**Programación orientada a objetos en JavaScript**

**Presentado a:** Instructor César Marino Cuéllar Chacón

**Por Aprendiz: Laura Sofia Agredo**

**Ficha:** 3064975

**Competencia:** Diseñar la solución de software de acuerdo con procedimientos y requisitos técnicos

**Resultado de Aprendizaje:** Verificar los entregables de la fase de diseño del software de acuerdo con lo establecido en el informe de análisis

Tecnólogo en Análisis y Desarrollo de Software

Servicio Nacional de Aprendizaje SENA

Centro de Teleinformática y Producción Industrial

Regional Cauca

Popayán, día **31** de **agosto** del año **2025**

**Tabla de Contenido**

[1. Actividad o Ejercicio1 3](#_Toc197272968)

[1.1 Enunciado 3](#_Toc197272969)

[1.2 Solución 3](#_Toc197272970)

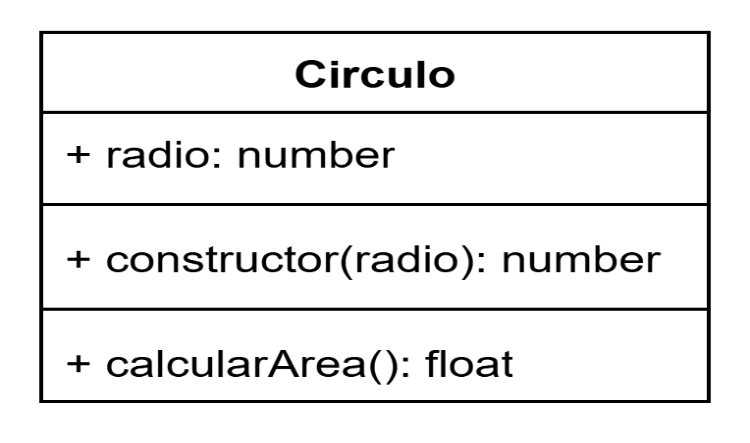
[2. Actividad o Ejercicio2 4](#_Toc197272971)

[2.1 Enunciado 4](#_Toc197272972)

[2.2 Solución 4](#_Toc197272973)

[3. Bibliografía 5](#_Toc197272974)

# **Actividad o Ejercicio 1 Constructores y métodos Unidad 2.**



class Circulo {

  /\*\*

   \* Constructor de la clase

   \* @param *{float}* *radio*

   \*/

  constructor(*radio*) {

*this*.radio = *radio*;

  }

  /\*\*

   \* Método que calcula el área

   \* del círculo

   \* @returns area

   \*/

  calcularArea() {

    return Math.PI \* *this*.radio \* *this*.radio;

  }

}

const radio = 5;

const circulo = **new** Circulo(radio);

const areaCirculo = circulo.calcularArea();

console.log(`El área del círculo de radio ${radio} es: ${areaCirculo.toFixed(2)}`);

# **Enunciado**

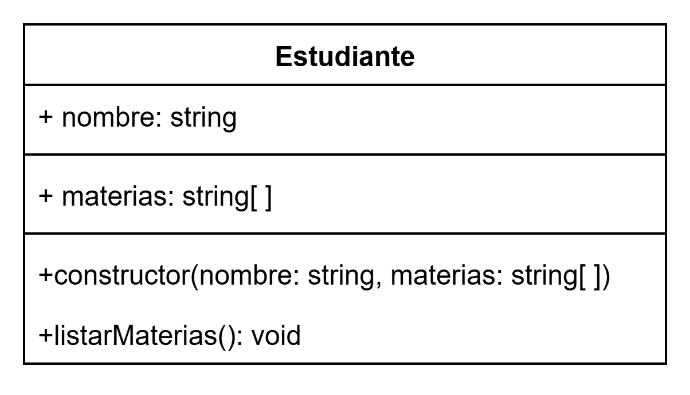
Crea una clase **Círculo** en JavaScript que tiene como atributo el radio. La clase debe contar con un constructor que reciba dicho valor y un método **calcularArea()** que retorne el área del círculo. Posteriormente, se crea un objeto de la clase y muestra en consola el área correspondiente.

