12. CLASES

Introducción a la Programación Orientada a Objetos (POO)

La **programación orientada a objetos** (**POO**) es un **paradigma de programación** fundamental para muchos lenguajes de programación, incluidos Java y C++.

Antes de revisar cómo JavaScript incorpora el paradigma de programación orientada a objetos, vamos primero a:

- revisar una descripción general de los conceptos básicos de OOP.
- describir 3 conceptos principales de la POO:
  - **clases** e **instancias**
  - **herencia**
  - **encapsulación**.
  - **polimorfismo**.
- por ahora, estos conceptos se describirán sin hacer referencia a JavaScript en particular, por lo que todos los ejemplos se dan en pseudocódigo.
- después de eso, en JavaScript, veremos cómo los **constructores** y la **cadena de prototipos** se relacionan con estos conceptos de programación orientada a objetos y en qué se diferencian.

Uno de los objetivos principales de la programación orientada a objetos es:

- modelar el sistema que necesitamos representar en una computadora o navegador como una colección de **objetos**, donde cada objeto representa algún aspecto particular del sistema.
- los objetos contienen funciones (comportamientos o **métodos**) y datos (atributos, propiedades o campos).
- un objeto proporciona una **interfaz pública** a otro código (objeto) que quiere usarlo pero mantiene su propio **estado interno privado**; otras partes del sistema no tienen que preocuparse por lo que sucede dentro del objeto.
- siempre se busca separar, aislar, encapsular el **cómo** (lo hago) del **qué** (hago), es decir, separar **implementación** (cómo) de la **interfaz** (qué) de una clase.

Clases e instancias

Cuando modelamos un problema en términos de objetos en programación orientada a objetos, creamos definiciones abstractas que representan los **tipos de objetos** que queremos tener en nuestro sistema.

- por ejemplo, si estuviéramos modelando una escuela, podríamos querer tener objetos que representen a los profesores.
- todos los profesores tienen
  - unas **propiedades** en común:
    - nombre
    - materia/s que imparten
    - dirección vivienda
    - email
    - ....
  - además, cada profesor puede hacer ciertas cosas (**métodos** o funciones):
    - calificar un trabajo
    - presentarse a sus alumnos al comienzo del año
    - realizar una explicación
    - mandar un email
    - ..

Entonces `Profesor` podría ser una **clase** en nuestro sistema.

- una clase será como una plantilla a partir de la cual crear objetos, que son **instancias** de la clase.
- la **definición de la clase** enumera los **datos** y **métodos** que tiene cada profesor.

En pseudocódigo, una clase de `Profesor` podría escribirse así:

```
class Profesor
    properties
        nombre
        materiaImpartida
        dirección
        email
    methods
        calificarTrabajo()
        presentarse()
        explicar()
        mandarEmail()
```

Esto define una clase de `Profesor` con:

- cuatro propiedades de datos

- 4 métodos

Por sí sola, una clase no hace nada: es una especie de plantilla para crear objetos concretos de ese tipo.

- cada profesor concreto que creamos se denomina **instancia** de la clase `Profesor`.
- el proceso de creación de una instancia lo realiza una función especial llamada **constructor**.
- **estado interno** que queramos inicializar en la nueva instancia, podremos pasar valores (parámetros) al constructor.

Por lo general, el constructor se escribe como parte de la **definición de la clase** y, por lo general, tiene el mismo nombre que la clase misma:

```
class Profesor
    properties
        nombre
        materiaImpartida
        dirección
        email
    constructor
        Profesor(nombre, materiaImpartida)
    methods
        calificarTrabajo()
        presentarse()
        explicar()
        mandarEmail()
```

Este constructor toma dos parámetros, por lo que podemos inicializar las propiedades `nombre` y `materiaImpartida` con valores concretos o particulares para un profesor dado cuando creamos una nueva instancia de `Profesor`.

Ahora que tenemos un **constructor**, podemos crear algunos profesores.

- los lenguajes de programación a menudo usan la palabra clave `new` para indicar que se está llamando a un constructor.
- vamso a crear dos objetos, ambos instancias de la clase `Profesor`

```
juan  = new Profesor("Juan", "Psicología");
lucia = new Profesor("Lucía", "Poesía");

juan.materiaImpartida;      // 'Psicología'
juan.presentarse();         // 'Mi nombre es Profesor Juan y seré su profesor de Psicología.'

lucia.materiaImpartida ;    // 'Poesía'
lucia.presentarse();        // 'Mi nombre es Profesora Lucía y seré su profesora de Poesía.'
```

Herencia

Supongamos que en nuestra escuela también queremos representar a los estudiantes.

- a diferencia de los profesores, los estudiantes no pueden calificar trabajos, no enseñan una materia en particular y pertenecen a un año en particular.
- sin embargo, los estudiantes tienen un nombre y es posible que también deseen presentarse, por lo que podríamos escribir la definición de una clase de estudiantes de esta manera:

```
class Estudiante
    properties
        nombre
        curso
    constructor
        Estudiante(nombre, curso)
    methods
        presentarse()
         presentarTrabajo()
         atender()
```

Por otro lado:

- sería útil si pudiéramos representar el hecho de que los estudiantes y los profesores comparten algunas propiedades, o más exactamente, el hecho de que, en algún nivel, son el mismo tipo de "clase".
- la **herencia** nos permite hacer esto.
- comenzamos observando que los estudiantes y los profesores son personas, y las personas tienen nombres y quieren presentarse.
- podemos modelar esto definiendo una nueva clase `Persona`, donde definimos todas las propiedades comunes de las personas.
- entonces, tanto `Profesor` como `Estudiante` pueden derivar de `Persona`, agregando sus propiedades adicionales:

```
class Persona
```

```
    properties
        nombre
    constructor
        Persona(nombre)
    methods
        presentarse()

class Profesor : extends Persona
    properties
        materiaImpartida
        dirección
        email
    constructor
        Profesor(nombre, materiaImpartida)
    methods
        presentarse()
        calificarTrabajo()
        explicar()
        mandarEmail()


class Estudiante : extends Persona
    properties
        curso
    constructor
        Estudiante(nombre, curso)
    methods
        presentarse()
        presentarTrabajo()
        atender()
```

En este caso diríamos:

- que `Persona` es la **superclase** o **clase principal** tanto de `Profesor` como de `Estudiante`.
- `Profesor` y `Estudiante` son **subclases** o **clases secundarias** de `Persona`.

