaN(size) || size < 0) throw new
InvalidValueException('size', size);
    if (!/^(\d+x\d+)$/.test(resolution)) throw new
InvalidValueException('resolution', resolution);

    // Atributos privados
    this.#memory = memory;
    this.#storage = storage;
    this.#resolution = resolution;
    this.#size = size;

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

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

    Object.defineProperty(this, 'storage', {
      enumerable: true,
      get() {
        return this.#storage;
      },
      set(value) {
        if (!/^((\d+GB)|(\d+TB))$/.test(value)) throw new
InvalidValueException('storage', value);
        this.#storage = value;
      },
    });

    Object.defineProperty(this, 'size', {
      enumerable: true,
      get() {
        return this.#size;
      },
      set(value) {
        value = Number.parseFloat(value);
        if (Number.isNaN(value) || value < 0) throw new
InvalidValueException('size', value);
        this.#size = value;
      },
    });
  }

  // Métodos públicos
  toString() {
    return `${super.toString()} System: ` + ` System: ${this.system
    } Memoria: ${this.memory} Almacenamiento: ${this.storage} Resolución:
${this.resolution} Size: ${this.size}''`;
  }
}
```

8.5. Objeto *ShoppingCart*

Realizamos los pasos de implementación del objeto *ShoppingCart*.

1.1.1 Clases de excepción

Declaramos las excepciones específicas como clases. *ShoppingCartException* extiende de *BaseException*. El resto de las excepciones extienden de *ShoppingCartException*.

```
class ShoppingCartException extends BaseException {
  constructor (fileName, lineNumber){
    super("Error: Shopping Cart Exception.", fileName, lineNumber);
    this.name = "ShoppingCartException";
  }
}

class ProductShoppingCartException extends ShoppingCartException {
  constructor (fileName, lineNumber){
    super("Error: The method needs a Product parameter.", fileName, lineN
umber);
    this.name = "ProductShoppingCartException";
  }
}

class ProductNotExistException extends ShoppingCartException {
  constructor (product, fileName, lineNumber){
    super("Error: The product doesn't exist in the cart. " + product.seri
al, fileName, lineNumber);
    this.name = "ProductShoppingCartException";
    this.product = product;
  }
}

class PositionOutBoundsException extends ShoppingCartException {
  constructor (product, fileName, lineNumber){
    super("Error: The position is out of bounds.", fileName, lineNumber);
    this.name = "PositionOutBoundsException";
    this.product = product;
  }
}
```

1.1.2 Patrón Singleton

Definimos la expresión IIFE con la variable para almacenar la instancia del *ShopppingCart* y la función *init()* que permite crear la instancia. La función anónima devuelve el objeto para instanciar *ShopppingCart*.

```
let ShoppingCart = (function () {
  let instantiated;
  function init() { //Inicialización del Singleton
  }
  return {
    // Devuelve un objeto con el método getInstance
    getInstance: function () {
      if (!instantiated) { //Si la variable instantiated es undefined, pr
iemera ejecución, ejecuta init.
        instantiated = init(); //instantiated contiene el objeto único
      }
      return instantiated; //Si ya está asignado devuelve la asignación.
    }
  };
})();
```

Dentro de la función anónima declaramos la clase *ShoopingCart*. La clase tiene dos campos privados para el array de productos y cantidades. El constructor tan solo valida la invocación con el operador `new`.

```
class ShoppingCart {
  //Definición de atributos privados del objeto
  #products = []; //array con los productos del carrito
  #quantities = []; //array con las cantidades de cada producto del carri
to.
  constructor (){
    //La función se invoca con el operador new
    if (!new.target) throw new InvalidAccessConstructorException();
  }
}
```

Los métodos privados son iguales, pero utilizando los campos privados de la clase.