# **Solución**

Se implementó la clase **Círculo** en JavaScript, con el atributo radio, el constructor y el método calcularArea().  
A continuación, se presenta el diagrama UML de la clase y el código fuente con un ejemplo de uso:

* **Diagrama UML**: muestra la clase Círculo con su atributo y métodos.
* **Código JS**: se crea un objeto con radio = 5 y se calcula el área, mostrando el resultado en consola.

# **Actividad o Ejercicio 2 Clase Estudiante.**



class Estudiante {

  /\*\*

   \* @param *{string}* *nombre*

   \* @param *{string[]}* *materias*

   \*/

  constructor(*nombre*, *materias* = []) {

*this*.nombre = *nombre*;

*this*.materias = *materias*;

  }

  /\*\*

   \* Lista las materias del estudiante

   \*/

  listarMaterias() {

    console.log(`Materias de ${*this*.nombre}: ${*this*.materias.join(", ")}`);

  }

}

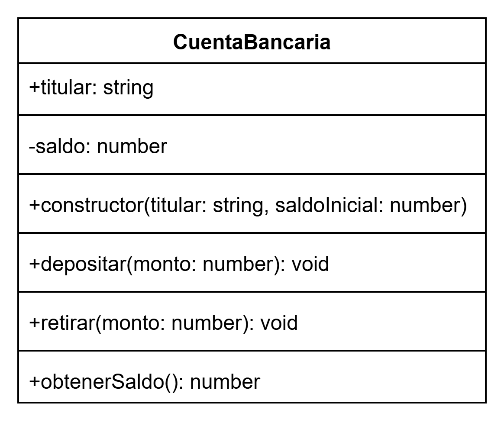
# **Enunciado**

Creo una clase **Estudiante** en JavaScript que tenga como atributos el nombre y una lista de materias. La clase debe contar con un constructor que inicialice estos valores y un método **listarMaterias()** que muestre en consola las materias del estudiante.

# **Solución**

Se implementó la clase **Estudiante** en JavaScript, con los atributos nombre y materias, además del método listarMaterias().  
Se creó un objeto de la clase, inicializando el nombre y un arreglo de materias, y se ejecutó el método para mostrar en consola las materias registradas.

# **Actividad o Ejercicio 3 Correo con Getter y Setter Unidad 3.**



class CuentaBancaria {

  /\*\*

   \* Definición de la propiedad saldo como privada

   \*/

  #saldo;

  /\*\*

   \* @param *{string}* *titular*

   \* @param *{number}* *saldoInicial*

   \*/

  constructor(*titular*, *saldoInicial*) {

*this*.titular = *titular*;

*this*.#saldo = *saldoInicial*;

  }

  /\*\*

   \* Método que permite realizar el proceso

   \* de depositar un valor a la cuenta

   \* @param *{number}* *monto*

   \*/

  depositar(*monto*) {

    if (*monto* > 0) {

*this*.#saldo += *monto*;

      console.log(`Se depositaron $${*monto*}. Saldo actual: $${*this*.#saldo}`);

    } else {

      console.log("El monto debe ser mayor que cero.");

    }

  }

  /\*\*

   \* Método que permite retirar dinero

   \* siempre y cuando no supere el saldo

   \* @param *{number}* *monto*

   \*/

  retirar(*monto*) {

    if (*monto* > 0 && *monto* <= *this*.#saldo) {

*this*.#saldo -= *monto*;

      console.log(`Se retiraron $${*monto*}. Saldo actual: $${*this*.#saldo}`);

    } else {

      console.log("Fondos insuficientes o monto inválido.");

    }

  }

  /\*\*

   \* Método que retorna el saldo actual

   \* @returns *{number}*

   \*/

  obtenerSaldo() {

    return *this*.#saldo;