Como habrás observado `presentarse()` está definido en las tres clases:

- la razón de esto es que, si bien todas las personas quieren presentarse (la **interfaz** es la misma, la forma en que lo hacen es diferente (la **implementación**):

```
juan  = new Profesor("Juan", "Psicología");
juan.presentarse();          // 'Mi nombre es Profesor Juan y seré su profesor de Psicología.'

xian = new Estudiante("Xian", 1);
xian.presentarse();         // 'Mi nombre es Xian y estoy en mi primer curso.'
```

Podemos tener también una implementación predeterminada de `presentarse()` para personas que no son estudiantes o profesores:

```
ana = new Persona("Ana");
ana.presentarse();          // 'Mi nombre es Ana.'
```

Cuando un método tiene el mismo nombre (**interfaz**) pero una **implementación** diferente en diferentes clases:

- se denomina **polimorfismo**.
- cuando un método en una **subclase** reemplaza la implementación de la **superclase**, decimos que la subclase **sobreescribe** (***overrides***) la versión en la superclase.
- el entorno de ejecución (de JavaScript o cualquier otro lenguaje con esta prestación), para que esto se pueda llevar a la práctica:
  - cuando se llama/invoca a un **método polifmórfico**
    - se realiza una **ligadura dinámica** (***dynamic binding***) del nombre de la función o método con el código concreto (cuerpo del método/función) a ejecutar
    - en vez de **ligadura estática** (***static binding***) en tiempo de interpretación/compilación (bueno, técnicamente... en JavaScript todo es dinámico, pero la idea vale...)
  - por ejemplo:
    - supongamos que tenemos un array de 100 objetos `Persona`, pero unos objetos son instancias de `Persona`, otros `Profesor` y otros `Estudiante`
    - este array se fue construyendo de forma dinámica (por ejemplo, a partir de un archivo JSON, un formulario o un escaner de retina) a medida que nuestro script/programa se fue ejecutando: no tenemos ni idea de que objetos de dicho array van a ser `Persona`, cuales `Profesor` y cuales `Estudiante`
    - queremos recorrer todo el array invocando el método `presentarse()` para cada objeto, y que cada objeto se presente de forma adecuada
    - no hay ningún problema: cada vez que invocamos `presentarse()` para un objeto, el própido entorno de ejecución, selecciona en ese momento concreto (utilizando **ligadura dinámica**) la implementación de `presentarse()` (y hay 3 distntas: Persona, Profesor, Alumno) correcta para ese objeto: esto es el **polimorfismo**.

- si no existiese el polimorfismo:
  - en cada parte de nuestro script/programa que tuvieramos que procesar una estructura de datos (array, cola, set, ...) que contiene objetos mezclados de varios tipos, invocando un método de cada objeto, tendríamos que tener, por ejmplo, una estructura de control tipo `switch` y el operador `instanceof`, por ejemplo, para comprobar de qué tipo concreto es el objeto y así invocar su método concreto.
  - además, cada vez que queramos modificar nuestro script/programa para añadir un nuevo tipo de Persona, por ejemplo, `Conserje` o `Tecnico`, tendríamos que buscar en todos los sitios de nuestra aplicación en la que tomamos decisiones en función del tipo concreto de `Persona` y cambiar el código para adaptarlo a los nuevas clases de `Persona`, `Conserje` o `Tecnico`: un engorro y sujeto a posibles errores
  - con el polimorfismo: simplemente creamos las 2 nuevas clases `Conserje` y `Tecnico` derivadas de `Persona` y el código polimórfico se encarga prácticamente de todo.

Encapsulación

Los objetos proporcionan una **interfaz** para otro código que quiere usarlos pero deberían mantener **oculto** su propio **estado interno**.

- el estado interno del objeto se mantiene **privado** (**oculto**), lo que significa que solo se puede acceder a él mediante los propios métodos del objeto, no desde otros objetos.
- mantener privado el estado interno de un objeto y, en general, hacer una división clara entre su **interfaz pública** y su **estado interno privado**, se denomina **encapsulación**.

Esta es una prestación muy útil porque permite al programador cambiar la **implementación interna** de un objeto A (para realizar una mejora, para cumplir con una nueva normativa, para adaptarlo a una actualización de hardware, ...) sin tener que buscar y actualizar todo el otro código que usa ese objeto A: crea una especie de cortafuegos entre este objeto y el resto del sistema.

Por ejemplo, continuando con el ejemplo anterior, supón que a los estudiantes se les permite estudiar tiro con arco si están en el segundo año o más.

- podemos implementar esto simplemente **exponiendo** la propiedad del curso del estudiante, y otro código podría examinar eso para decidir si el estudiante puede matricularse en la materia "tiro con arco":

```
if (estudiante.curso > 1) {
    // permitir que el estudiante entre a la clase
}
```

- uno de los problemass de esta solución es que, si decidimos cambiar los criterios para permitir que los estudiantes estudien tiro con arco, por ejemplo, solicitando también el permiso de los padres o tutores, necesitaremos actualizar todos los lugares de nuestro sistema que realizan esta prueba.
- sería mejor tener un método `puedeCursarArqueria()` en los objetos `Estudiante`, que **implementa la lógica** en un solo lugar:

```
class Estudiante : extends Persona
    properties
        curso
    constructor
        Estudiante(nombre, curso)
    methods
        presentarse()
        presentarTrabajo()
        atender()
        puedeCursarArqueria()  { return this.curso > 1; }
```

```
if ( estudiante.puedeCursarArqueria() ) {
    // permitir que el estudiante entre a la clase
}
```

- de esa manera, si queremos cambiar las reglas sobre el estudio del tiro con arco, solo tenemos que actualizar la clase `Estudiante`, y todo el código que la use seguirá funcionando.
- este es sólo un pequeño ejemplo, de las profundas ventajas que ofrece la encapsulación.

En muchos lenguajes OOP, podemos evitar que otro código acceda al estado interno de un objeto marcando algunas propiedades como **privadas**.

- esto generará un error si el código fuera del objeto intenta acceder a ellos:

```
class Estudiante : extends Persona
    properties
        private curso
    constructor
        Estudiante(nombre, curso)
    methods
        presentarse()
```

```
        presentarTrabajo()
        atender()
        puedeCursarArqueria()  { return this.curso > 1; }

roberta = new Estudiante("Roberta", 1);
roberta.curso;        // error: la propiedad 'curso' es privada
```

En los lenguajes que no disponen de un **mecanismo de control de acceso** como este, los programadores usan **convenciones de nomenclatura**, como comenzar el nombre con un guión bajo, para indicar que la propiedad debe considerarse privada.

Programación orientada a objetos y JavaScript

Hasta ahora, hemos revisado algunas de las características básicas de la programación orientada a objetos basada en clases tal como se implementa en lenguajes como Java y C++.

En JavaScript tenemos:

- **constructores** y **prototipos**: estas características ciertamente tienen alguna relación con algunos de los conceptos de programación orientada a objetos descritos anteriormente.
  - los **constructores** en JavaScript nos brindan algo así como una definición de clase, lo que nos permite definir la "forma" de un objeto, incluidos los métodos que contiene, en un solo lugar.
    - pero para esto también se pueden usar prototipos.
    - por ejemplo, si un método se define en la propiedad de prototipo (`prototype`) de un constructor, entonces todos los objetos creados con ese constructor obtienen ese método a través de su prototipo, y no necesitamos definirlo en el constructor.
- la **cadena de prototipos** parece una forma natural de implementar la **herencia**.
  - por ejemplo, si podemos tener un objeto `Estudiante` cuyo prototipo sea `Persona`, entonces puede **heredar** la propiedad `nombre` y **sobreescribir** (***override***) `presentarse()`.

Pero vale la pena pararse a comprender las diferencias entre como implementa JavaScript estas prestaciones y los conceptos de OOP "clásicos" descritos anteriormente. Destacaremos un par de ellos aquí.