```
//Dado un product, devuelve la posición de ese product en el carrito o -
1 si no lo encontramos.
getProductPosition (product){
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException ();
  }
  return this.#products.findIndex(x => x.serial === product.serial);
}

//Añade un nuevo product en el carrito o incrementa su cantidad si ya exi
ste. La cantidad es opcional.
addProduct (product, quantity = 1){
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException ();
  }
  let position = this.getProductPosition(product);
  if (position === -1){
    this.#products.push(product);
    this.#quantities.push(quantity);
  } else{
    this.#quantities[position] = this.#quantities[position] + quantity;
  }
  return this;
}

//Devuelve el número total de item en el carrito diferentes
getNumberProducts (){
  return this.#products.length;
}

//Devuelve la cantidad de products que se van a comprar.
getQuantityProducts (product){
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException ();
  }
  let position = this.getProductPosition(product);
  if (position === -1){
    throw new ProductNotExistException(product);
  }
  return this.#quantities[position];
}
```

```
//Devuelve la cantidad de products en función de la posición
getQuantityProductPosition (position){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= this.#products.length || position < 0) throw new Positi
onOutBoundsException();
  return this.#quantities[position];
}

//Devuelve la cantidad de products en función de la posición
getProduct (position){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= this.#products.length || position < 0) throw new Positi
onOutBoundsException();
  return this.#products[position];
}

//Modifica un product del carrito si existe, sino lanzamos una excepción
updateProduct (product, quantity = 1){
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException ();
  }
  if (quantity < 0) throw new InvalidValueException("quantity",quantity);
  let position = this.getProductPosition(product);
  if (position !== -1){
    this.#quantities[position] = quantity;
    if (this.#quantities[position] === 0){
      this.#products.splice(position, 1);
      this.#quantities.splice(position, 1);
    }
  } else{
    throw new ProductNotExistException(product);
  }
  return this;
}
```

```
//Modifica un product del carrito si existe por posición, sino lanzamos u
na excepción
updateProductPosition (position, quantity = 1){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= this.#products.length || position < 0) throw new Positi
onOutBoundsException();
  if (quantity < 0) throw new InvalidValueException("quantity",quantity);
  this.#quantities[position] = quantity;
  if (this.#quantities[position] === 0){
    this.#products.splice(position, 1);
    this.#quantities.splice(position, 1);
  }
  return this;
}

//Elimina un product del carrito si existe, sino lanzamos una excepción
removeProduct (product){
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException ();
  }
  let position = this.getProductPosition(product);
  if (position !== -1){
    this.#quantities[position] = this.#quantities[position] - 1;
    if (this.#quantities[position] <= 0){
      this.#products.splice(position, 1);
      this.#quantities.splice(position, 1);
    }
  } else{
    throw new ProductNotExistException(product);
  }
  return this;
}
```

```
//Elimina un product completo del carrito si existe, sino lanzamos una ex
cepción
removeAllProduct (product){
  if (!(product instanceof Product)) {
    throw new ProductShoppingCartException ();
  }
  let position = this.getProductPosition(product);
  if (position !== -1){
    this.#products.splice(position, 1);
    this.#quantities.splice(position, 1);
  } else{
    throw new ProductNotExistException(product);
  }
  return this;
}

//Elimina un product del carrito si existe por posición, sino lanzamos un
a excepción
removeProductPosition (position){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= this.#products.length || position < 0) throw new Positi
onOutBoundsException();
  this.#quantities[position] = this.#quantities[position] - 1;
  if (this.#quantities[position] <= 0){
    this.#products.splice(position, 1);
    this.#quantities.splice(position, 1);
  }
  return this;
}

//Elimina un product completo del carrito si existe por posición, sino la
nzamos una excepción
removeAllProductPosition (position){
  if (position === 'undefined') throw new EmptyValueException("position")
;
  if (position >= this.#products.length || position < 0) throw new Positi
onOutBoundsException();
  this.#products.splice(position, 1);
  this.#quantities.splice(position, 1);
}

//Vacía el carrito completo
clear (){
  this.#products.length = 0;
  this.#quantities.length = 0;
}
```

Por último, debemos instanciar un objeto en la función *init()*, lo congelamos y lo devolvemos para que pueda ser asignado a la variable *instanciated*.

```
let sc = new ShoppingCart
Object.freeze(sc);
return sc;
```

1.1.3 Iteradores de productos y cantidades

Los iteradores que trabajan con los campos privados hacemos una modificación para que sean iteradores e iterables, lo que nos permitirá recorrer el iterador con un bucle `for..of`. El objeto devuelto debe implementar la propiedad `System.iterator` para que devuelva la referencia `this`, así como el método `next()`.

Tenemos que hacer una leve modificación para disponer una referencia al array en la función en lugar de utilizarlo directamente, ya que el campo privado no se mantiene en el *closure* del método `next()`, pero con la referencia *array* sí que está incluido.