  }

}

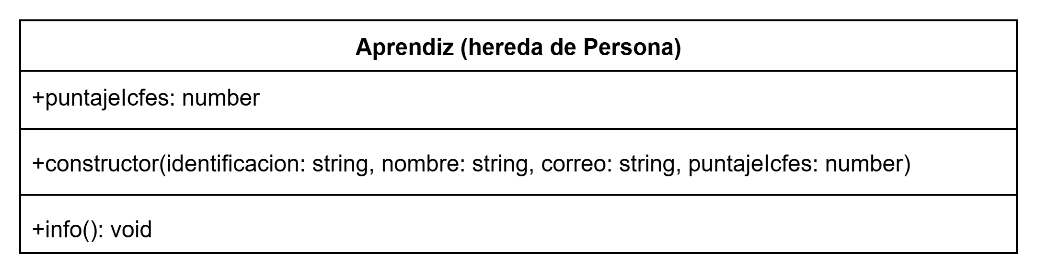
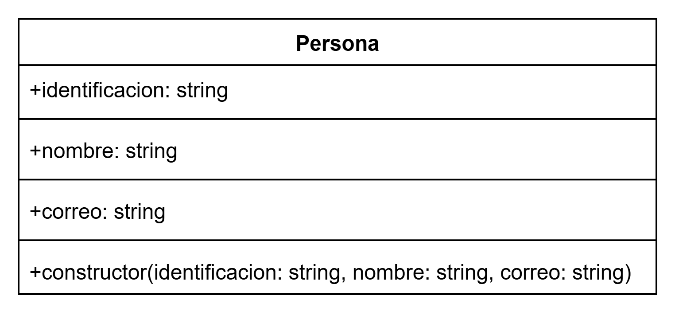
**Enunciado.**

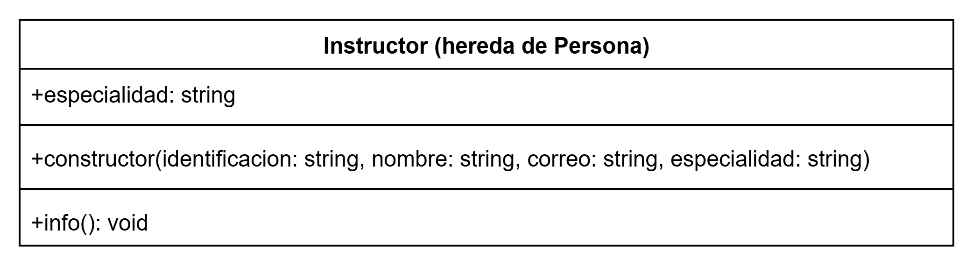
# En este ejercicio debía trabajar con la clase **CuentaBancaria** usando un atributo privado para el saldo. La idea era crear un método que permitiera retirar dinero siempre y cuando el monto no fuera mayor al saldo disponible. Después de hacer el retiro, se debe mostrar el saldo actualizado por consola.

# **Solución**

# Lo resolví creando la clase **CuentaBancaria** en JavaScript con un atributo privado #saldo. A la clase le agregué el constructor, el método depositar() y el método obtenerSaldo(). Para cumplir con la actividad implementé también el método retirar(), que revisa si el monto es válido y no supera el saldo antes de descontarlo. Con esto se logra controlar mejor el acceso al atributo privado y evitar retiros incorrectos.

**Ejercicio 1 – Persona, Instructor y Aprendiz Unidad 4.**





class Persona {

  constructor(*identificacion*, *nombre*, *correo*) {

*this*.identificacion = *identificacion*;

*this*.nombre = *nombre*;

*this*.correo = *correo*;

  }

}

// Subclase Aprendiz

class Aprendiz extends Persona {

  constructor(*identificacion*, *nombre*, *correo*, *puntajeIcfes*) {

*super*(*identificacion*, *nombre*, *correo*);

*this*.puntajeIcfes = *puntajeIcfes*; }

info() {

    console.log(`Aprendiz:

      ID: ${*this*.identificacion}

      Nombre: ${*this*.nombre}

      Correo: ${*this*.correo}

      Puntaje ICFES: ${*this*.puntajeIcfes}`);