- Primero:
  - en la programación orientada a objetos basada en clases, las clases y los objetos son dos construcciones separadas, y los objetos siempre se crean como instancias de clases (no existe la **cadena de prototipos**).
  - además, existe una distinción entre la prestación utilizada para definir una clase (la sintaxis de la clase en sí) y la prestación utilizada para instanciar un objeto (un constructor).
  - en JavaScript, podemos y, a menudo creamos objetos:
    - sin ninguna definición de clase separada, ya sea usando una **función** o un **objeto literal**.
    - esto puede hacer que trabajar con objetos sea mucho más ligero que en la programación orientada a objetos clásica.
- En segundo lugar:
  - aunque una cadena de prototipos se parece a una **jerarquía de herencia** y se comporta como tal en algunos aspectos, es diferente en otros.
    - cuando se creas una instancia de una subclase, se crea un solo objeto que combina propiedades definidas en la subclase con propiedades definidas más arriba en la jerarquía.
    - con la creación de prototipos:
      - cada nivel de la jerarquía está representado por un objeto separado y están vinculados entre sí a través de la propiedad `__proto__`.
      - el comportamiento de la cadena prototipo es menos como herencia y más como **delegación**.
        - la delegación es un **patrón de programación** en el que un objeto, cuando se le pide que realice una tarea, puede realizar la tarea por sí mismo o pedirle a otro objeto (su delegado) que realice la tarea en su nombre.
        - en muchos sentidos, la delegación es una forma más flexible de combinar objetos que la herencia (por un lado, es posible cambiar o reemplazar completamente el delegado en tiempo de ejecución).

Dicho esto:

- los **constructores** y **prototipos** se pueden usar para implementar patrones OOP basados en clases en JavaScript.
- pero usarlos directamente para implementar prestaciones como la **herencia** es complicado, por lo que JavaScript proporciona prestaciones adicionales, añadida en una capa sobre el modelo de cadena de prototipos ya existente, que se relacionan más directamente con los conceptos de programación orientada a objetos basada en clases.
- estas características adicionales son el tema del siguiente apartado.

Introducción a las Clases en JavaScript

En el último artículo, presentamos algunos conceptos básicos de la programación orientada a objetos (OOP) y discutimos un ejemplo en el que usamos los principios de la OOP para modelar profesores y estudiantes en una escuela.

También hablamos sobre cómo es posible usar prototipos y constructores para implementar un modelo como este, y que JavaScript también proporciona características que se asemejan más a los conceptos clásicos de programación orientada a objetos.
En este artículo, repasaremos estas características. Vale la pena tener en cuenta que las características descritas aquí no son una nueva forma de combinar objetos: debajo del capó, todavía usan prototipos. Son solo una forma de facilitar la configuración de una cadena de prototipos.

Clases y constructores

Puede declarar una clase usando la palabra clave class. Aquí hay una declaración de clase para nuestra **`Persona`** del artículo anterior:

```
class Persona {

  nombre;

  constructor(nombre) {
    this.nombre = nombre;
  }

  presentarse() {
    console.log(`¡Hola! Soy ${this.nombre}`);
  }
}
```

Esto declara una clase llamada `Persona`, con:

- una **propiedad** nombre.
  - la declaración **`nombre;`** es opcional: puedes omitirla y la línea **`this.nombre = nombre;`** en el constructor creará la propiedad de **`nombre`** antes de inicializarla.
    - sin embargo, **enumerar las propiedades explícitamente** en la **declaración de la clase** podría facilitar que otros programadores que leen tu código vean más fácilmente (mejor **legilibilidad** del código) qué propiedades forman parte de esta clase.
  - también puedes inicializar la propiedad **`nombre`** a un valor predeterminado cuando lo declaras, con una línea como `nombre = '';`
- un **constructor** que toma un parámetro de **`nombre`** que se usa para inicializar la propiedad de **`nombre`** del nuevo objeto
  - se define mediante la palabra clave **`constructor`**.
  - al igual que un constructor fuera de una definición de clase, hará lo siguiente:
    - crear un nuevo objeto (pore sólo si se invoca a través del operador **`new`**) en memoria
    - vincular **`this`** al nuevo objeto, para que puedas utilizar **`this`** en el código del constructor y otros métodos de instancia
    - ejecutar el código del cuerpo del constructor
    - devolver (una **referencia** a) el nuevo objeto.
- un método **`presentarse()`** que puede hacer uso de las propiedades del objeto usando **`this`**.

Dado el código de **declaración de clase** anterior, puedes crear y usar una nueva **instancia** de `Persona` asi:

```
const lara = new Persona('Lara');

lara.presentarse();   // ¡Hola! Soy Lara
```

Ten en cuenta que llamamos al constructor usando el nombre de la clase, `Persona` en este ejemplo.

Omitir constructores

Si no necesita realizar ninguna inicialización especial, puede omitir el constructor y se generará un **constructor predeterminado** automáticamente:

```
class Animal {
  dormir() {
    console.log(`zzzzzzzz`);
  }
}
```

```
const tigre = new Animal();

tigre.dormir (); // 'zzzzzzzz'
```

Herencia

Dada nuestra clase Persona anterior, definamos la subclase Profesor.

```
class Persona {

  nombre;

  constructor(nombre) {
    this.nombre = nombre;
  }

  presentarse() {
    console.log(`¡Hola! Soy ${this.nombre}`);
  }
}


class Profesor extends Persona {

  materiaImpartida;

  constructor(nombre, materiaImpartida) {
    super(nombre);
    this.materiaImpartida = materiaImpartida;
  }

  presentarse() {
    console.log(`Mi nombre es ${this.nombre}, y yo os enseñaré ${this.materiaImpartida}`);
  }

  calificarTrabajo(trabajo) {
    const nota = Math.floor(Math.random() * (10 - 1) + 1);
    console.log(`La calificación es: ${nota}`);
  }

  explicar() {
    console.log(`Había una vez la asignatura ${this.materiaImpartida }`);
  }


  mandarEmail(email) {
    console.log(`email:${email}: esto es un correo de ${this.nombre}`);
  }
}
```

Usamos la palabra clave **`extends`** para decir que esta clase **hereda de** otra clase:

- la clase **`Profesor`** agrega una nueva propiedad enseña, por lo que declaramos eso.
- dado que queremos configurar las enseñanzas cuando se crea un nuevo **`Profesor`**, definimos un constructor, que toma el nombre y las enseñanzas como argumentos.
  - lo primero que hace este constructor es llamar al constructor de **superclase** usando **`super()`**, pasando el parámetro nombre.
  - el constructor de la superclase se encarga de establecer el nombre.
  - después de eso, el constructor **`Profesor`** establece la propiedad enseña.

Nota: si una subclase tiene que hacer alguna **inicialización** propia, primero debe llamar al **constructor de la superclase** usando **`super()`**, pasando cualquier **parámetro** que el constructor de la superclase esté esperando.

También **sobreescribimos** (***override***) el método **`presentarse()`** de la **superclase** y agregamos nuevos métodos:

- **`calificarTrabajo()`**
- **`explicar()`**
- **`mandarEmail()`**

Tras declarar la clase ahora podemos crear y usar profesores:

```
const marga = new Profesor('Marga', 'Historia');

marga.presentarse();                        // Mi nombre es Marga y yo os enseñaré Historia
marga.calificarTrabajo('mi trabajo');       // La calificación es: 7
```

Encapsulación

Finalmente, veamos cómo implementar la **encapsulación** en JavaScript.