```
// Devuelve un iterator de los products del carrito
Object.defineProperty(this, 'products', {
  enumerable: true,
  get() {
    let nextIndex = 0;
    // referencia para habilitar el closure en el objeto
    const array = this.#products;
    return { // Objeto iterable
      [Symbol.iterator]() { // La propiedad devuelve una referecia al
propio objeto como iterador
        return this;
      },
      next() {
        return nextIndex < array.length
          ? { value: array[nextIndex++], done: false }
          : { done: true };
      },
    };
  },
});
```

```
// Devuelve un iterator de las cantidades del carrito
Object.defineProperty(this, 'quantities', {
  enumerable: true,
  get () {
    let nextIndex = 0;
    // referencia para habilitar el closure en el objeto
    const array = this.#quantities;
    return {
      [Symbol.iterator]() { // La propiedad devuelve una referecia al
propio objeto como iterador
        return this;
      },
      next() {
        return nextIndex < array.length
          ? { value: array[nextIndex++], done: false }
          : { done: true };
      },
    };
  },
});
```

El objeto *ShoppingCart* tiene naturaleza iterable. Vamos a hacer que este objeto sea un iterable implementando Symbol.iterator. En la función crearemos el iterador que nos devuelva objetos literales con dos propiedades. La propiedad *product* contendrá un objeto *Product*, y la propiedad *quantity* la cantidad de productos que se quieren comprar.

```
[Symbol.iterator](){
  let nextIndex = 0;
  // referencia para habilitar el closure en el objeto
  let aQuantities = this.#quantities;
  let aProducts = this.#products;
  return {
    [Symbol.iterator]() { // La propiedad devuelve una referecia al
propio objeto como iterador
      return this;
    },
    next: function(){
      return nextIndex < aProducts.length ?
        {value: {
          product: aProducts[nextIndex],
          quantity: aQuantities[nextIndex++]
        }, done: false} : {done: true};
    }
  }
}
```

Por último, los métodos del objeto que utilizan los iteradores los reescribimos para que utilicen un bucle `for..of` sobre el propio objeto.

```
//Muestra el carrito en formato string
toString (separator = "\n"){
  let str = "";
  for (let item of this){
    str = str + item.product.toString() + " Quantity: " + item.quantity +
separator;
  }
  return str;
}

//Devuelve el coste total del carrito
getTotal (){
  let total = 0;
  for (let item of this){
    total = total + item.product.price * item.quantity;
  }
  return total;
}

getTotalWithoutTaxes (){
  let total = 0;
  for (let item of this){
    total = total + item.product.priceWithoutTaxes * item.quantity;
  }
  return total;
}

getTaxes (){
  let total = 0;
  for (let item of this){
    total = total + item.product.tax * item.quantity;
  }
  return total;
}
```

1.1.4 Testeo

El testeo es el mismo que teníamos para la versión ES5, pero añadimos en el testeo de *ShoppingCart*, justo después de añadir productos al carrito, la visualización de su contenido utilizando los iteradores.

```
console.log('Elementos en el carrito');
for (let item of sc){
  console.log('Product: ' + item.product.toString());
  console.log('Quantity: ' + item.quantity);
}
console.log('Productos del carrito');
for (let product of sc.products){
  console.log('Product: ' + product.toString());
}
console.log('Cantidades en el carrito');
for (let quantity of sc.quantities){
  console.log('Quantity: ' + quantity);
}
```

8.6. Transformación de iteradores en generadores

Los generadores no se pueden aplicar a propiedades de acceso dentro de un objeto, pero si podemos hacer que estas propiedades devuelvan un objeto iterable implementado con un generador iterando sobre los arrays y devolviendo los valores con `yield`. Las propiedades *products* y *quantities* devuelve un objeto que implementa `Symbol.iterator`.