  }

}

// Subclase Instructor

class Instructor extends Persona {

  constructor(*identificacion*, *nombre*, *correo*, *especialidad*) {

*super*(*identificacion*, *nombre*, *correo*);

*this*.especialidad = *especialidad*;

  }

  info() {

    console.log(`Instructor:

      ID: ${*this*.identificacion}

      Nombre: ${*this*.nombre}

      Correo: ${*this*.correo}

      Especialidad: ${*this*.especialidad}`);

  }

}

const aprendiz1 = **new** Aprendiz("123", "Sebastián", "sebas@mail.com", 350);

const instructor1 = **new** Instructor("456", "Carlos", "carlos@mail.com", "POO");

aprendiz1.info();

instructor1.info();

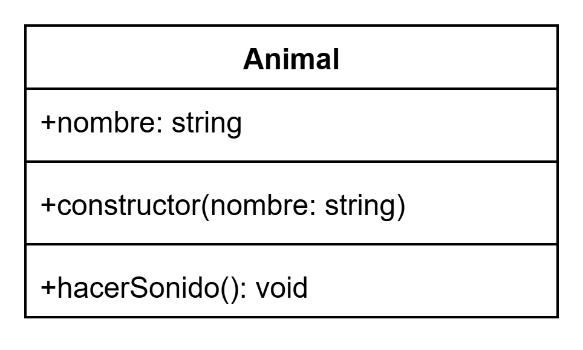
**Enunciado**

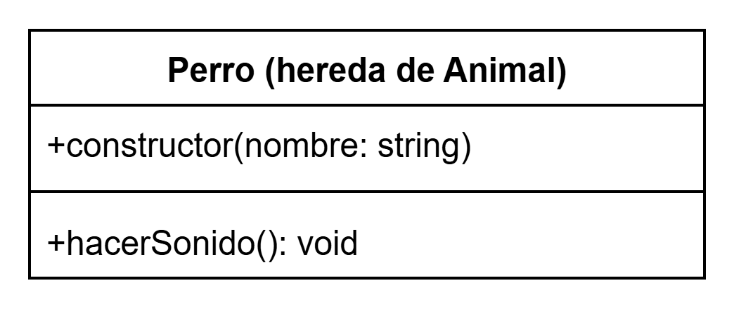
Crear una clase **Persona** con los atributos identificación, nombre y correo. A partir de ella, crear dos clases hijas: **Aprendiz** con atributo puntajeIcfes y **Instructor** con atributo especialidad. Cada una debe tener un método info() que muestre en consola todos sus atributos.

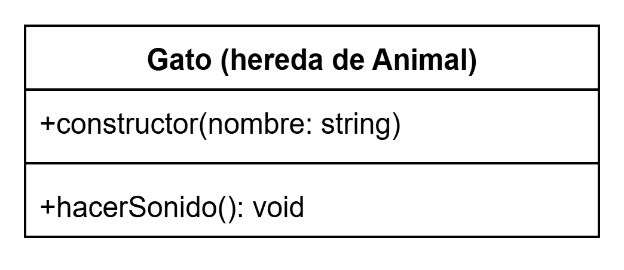
**Solución**

Se desarrolló la clase **Persona** como clase base con los atributos identificación, nombre y correo. Luego se implementaron las clases hijas **Aprendiz** y **Instructor**, que heredan de Persona mediante extends. Cada clase define su propio atributo adicional (puntajeIcfes o especialidad) y un método info() que imprime todos sus atributos en consola. Se probaron los objetos creados mostrando los datos completos de cada instancia.