- anteriormente discutimos que podría ser útil hacer la propiedad curso de Estudiante privada, para que, si nos interesa por alguna razón, poder cambiar las reglas sobre las clases de tiro con arco sin causar que otro código, quizás en otro fichero, que use la clase Estudiante deje de funcionar correctamente o simplemente no compile.
- las propiedades de datos **privadas** deben declararse en la declaración de clase y sus nombres comienzan con #.
- una declaración de la clase `Estudiante` que logra precisamente eso es:

```
class Estudiante extends Persona {

  #curso;                           // propiedad privada

  constructor(nombre, curso) {
    super(nombre);
    this.#curso = curso;
  }


  presentarse() {
    console.log(`¡Hola! Soy ${this.name}, y estoy en el curso ${this.#curso}.`);
  }

  atender() {
    console.log(`Estoy atendiendo....`);
  }

  puedeCursarArqueria() {
    return this.#curso > 1;
  }

}
```

En esta declaración de clase, `#curso` es una **propiedad privada**

- puedes construir un objeto `Estudiante`, y puedes usar `#curso` internamente dentro de cualquier parte del código (no estático) de la clase, pero si el código fuera del objeto intenta acceder a `#curso`, el entorno de ejecución (el navegador, node.js, ...) arroja (throws) un error:

```
const lucas = new Estudiante('Lucas', 2);

lucas.presentarse();              // ¡Hola! Soy Lucas, y estoy en el curso 2.
lucas.puedoCursarArqueria();      // true

lucas.#curso;                     // SyntaxError (error sintáctico)
```

Métodos privados

Puedes utilizar **métodos privados** de la misma forma que propiedades de datos privadas.

- al igual que las propiedades de datos privadas, sus nombres comienzan con `#` y solo pueden ser llamados por los propios métodos del objeto:

```
class Ejemplo {
  unMetodoPublico() {
    this.#unMetodoPrivado();
  }

  #unMetodoPrivado() {
    console.log('¿Me llamaste?');
```

```
  }
}

const miEjemplo = new Ejemplo();

miEjemplo.unMetodoPublico();         // ¿Me llamaste?

miEjemplo.#unMetodoPrivado();        // SyntaxError
```

Programación Orientada a Objetos (POO)

JavaScript:

- es un **lenguaje basado en prototipos**: los **comportamientos de un objeto** están especificados por
  - sus propias propiedades y métodos
  - las propiedades y métodos de su prototipo (o mejor dicho, su cadena de prototipos)
- sin embargo, con la adición al lenguaje de **clases**, la creación de **jerarquías de objetos** y la **herencia** de propiedades y sus **valores** están mucho más en línea con otros lenguajes orientados a objetos como Java.
- en muchos otros lenguajes, las **clases** o **constructores** se distinguen claramente de los **objetos** o **instancias**.
  - **en JavaScript, las clases son principalmente una abstracción sobre el mecanismo de herencia basado en prototipos existente: todos los patrones se pueden convertir en herencia basada en prototipos.**
  - las clases en sí mismas también son valores JavaScript normales y tienen sus propias cadenas de prototipos.
  - de hecho, la mayoría de las funciones simples de JavaScript se pueden usar como constructores: usas el **operador `new`** con una función constructora para crear un nuevo objeto.

Resumen de clases

Si tienes alguna experiencia práctica con JavaScript probablemente ya has usado clases, incluso si no has creado una.

- por ejemplo, este código puede parecerte conocído:

```
const granDia = new Date(2019, 6, 19);
console.log(granDia.toLocaleDateString());
if (granDia.getTime() < Date.now()) {
  console.log("Érase una vez ...");
}
```

- en la primera línea, creamos una instancia de la clase `Date` y la llamamos **`granDia`**.
- en la segunda línea, llamamos a un método `toLocaleDateString`() en la instancia de **`granDia`**, que devuelve una cadena.
- luego, comparamos dos números: uno devuelto por el método `getTime`(), el otro llamado directamente desde la propia clase `Date`, como `Date.now`().

`Date` es una **clase integrada** (***built-in***) de JavaScript; a partir de este ejemplo, podemos obtener algunas ideas básicas de lo que hacen las clases:

- las clases permiten crear objetos a través del operador **`new`**.
- cada objeto tiene algunas propiedades (datos o métodos) agregadas por la clase.
- la clase almacena algunas propiedades (datos o métodos) en sí misma, que generalmente se usan para interactuar con **instancias**.

Estos corresponden a las tres características clave de las clases:

- **constructor**
- **métodos de instancia** y **campos de instancia**
- **métodos estáticos y campos estáticos**

Declarar una clase

Las clases generalmente se crean con **declaraciones de clase***:*

```
class MiClase {
// cuerpo de la clase...
}
```

Dentro de un **cuerpo de clase**, hay una variedad de características disponibles.

```
class MiClase {

// Constructor
constructor() {
   // Cuerpo constructor
}

// Campo de instancia
miCampo = "foo";

// Método de instancia
miMetodo() {
  // cuerpo de mi método
}
```

```
// campo estático
static miCampoEstatico = "barra";

 // método estático
static miMetodoEstatico () {
   // cuerpo de miMetodoEstatico
}



// Bloque estático
static {
    // código de inicialización estática
}

// Los campos, métodos, campos estáticos y métodos estáticos
// tienen la posibilidad de ser privados
#miCampoPrivado = "barra";
static #miCampoEstaticoPrivado = 0; // contador de instancias de esta clase creadas

}
```

Si vienes de un mundo anterior a ES6, es posible que esté más familiarizado con el uso de funciones como constructores.

- el patrón anterior se traduciría aproximadamente a lo siguiente con constructores de funciones:

```
function MiClase() {
  // Campo de instancia
  this.miCampo = "foo";
  // Cuerpo del constructor
}


// Metodos de instancia:
MiClase.prototype.miMetodo = function () {
  // cuerpo de miMetodo
};

// Campo estático:
MiClase.miCampoEstatico = "bar";

// Método estático:
MiClase.miMetodoEstatico = function () {
  //  cuerpo de miMetodoEstatico
};


(function () {
  // Codigo de inicialización estatico
})();
```

Nota: los campos privados y métodos privados son características nuevas en las clases sin un equivalente trivial en los constructores de funciones.

Construyendo una clase

Una vez declarada una clase, puedes crear instancias de ella utilizando el operador `new`.

```
const miInstancia = new MiClase();
console.log(miInstancia.miCampo); // 'foo'
miInstancia.miMetodo();
```

Tenemos que:

- los constructores de funciones típicos pueden construirse con `new` y tambien llamarse sin `new` (en este último caso su comportamiento es diferente a cuando se invocan con `new`)
- sin embargo, intentar "llamar" a una clase sin `new` resultará en un error.

```
const miInstancia = MiClase();
// TypeError: el constructor de clase MiClase no se puede invocar sin 'new'
```

Elevar (*hoisting*) la declaración de clase

A diferencia de las declaraciones de funciones, las declaraciones de clases no se elevan (o, en algunas interpretaciones, se elevan pero con la **restricción de zona muerta temporal**), lo que significa que no puedes usar una clase antes de que se declare.

```
new MiClase(); // ReferenceError: no se puede acceder a 'MiClase' antes de la inicialización

class MiClase {}
```

Este comportamiento es similar a las variables declaradas con `let` y `const`.