```
// Devuelve un iterator de los products del carrito
Object.defineProperty(this, 'products', {
  enumerable: true,
  get() {
    const nextIndex = 0;
    // referencia para habilitar el closure en el objeto
    const array = this.#products;
    return {
      * [Symbol.iterator]() {
        for (const product of array) {
          yield product;
        }
      },
    };
  },
});
```

```
// Devuelve un iterator de las cantidades del carrito
Object.defineProperty(this, 'quantities', {
  enumerable: true,
  get () {
    const nextIndex = 0;
    // referencia para habilitar el closure en el objeto
    const array = this.#quantities;
    return {
      * [Symbol.iterator]() {
        for (const quantity of array) {
          yield quantity;
        }
      },
    };
  },
});
```

La propiedad `Symbol.iterator` del objeto para que se iterable si que puede ser directamente un generador.

```
* [Symbol.iterator](){
    for (let i = 0; i < this.#products.length; i++){
      yield {
        product: this.#products[i],
        quantity: this.#quantities[i]
      }
    }
}
```

9. Módulos

ES6 tiene un mecanismo que nos permite crear módulos de forma nativa. Los módulos son entidades que nos permite **encapsular** nuestro código, es decir, nos permite empaquetar nuestro código eligiendo como controlar su acceso. Podemos elegir el interfaz de comunicación con el módulo, que fragmentos estarán protegidos desde el exterior. También nos permite definir cuáles son sus dependencias de otros módulos, por ejemplo, evitando tener que declarar variables globales que puedan ser accedidas desde cualquier lugar de nuestra aplicación.

Debemos tener en cuenta algunas consideraciones. La primera es que los módulos son *singletons*, por lo tanto, en todos los puntos de la aplicación que se importe el módulo recibirán la **misma instancia**, teniendo que compartir su uso. Esta característica hace que, si tenemos guardado el estado de un módulo mediante las propiedades de un objeto, éste puede ser cambiado desde cualquier punto de la aplicación.

En JavaScript el comportamiento de los objetos es dinámico, podemos cambiar la estructura de un objeto en tiempo de ejecución. Los objetos expuestos en un módulo tienen un **comportamiento estático**, no podemos cambiar su estructura.

Tendremos un módulo por fichero, ya que su estructura está basada en ficheros. Cada módulo debe ser solicitado mediante una conexión HTTP, lo que ralentiza la carga de la página. Aunque hay mecanismos de empaquetar varios módulos en un único recurso, función que deberíamos realizar en la fase de despliegue de la aplicación.

Las dos instrucciones que nos permiten trabajar con módulos son:

- `export`: Nos permite exportar un módulo para que sea utilizado por terceros.
- `import`: Nos permite importar un módulo en nuestra aplicación o en el módulo en el que estemos trabajando.

9.1. Importar módulos en el navegador

En la siguiente tabla mostramos la diferencia entre utilizar *scripts tradicionales* y *módulos*.

	Scripts	Módulos
Elemento HTML	`<script>`	`<script type="module">`
Modo por defecto	No estricto	Estricto
Variables	globales	Locales al módulo
Ejecución	Síncrona	Asíncrona
Declaración de `import`	No permitida	Permitida. Un módulo puede importar otros módulos.
Extensión	.js	.js

Vamos a crear un fichero denominado *mitienda.js* que sirva como módulo para nuestra aplicación.

En el fichero *index.html* cambiamos las importaciones de los *scripts* para la aplicación ya que estarán integrados en el módulo.

```
<script src="./js/mitienda/mitienda.js" type="module"></script>
```

9.2. Exportación de módulos por nombre e importación

Exportar por nombre es indicar qué funciones, clases, constantes o variables globales queremos exportar del módulo, formando el interfaz público del módulo. Lo que no exportemos permanecerá privado en el módulo. Utilizando la instrucción `export` indicamos qué queremos exportar.

Empezamos creando el **módulo para las excepciones**. En el fichero *exceptions.js* podemos redefinir la firma de cada clase de la siguiente forma.

```
export class BaseException extends Error {
}

export class InvalidAccessConstructorException extends BaseException {
}

export class EmptyValueException extends BaseException {
}

export class InvalidValueException extends BaseException {
}

export class AbstractClassException extends BaseException {
}
```

Podemos abreviar la exportación en una única sentencia indicando qué queremos exportar.

```
export {BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException
};
```

Pasamos a definir el **módulo para los productos** denominado *products* con las entidades. Este módulo lo primero que debe hacer es resolver las **dependencias** que tenga, es decir, debe importar los módulos que necesite para funcionar, básicamente el módulo de excepciones.

La sentencia `import` debe indicar qué importamos desde el módulo de las excepciones, indicando la ubicación del módulo utilizando la cláusula `from`. **La ruta al módulo es relativa al fichero donde lo estamos importando**.

Las importaciones son declaraciones y por tanto el intérprete las eleva automáticamente al inicio del fichero en el proceso de *hoisting*.