**Ejercicio 2 Animal y clases hijas Perro, Gato, con comportamiento específico.**







class Animal {

  constructor(*nombre*) {

*this*.nombre = *nombre*;

  }

  hacerSonido() {

    console.log(`${*this*.nombre} hace un sonido genérico.`);

  }

}

// Subclase Perro

class Perro extends Animal {

  constructor(*nombre*) {

*super*(*nombre*);

  }

  hacerSonido() {

    console.log(`${*this*.nombre} ladra: ¡Guau guau!`);

  }

}

// Subclase Gato

class Gato extends Animal {

  constructor(*nombre*) {

*super*(*nombre*);

  }

  hacerSonido() {

    console.log(`${*this*.nombre} maúlla: ¡Miau miau!`);

  }

}

const perro1 = **new** Perro("Max");

const gato1 = **new** Gato("Michi");

perro1.hacerSonido();

gato1.hacerSonido();

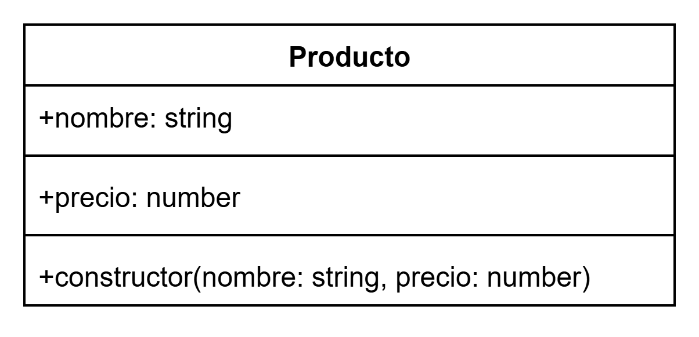
**Enunciado**

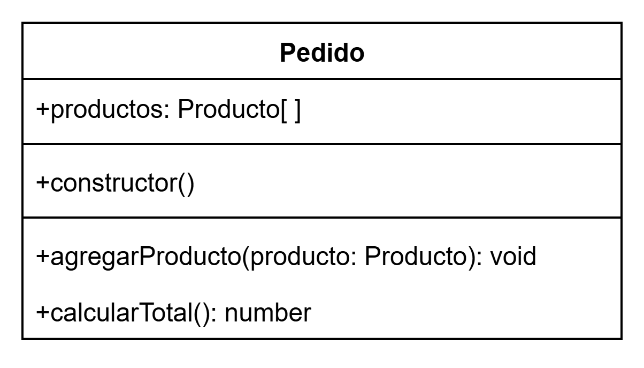
En esta actividad debía crear una clase Animal con un atributo para el nombre y un método hacerSonido(). A partir de esa clase, tenía que implementar las clases hijas Perro y Gato, cada una con su propio comportamiento específico, sobrescribiendo el método hacerSonido().

**Solución**

La solución fue crear la clase Animal en JavaScript con un atributo nombre y un método hacerSonido() genérico. Luego implementé las subclases Perro y Gato usando herencia con extends. En cada subclase redefiní el método hacerSonido() para mostrar sonidos característicos: el perro ladra y el gato maúlla. Para probarlo instancié un objeto de cada clase y ejecuté sus métodos en la consola, mostrando el comportamiento esperado.

**Ejercicio 1 UNIDAD 5: Composición y Relaciones.**

****

****

class Producto {

  constructor(*nombre*, *precio*) {

*this*.nombre = *nombre*;

*this*.precio = *precio*;

  }

}

// Clase Pedido (contiene productos)

class Pedido {

  constructor() {

*this*.productos = [];

  }

  agregarProducto(*producto*) {

*this*.productos.push(*producto*);

    console.log(`Se agregó el producto: ${*producto*.nombre} ($${*producto*.precio})`);

  }

  calcularTotal() {

    let total = 0;

    for (const producto of *this*.productos) {

      total += producto.precio;

    }

    return total;

  }

}

const pedido1 = new Pedido();

const prod1 = new Producto("Laptop", 3000);

const prod2 = new Producto("Mouse", 100);

const prod3 = new Producto("Teclado", 200);

pedido1.agregarProducto(prod1);

pedido1.agregarProducto(prod2);

pedido1.agregarProducto(prod3);

console.log("Total del pedido: $" + pedido1.calcularTotal());