Expresiones de clase

Al igual que las funciones, las declaraciones de clase también tienen sus contrapartes de **expresión clase**.

```
const MiClase = class {
   // Cuerpo de la clase...
};
```

- las expresiones clase también pueden tener nombres.
- el nombre de la expresión solo es visible para el **cuerpo de la clase**.

```
const MiClase = class NombreLargoMiClase {
   // Cuerpo de la clase. Aquí MiClase y NombreLargoMiClase apuntan a la misma clase.
... new  NombreLargoMiClase(); // correcto
};
new  NombreLargoMiClase(); // ReferenceError: NombreLargoMiClase no está definido
```

Constructor

Quizás el trabajo más importante de una clase es actuar como una " **fábrica" de objetos**.

- por ejemplo, cuando usamos el **constructor** `Date`, esperamos que devuelva un nuevo objeto A que represente los datos de fecha que le pasamos: que luego podemos manipular con otros métodos que expone la instancia objeto A.
- en las clases, la **creación de la instancia** la realiza el **constructor**.
- como ejemplo, crearíamos una clase llamada `Color`, que representa un color específico.
  - los usuarios crean colores pasando un triplete RGB.

```
class Color {
  constructor(r, g, b) {
    // Assign the RGB values as a property of `this`.
    this.valores = [r, g, b];
  }
}

const rojo = new Color(255, 0, 0);
console.log(rojo);
```

Deberías ver una salida como esta:

```
Object { values: (3) […] }
  values: Array(3) [ 255, 0, 0 ]
```

- has creado con éxito una instancia de `Color`,
- la instancia tiene una propiedad **`valores`**, que es un array de los valores RGB que pasaste al constructor.
- lo anterior es más o menos equivalente a lo siguiente:

```
function crearColor(r, g, b) {
   return {
       valores: [r, g, b],
   };
}
```

La sintaxis del constructor es exactamente la misma que una función normal, lo que significa que puedes usar otras sintaxis, como **parámetros *rest*** :

```
class Color {
  constructor(...valores) {
    this.valores = valores;
  }
}

const rojo = new Color(255, 0, 0);
// Crea una instancia con la misma forma que la anterior.
```

- cada vez que llamas a **new**, se crea una instancia diferente.

```
const rojo = new Color(255, 0, 0);
const otroRojo = new Color(255, 0, 0);
console.log(rojo === otroRojo);  // false
```

Dentro de un constructor de clase, el valor de **this** apunta a la instancia recién creada.

- puede asignarle propiedades o leer propiedades existentes (especialmente métodos, que veremos a continuación).
- este valor se devolverá automáticamente como resultado de **new**.
- se le aconseja que no devuelva ningún valor desde el constructor: porque si devuelve un valor no primitivo, se convertirá en el valor de la nueva expresión, y el valor de **this** se descarta.

```
class MiClase {
  constructor() {
    this.miCampo = "foo";
    return {};
  }
}

console.log(new MiClase().miCampo); // undefined
```

Métodos de instancia

Si una clase solo tiene un constructor, no es muy diferente de una **función fábrica** (***factory function***) `crearX` que solo crea objetos simples.

- sin embargo, el poder de las clases es que pueden usarse como "plantillas" que automáticamente asignan métodos a las **instancias**.
- por ejemplo, para instancias de `Date`
  - puedes usar una variedad de métodos `getX` para obtener información diferente de un solo valor de fecha, como el año, el mes, el día de la semana, etc.
  - también puedes establecer esos valores a través de los correspondientes `setX` (ejemplo: `setFullYear`.)

Para nuestra propia clase `Color`, podemos agregar un método llamado `getRojo` que devuelve el valor rojo del color.

```
class Color {
  constructor(r, g, b) {
    this.valores = [r, g, b];
  }
  getRojo() {
    return this.valores[0];
  }
}

const rojo = new Color(255, 0, 0);
console.log(rojo.getRojo());         // 255
```

- si no existiesen los métodos miembros, quizás te verías tentado a definir una función dentro del constructor:

```
class Color {
  constructor(r, g, b) {
    this.valores = [r, g, b];
    this.getRojo = function() {
       return this.valores[0];
  }
  }
```

```
}
```

- esto también funcionaría: sin embargo, un problema es que esto crea una nueva función (duplica el código de función) cada vez que se crea una instancia de `Color`, ¡incluso cuando todos hacen lo mismo!

```
console.log( new Color().getRojo === new Color().getRojo ); // falso
```

- en cambio, si usas un **método miembro**, se compartirá entre todas las instancias.
  - una función se puede compartir entre todas las instancias, pero aún así su comportamiento difiere cuando la llaman diferentes instancias, porque el valor de `this` es diferente.
  - si tienes curiosidad sobre dónde se almacena este método, está definido en el **prototipo** de todas las instancias, o `Color.prototype`, que se explica con más detalle en Herencia y la cadena de prototipos.

Análogamente, podemos crear un nuevo método llamado `setRojo`, que establece el valor rojo del color.

```
class Color {

  constructor(r, g, b) {
    this.valores = [r, g, b];
  }

  getRojo() {
    return this.valores[0];
  }

  setRojo(valor) {
    this.valores[0] = valor;
  }
}

const rojo = new Color(255, 0, 0);
rojo.setRojo(0);
console.log(rojo.getRojo());  //  Devuelve 0
```

Campos privados

MOTIVACIÓN: EL PROBLEMA

Quizás te preguntes: ¿por qué queremos tomarnos la molestia de usar los métodos `getRojo` y `setRojo`, cuando podemos acceder directamente a la matriz de `valores` en la instancia?

```
class Color {
  constructor(r, g, b) {
    this.valores = [r, g, b];
  }
}

const rojo = new Color(255, 0, 0);
rojo.valores [0] = 0;
console.log(rojo.valores[0]);          // 0
```

- hay una filosofía en la programación orientada a objetos llamada " **encapsulación** ".
  - esto significa que para interactuar con un objeto, no debes acceder a la **implementación subyacente** del "interior" de un objeto (los detalles de **cómo** se representan y almacenan los datos, y **cómo** se implementan los métodos de un objeto), sino usar **métodos públicos** (que constituyen su **interfaz**)
  - podemos decir que las propiedades públicas del objeto constituyen la **API del objeto**
  - por ejemplo, imaginemos que, por la razón que sea, de repente tuvieramos que representar los colores como **HSL**:

```
class Color {
  constructor(r, g, b) {
    this.valores = rgbToHSL([r, g, b]);    // Los valores se almacenan otro formato ahora: HSL
  }

  getRojo() {
    return this.valores[0];
  }

  setRojo(valor) {
    this.valores[0] = valor;
  }
```

```
}

const rojo = new Color(255, 0, 0);
console.log(rojo.valores[0]);       // Devuelve 0: ya no es 255 (el valor H para rojo puro es 0)
```

- la suposición del código cliente de nuestro objeto `Color` de que los **`valores`** significan que el valor RGB colapsa repentinamente y puede causar que la lógica de su código no funcione correctamente:
  - por lo tanto, si eres un buen implementador de una clase, querrás **ocultar la estructura de datos interna** de tu clase a otro código cliente, tanto para:
    - **mantener la API limpia**
    - evitar que el código del usuario "se rompa" cuando tu realizes alguna "refactorización" (cambios) del código de tu clase inofensiva
  - en las clases, esto se logra usando **campos privados**.