```
import {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
} from '../exceptions.js';
```

Realizamos la exportación por nombre de las clases del módulo.

```
export {Product, Laptop, Camera, Smartphone, Tablet};
```

9.3. Exportación de módulos por defecto

La exportación de módulos por nombre implica que tenemos que conocer los nombres de las funciones u objetos. En ocasiones, sobre todo si estamos haciendo un módulo para ser utilizados por terceros, no queremos hacer públicos nuestros nombres, esto lo hacemos con la exportación por defecto.

Vamos a crear el módulo para *ShoppingCart*, este módulo tiene dependencias con las excepciones y con las entidades con las clases. Lo primero que hacemos es realizar la importación de las dependencias.

```
import {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
} from '../exceptions.js';
import {
  Product, Laptop, Camera, Smartphone, Tablet,
} from '../entities/products.js';
```

Exportamos las excepciones por nombre.

```
export {ShoppingCartException, ProductShoppingCartException,
ProductNotExistException, PositionOutBoundsException};
```

Exportamos el objeto *ShoppingCart* obtenido a partir del patrón Singleton en una exportación por defecto.

```
export default ShoppingCart;
```

Con esto ya tendríamos la aplicación lista para usar. Vamos a testear *ShoppingCart* transformando el script en un módulo. Empezamos importar el objeto *ShoppingCart* por defecto en el módulo de *testeo*. En este caso estamos dando el mismo nombre, aunque podríamos haber redefinido el nombre como hubiésemos querido para utilizar en nuestro código.

```
import ShoppingCart from './shoppingcart/shoppingcart.js';
```

Para terminar el testeo debemos importar el resto de las dependencias.

9.4. Exportaciones agregadas

Nuestro módulo *shoppingcart* tiene una serie de dependencias que debe exportar para que un cliente, nuestro módulo de testeo, pueda funcionar. Podemos hacer que un módulo exporte estas dependencias desde él mismo. En nuestro módulo *shoppingcart* vamos a exportar *exceptions* y *products* para que puedan importarse directamente desde *shoppingcart*.

```
export {
  BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException,
} from '../exceptions.js';
export {
  Product, Laptop, Camera, Smartphone, Tablet,
} from '../entities/products.js';
```

Para importarlo en el testeo lo podemos hacer desde *test* y lo hacemos contra *shoppingcart*.

```
import { BaseException,
  InvalidAccessConstructorException,
  EmptyValueException,
  InvalidValueException,
  AbstractClassException } from './shoppingcart.js';
import {Product, Laptop, Camera, Smartphone, Tablet} from
'./shoppingcart.js';
import {ShoppingCartException, ProductShoppingCartException,
ProductNotExistException, PositionOutBoundsException} from
'./shoppingcart.js';
```

9.5. Uso de comodines

Exportamos las funciones de testeo, pero antes debemos eliminar el patrón IIFE para que estén definidas a alto nivel. Al tener el módulo no es necesario el patrón.

```
export {
  testShoppingCart, laptopTest, cameraTest, smartphoneTest, tabletTest,
};
```

En el módulo *mitienda* importamos las funciones, pero utilizando el comodín "*" y la cláusula `as` para darle un nombre.

```
import * as ShoppingCartTest from './test.js';
```

Ahora ya podemos invocar las funciones que estarán embebidas en el objeto *ShoppingCartTest*.

```
ShoppingCartTest.testShoppingCart();
ShoppingCartTest.laptopTest();
ShoppingCartTest.cameraTest();
ShoppingCartTest.smartphoneTest();
ShoppingCartTest.tabletTest();
```

Ahora ya podemos visualizar el testeo de la aplicación.

9.6. Renombrado de exportaciones

En caso de conflicto podrías renombrar las exportaciones por nombre que estamos realizando utilizando la cláusula `as`. Importamos una función de testeo y le damos un nuevo nombre.

```
import {testShoppingCart as newTestFunctionName} from './test.js';
```

Ahora podemos invocar la función directamente a través del nombre que le hemos dado.

```
newTestFunctionName();
```