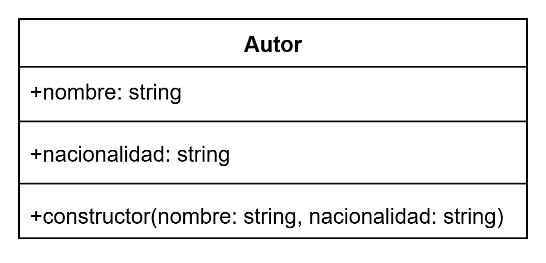
**Enunciado**

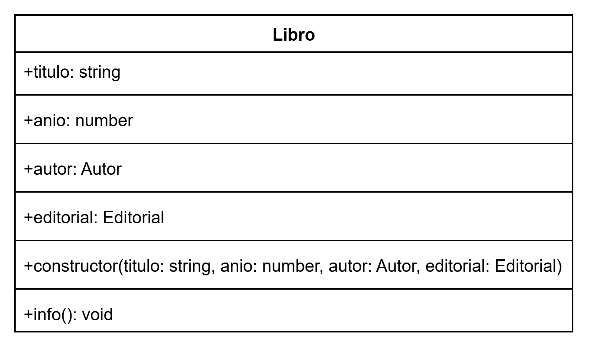
Crear una clase Producto con atributos nombre y precio. Luego, crear la clase Pedido que contenga una lista de productos, un método agregarProducto() para añadir productos y un método calcularTotal() que sume el precio de todos los productos.

**Solución**

La solución fue implementar la clase Producto con los atributos básicos y la clase Pedido con un arreglo de productos. Se añadieron los métodos agregarProducto() y calcularTotal(). En la prueba se creó un pedido, se agregaron varios productos y se calculó el total, mostrando el resultado en consola.

**Ejercicio 2 – Biblioteca Libro, Autor, Editorial.**





class Autor {

  constructor(*nombre*, *nacionalidad*) {

*this*.nombre = *nombre*;

*this*.nacionalidad = *nacionalidad*;

  }

}

// Clase Editorial

class Editorial {

  constructor(*nombre*, *ciudad*) {

*this*.nombre = *nombre*;

*this*.ciudad = *ciudad*;

  }

}

// Clase Libro (compuesta por Autor y Editorial)

class Libro {

  constructor(*titulo*, *anio*, *autor*, *editorial*) {

*this*.titulo = *titulo*;

*this*.anio = *anio*;

*this*.autor = *autor*;

*this*.editorial = *editorial*;

  }

  info() {

    return `Libro: "${*this*.titulo}" (${*this*.anio})

    Autor: ${*this*.autor.nombre} - ${*this*.autor.nacionalidad}

    Editorial: ${*this*.editorial.nombre}, ${*this*.editorial.ciudad}`;

  }

}

const autor1 = **new** Autor("Gabriel García Márquez", "Colombiano");

const editorial1 = **new** Editorial("Sudamericana", "Buenos Aires");

const libro1 = **new** Libro("Cien años de soledad", 1967, autor1, editorial1);

console.log(libro1.info());

**Enunciado**

Crear una clase **Autor** con atributos nombre y nacionalidad, una clase **Editorial** con atributos nombre y ciudad, y una clase **Libro** que contenga como atributos el título, el año, un Autor y una Editorial. La clase Libro debe tener un método info() que muestre en consola todos los datos del libro, incluyendo los del autor y la editorial.

**Solución**

La solución fue implementar las clases **Autor** y **Editorial** con sus atributos respectivos, y la clase **Libro** que las contiene como parte de su composición. En el método info() de la clase Libro se muestran en consola el título, el año, el autor con su nacionalidad y la editorial con su ciudad. En la prueba se creó un autor, una editorial y un libro, mostrando la información completa en consola.

# **Bibliografía**

Relacionar aquí la bibliografía utilizada.

* Piñeiro Gómez, José Manuel. Diseño de bases de datos relacionales. Editorial Paraninfo, 2014. 164 páginas.
* Bases de datos, sitio web: <https://josejuansanchez.org/bd/>