SOLUCIÓN: CAMPOS PRIVADOS

Un **campo privado** es un **identificador** con el prefijo `#` (el símbolo almohadilla o hash).

- El hash es una parte integral del nombre del campo.
- Para hacer referencia a un campo privado en cualquier parte de la clase, debe declararlo en el cuerpo de la clase.
- Aparte de esto, un campo privado es más o menos equivalente a una propiedad normal.

```
class Color {

  #valores;  // cada instancia de Color (creada por ejmplo con new Color) tendrá
             //  una copia independiente de este dato

  constructor(r, g, b) {
    this.#valores = [r, g, b];
  }

  getRojo() {
    return this.#valores[0];
  }

  setRojo(valor) {
    this.#valores[0] = valor;
  }
}

const rojo = new Color(255, 0, 0);
console.log(rojo.getRojo());        //  Devuelve 255
```

- acceder a **campos privados** fuera de la clase es un **error de sintaxis temprano** (**error en tiempo de interpretación/compilación**)
- el lenguaje puede protegerse contra esto porque **`#campoPrivado`** es una sintaxis especial, por lo que puede realizar un **análisis estático** y encontrar todo el uso de campos privados incluso antes de evaluar/ejecutar el código.

```
console.log(rojo.#valores);
// SyntaxError: el campo privado '#valores' debe declararse en una clase adjunta
```

Los campos privados en JavaScript

- son "privados estrictos": si la clase no implementa métodos que expongan estos campos privados, no hay absolutamente ningún mecanismo para recuperarlos desde fuera de la clase.
- esto significa que puedes realizar de forma segura cualquier **refactorización** en los campos privados de tu clase, siempre que el comportamiento de los métodos expuestos (públicos) permanezca igual.

Ahora que hemos hecho que el campo de valores sea privado, podemos agregar algo más de lógica en los métodos `getRojo` y `setRojo`, en lugar de convertirlos en simples métodos de transferencia.

- por ejemplo, podemos agregar una validación en `setRojo` para ver si es un valor R válido:

```
class Color {
  #valores;

  constructor(r, g, b) {
    this.#valores = [r, g, b];
  }

  getRojo() {
    return this.#valores[0];
```

```
  }

  setRojo(valor) {
    if (value < 0 || value > 255) {
      throw new RangeError("Componente R inválida");
    }
    this.#valores[0] = valor;
  }
}

const rojo = new Color(255, 0, 0);
rojo.setRojo(1000);       // RangeError: Componente R inválida
```

Si dejamos expuesta la propiedad de los valores:

- nuestros usuarios pueden eludir fácilmente esa verificación al asignar `valores[0]` directamente y crear colores no válidos.
- pero con una **API bien encapsulada**, podemos hacer que nuestro código sea más **robusto** y evitar **errores lógicos** posteriores.

ACCESO A CAMPOS PRIVADOS DE OTRAS INSTANCIAS DE MISMA CLASE

Un **método de clase** (**método miembro**) puede leer los campos privados de otras instancias de la misma clase:

```
class Color {
  #valores;

  constructor(r, g, b) {
    this.#valores = [r, g, b];
  }

  diferenciaRojo(otroObjetoColor) {
    // distintos objetos instancia de la misma clase pueden acceder a sus campos privados
    return this.#valores[0] - otroObjetoColor.#valores[0];
  }
}

const rojo   = new Color(255, 0, 0);
const carmin = new Color(220, 20, 60);
rojo.diferenciaRojo(carmin); // 35
```

- sin embargo, si `otroObjetoColor` no es una instancia de `Color`, los `#valores` no existirán.
- incluso si otra clase `C` tiene un campo privado `#valores`, desde el código de la clase `Color` no podremos acceder a él.

Para comprobar de antemano si el campo existe, podemos utilizar una comprobación con el operador `in` .

```
class Color {
  #valores;

  constructor(r, g, b) {
    this.#valores = [r, g, b];
  }

  diferenciaRojo(otroObjetoColor) {
    if (!(#valores in otroObjetoColor)) {
      throw new TypeError("Color instance expected");
    }
    return this.#valores[0] - otroObjetoColor.#valores[0];
  }
}
```

Nota: ten en cuenta que el # es una sintaxis de identificador especial y no puede usar el nombre del campo como si fuera una cadena.

- `#valores in otroObjetoColor`: buca un nombre de propiedad llamado literalmente "`#valores`", en lugar de un campo privado que se llame `valores`

MÉTODOS PRIVADOS

Los métodos también pueden ser privados.

```
class Color {
```

```
  #valores;

  constructor(r, g, b) {
    this.#miMetodoPrivado();
    this.#valores = [r, g, b];
  }

  #miMetodoPrivado() {
    // …
  }
}
```

Campos accesores

`color.getRojo()` y `color.setRojo()` nos permiten leer y escribir en el valor rojo de un color.

- si vienes de lenguajes como Java, estarás muy familiarizado con este patrón.
- sin embargo, el uso de métodos para simplemente acceder a una propiedad sigue siendo algo poco ergonómico en JavaScript.
- Un **campo accesor** (***accessor field***) nos permite manipular algo como si fuera una "propiedad real".

```
class Color {

  constructor(r, g, b) {
    this.valores = [r, g, b];
  }

  get rojo() {
    return this.valores[0];
  }

  set rojo(valor) {
    this.valores[0] = valor;
  }
}

const colorRojo = new Color(255, 0, 0);
colorRojo.rojo = 0;
console.log(colorRojo.rojo);     // 0
```

- parece como si el objeto tuviera una propiedad llamada `rojo`, pero en realidad, ¡tal propiedad no existe en la instancia!
- solo hay dos métodos, pero tienen el prefijo `get` y `set`, lo que permite utitlizar los métodos como si fueran propiedades.
- si un campo solo tiene un **getter** pero no un **setter**, será efectivamente de solo lectura.

```
class Color {

  constructor(r, g, b) {
    this.valores = [r, g, b];
  }

  get rojo() {
    return this.valores[0];
  }
}

const colorRojo = new Color(255, 0, 0);
colorRojo.rojo = 0;
console.log(colorRojo.rojo);     // 255
```

- en **modo estricto**, la línea `colorRojo.rojo = 0;` lanzará un **error de tipo**: `"Cannot set property red of #<Color> which has only a getter"`.
- en el **modo no estricto**, la asignación **se ignora silenciosamente**.

Campos públicos

De la misma forma que existen los **campos privados** también hay **campos públicos**, que permiten que cada instancia tenga una copia independiente de una propiedad.

- los campos generalmente están diseñados para ser independientes de los parámetros del constructor.

```
class MiClase {
```

```
  numeroDeLaSuerte = Math.random();
}
console.log(new MiClase().numeroDeLaSuerte); // 0.5
console.log(new MiClase().numeroDeLaSuerte); // 0.3
```

- los campos públicos son casi equivalentes a asignar una propiedad a `this`.
- por ejemplo, el ejemplo anterior también se puede convertir a:

```
class MiClase {
  constructor() {
    this.numeroDeLaSuerte = Math.random();
  }
}
```

Propiedades estáticas

Con el ejemplo de `Date`, también nos hemos encontrado con el método `Date.now()`, que devuelve la fecha actual.

- este método no pertenece a ninguna instancia de fecha: pertenece a la clase misma.
- sin embargo, se coloca en la clase `Date` en lugar de exponerse como una **función global** `DateNow()`, porque es útil principalmente cuando se trata de instancias de fecha.

Nota: poner como prefijo de los métodos de utilidad (*utility methods*) un identificador que indique el contexto de su función se denomina "**crear un espacio de nombres** " (***namespacing***) y se considera una buena práctica.

- por ejemplo, además del método global antiguo `parseInt()` y sin prefijo, existe un método agregado más tarde `Number.parseInt()` para indicar que es para tratar con números.

Una propiedad estática:

- es una propiedad que pertenece a la propia clase y no se realiza una copia independiente para cada instancia de la clase, como es el caso con una **propiedad miembro** (**propiedad de clase**)
- una propiedad estática está disponible aunque no se haya creado ninguna instancia de la clase.
- disponemos de las prestaciones del lenguaje:
  - **método estático**
  - **campo estático**
  - **getter estático**
  - **setter estático**
  - **bloque de inicialización estática**
- además, cada uno de los anteriores puede ser **público** o **privado**.

Por ejemplo, para nuestra clase `Color`, podemos crear un **método estático** que verifique si un triplete dado es un valor RGB válido:

```
class Color {
  static esValido(r, g, b) {
    return (r >= 0 && r <= 255) && (g >= 0 && g <= 255) && (b >= 0 && b <= 255);
  }
}

Color.esValido(255, 0, 0);      // true
Color.esValido(1000, 0, 0);     // false
```

La sintaxis de las **propiedades estáticas** es similar a las propiedades de instancia, excepto que:

- todos tienen el prefijo `static`, y
- no son **accesibles** desde una referencia a una instancia de esa clase (o cualquier otra clase):
  - es necesario utilizar la sintaxis *`NombreDeClase.propiedadEstatica`* para acceder a ellas

```
const colorRojo = new Color(255, 0, 0);
console.log(colorRojo.isValid(255, 34, 43) );  // indefinido
```

También hay una construcción especial llamada **bloque de inicialización estático**, que es un bloque de código que se ejecuta cuando la clase **se carga** por primera vez.

```
class MiClase {
  static {
    MiClase.miPropiedadEstatica = "foo";
  }
}
```

```
console.log(MiClase.miPropiedadEstatica);      // 'foo'
```

- los bloques de inicialización estáticos son casi equivalentes a ejecutar inmediatamente algún código después de que se haya declarado una clase.
- la única diferencia es que tienen acceso a **propiedades estáticas privadas**.

Extiende y herencia

Una característica clave que ofrecen las clases (además de la **encapsulación** ergonómica con campos privados) es la ***herencia***

- la herencia permite que una clase B (objeto) pueda "tomar prestada" una gran parte de los comportamientos (métodos) de otra clase A (objeto), pudiendo para algunos (o todos) de los comportamientos heredados:
  - **sobreescribirlos** (***override***): implica que la clase B crea desde cero código nuevo para un método ya existente en la clase de la que hereda A.
  - **ampliar** (***enhances***): implica que la clase B crea código nuevo para un método "`metodoHeredado`" ya existente en la clase de la que hereda A, utilizando para ello posiblemente el método con el mismo nombre "`metodoHeredado`" de la clase A, usando el operador `super: super.metodoHeredado()`
- por ejemplo, supongamos que nuestra clase `Color` ahora necesita admitir transparencia: podemos tener la tentación de agregar un nuevo campo que indique su transparencia:

```
class Color {
  #valores;

  constructor(r, g, b, a = 1) {
    this.#valores = [r, g, b, a];
  }

  get alfa() {
    return this.#values[3];
  }

  set alfa(valor) {
    if ( valor < 0 || valor > 1) {
      throw new RangeError("El canal alfa debe estar entre 0 y 1");
    }
    this.#valores[3] = valor;
  }
}
```

- sin embargo, esto significa
  - todas las instancias, incluso la gran mayoría que no son transparentes (aquellas con un valor alfa de 1), deberán tener el valor alfa adicional, que no es muy elegante.
  - además, si las características siguen creciendo, nuestra clase `Color` se volverá muy hinchada y difícil de mantener.
- en cambio, en la programación orientada a objetos, crearíamos una **clase derivada**.
  - la clase derivada tiene acceso a todas las **propiedades públicas** de la **clase padre**.
  - en JavaScript, las clases derivadas se declaran con una **cláusula** `extends`, que indica la clase de la que se extiende.

```
class ColorConAlfa extends Color {
  #alfa;

  constructor(r, g, b, a) {
    super(r, g, b);
    this.#alfa = a;
  }

  get alfa() {
    return this.#alfa;
  }

  set alfa(valor) {
    if ( valor < 0 || valor > 1) {
      throw new RangeError("El canal alfa debe estar entre 0 y 1");
    }
    this.#alfa = valor;
  }
}
```

Observando el código anterior, hay algunas cosas que tenemos que entender:

`super` y `this`

Primero es que en el constructor, estamos llamando a `super(r, g, b)`.

- es un requisito del lenguaje llamar a `super()` antes de poder utilizar `this`.
  - la llamada `super()` llama al **constructor de la clase principal** para inicializar `this`: aquí es aproximadamente equivalente a `this = new Color(r, g, b)`.
- puedes tener código antes de `super()`, pero no puedes acceder a `this` antes de `super()` : el intérprete/compilador te impide acceder al `this` no inicializado.
- después de que la clase principal termine de modificar `this`, la **clase derivada** puede hacer su propio trabajo de inicialización.
- aquí agregamos un campo privado llamado `#alpha` y también proporcionamos un par de getter/setters para interactuar con ellos.

anulación del método

Una **clase derivada** hereda todos los métodos de su padre.

- por ejemplo, aunque `ColorConAlfa` no declara un método accesor `get rojo()`, aún se puede acceder a `rojo` porque este comportamiento lo especifica la **clase principal**:

```
const color = new ColorConAlfa(255, 0, 0, 0.5);
console.log(color.rojo);                              // 255
```

Las clases derivadas también pueden **sobreescribir** los métodos de la clase principal.

- por ejemplo, todas las clases heredan implícitamente de la clase `Object`, que define algunos métodos básicos como, por ejemplo, `toString()`.
- sin embargo, el **método base** `toString()` es notoriamente inútil, porque imprime `[object Object]` en la mayoría de los casos:

```
console.log(color.toString()); // [objeto Objeto]
```

  - en cambio, nuestra clase puede sobreescribirlo para imprimir los valores RGB del color:

```
class Color {
  #valores;
  // …
  toString() {                          // sobreescritura de método = method overriding
    return this.#valores.join(", ");
  }
}

console.log(new Color(255, 0, 0).toString()); // '255, 0, 0'
```

Acceso a los miembros de la clase padre

Dentro de las **clases derivadas**, puede acceder a los métodos de la **clase principal** usando `super`.

- esto le permite crear métodos ampliados/mejorados/adaptados y evitar la duplicación de código.

```
class ColorConAlfa extends Color {
  #alfa;
  // …
  toString() {
    // Para programar el nuevo método usamos una llamada al método de la clase madre toString()
    return `${super.toString()}, ${this.#alfa}`;
  }
}

console.log(new ColorConAlfa(255, 0, 0, 0.5).toString()); // '255, 0, 0, 0.5'
```

Cuando usas `extends`, los **métodos estáticos** también se heredan entre sí, por lo que también puede sobreescribirlos o mejorarlos.

```
class ColorConAlfa extends Color {
  // ...
  static esValido(r, g, b, a) {
```

```
    // Ampliamos el método de la clase madre isValid() haciendo uso de él
    return super.esValido(r, g, b) && (a >= 0 && a <= 1);
  }
}

console.log(ColorConAlfa.esValido(255, 0, 0, -1)); // falso
```

Las **clases derivadas** no tienen acceso a los **campos privados** de la clase principal:

- este es otro aspecto clave para que los campos privados de JavaScript sean "**privados estrictos**".
- los campos privados tienen como alcance el propio **cuerpo de la clase** y no otorgan acceso a *ningún* código externo.

```
class ColorConAlfa extends Color {
  log() {
    console.log(this.#values);
    // SyntaxError: Private field '#values' must be declared in an enclosing class
  }
}
```

HERENCIA MULTIPLE

Una clase solo puede **extenderse** o **heredar** de otra única clase.

- esto evita problemas de **herencia múltiple** como el **problema del diamante**.
- sin embargo, debido a la naturaleza dinámica de JavaScript, aún es posible lograr el efecto de herencia múltiple a través de la **composición de clases** y **mixins**.
- instancias de las clases derivadas también son instancias de la clase base.

```
const color = new ColorConAlfa(255, 0, 0, 0.5);
console.log(color instanceof Color);          // true
console.log(color instanceof ColorConAlfa);   // true
```

¿Por qué clases?

Hasta ahora hemos seguido un estudio pragmático: nos hemos centrado en *cómo* se pueden usar las clases, pero hay una pregunta sin respuesta: *¿por qué* se usaría una clase? La respuesta es: depende.

- las clases introducen un ***paradigma*** o una forma de organizar su código.
- las clases son los cimientos de la programación orientada a objetos, que se basa en conceptos como la **herencia** y el **polimorfismo** (especialmente el ***polimorfismo de subtipo***).
- sin embargo, muchas personas están filosóficamente en contra de ciertas prácticas de OOP y, como resultado, no usan clases.

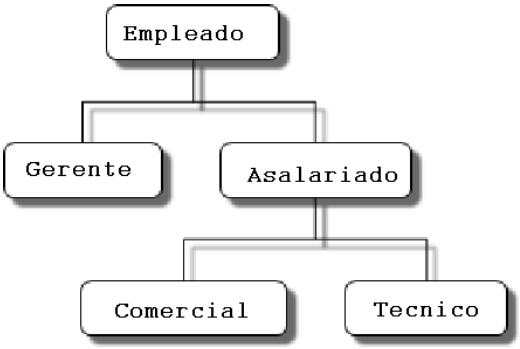
Por ejemplo, una cosa que hace que los objetos `Date` sean infames es que son **mutables**.

```
function incrementarDia(fecha) {
  return fecha.setDate(fecha.getDate() + 1);
}

const fecha = new Date();                // 2019-06-19
const nuevoDia = incrementarDia(fecha);
console.log(nuevoDia);                   // 2019-06-20

                                         // La fecha antigua también es modificada!?
console.log(fecha);                      // 2019-06-20
```

- **mutabilidad** y **el estado interno** son aspectos importantes de la programación orientada a objetos, pero a menudo dificultan razonar con el código: cualquier operación aparentemente inocente puede tener **efectos secundarios** (***side-effects***) inesperados y cambiar el comportamiento en otras partes del programa.

Para reutilizar el código, solemos recurrir a la **extensión de clases**, lo que puede crear grandes **jerarquías de patrones de herencia**:

- el siguiente diagrama ilustra una jerarquía de **herencia simple** (cada clase hereda solamente de otra clase)
- la **herencia múltiple** (una clase puede heredar de dos o más clases) presenta numerosos problemas, que no vamos a cubrir aquí (algunos lenguajes ofrecen las **interfaces** o los **traits** como solución alternativa a la herencia múltiple).



- sin embargo, a menudo es difícil describir la herencia claramente cuando una clase solo puede extender otra clase.
- a menudo, queremos disponer en una clase A del comportamiento de varias otras clases al crearla (herencia múltiple)
- en Java, esto se hace a través de interfaces; en JavaScript, se puede hacer a través de **mixins** (pero, después de todo, la tienen también sus desventajas).

En el lado positivo, las clases son una forma muy poderosa de organizar nuestro código en un nivel superior.

- por ejemplo, sin la clase `Color`, es posible que necesitemos crear una docena de funciones de utilidad:

```
function esRojo(color) {
  return color.red === 255;
}

function esUnColorValido(color) {
  return (
    color.red >= 0 &&
    color.red <= 255 &&
    color.green >= 0 &&
    color.green <= 255 &&
    color.blue >= 0 &&
    color.blue <= 255
  );
}
// ...
```

- pero usando clases, podemos situar a todos bajo el espacio de nombres `Color` espacio de nombres, lo que mejora la **legibilidad** del código.
- además, la introducción de **campos privados** nos permite ocultar ciertos datos a los usuarios finales de nuestra clase, creando una API limpia.

En general:

- debes considerar el uso de clases cuando desees crear objetos que almacenen sus propios datos internos y expongan una gran cantidad de comportamiento
- tomemos las clases integradas de JavaScript como ejemplos:
  - Las clases `Map` y `Set` almacenan una colección de elementos (implementados de una forma que no podemos ver, pues está **oculta**) y te permiten acceder a ellos por clave a través de una API o interfaz usando `get()`, `set()`, `has()`, etc.
  - La clase `Date` almacena una fecha como una marca de tiempo de Unix (un número) y te permite formatear, actualizar y leer componentes de fecha individuales.
  - La clase `Error` almacena información sobre una excepción concreta, incluido el mensaje de error, el seguimiento de la pila, la causa, etc.
    - es una de las pocas clases que vienen con una rica estructura de herencia: hay varias clases integradas como `TypeError` y `ReferenceError` que amplían `Error`.
    - en el caso de errores, esta herencia permite refinar la semántica de los errores: cada clase de error representa un tipo específico de error, que se puede comprobar fácilmente con `instanceof`.

JavaScript ofrece el mecanismo para organizar tu código de una manera orientada a objetos canónica, pero la decesión de si usarlo y cómo usarlo queda totalmente a discreción del programador.