Javascript

Juan Carlos Trejo

1

Módulo 1

Introducción

Variables

Hay ocho tipos de datos básicos en JavaScript. Podemos almacenar un valor de cualquier tipo dentro de una variable. Por ejemplo, una variable puede contener en un momento un string y luego almacenar un número.

```
1  // no hay error
2  let message = "hola";
3  message = 123456;
```

Variables

Number	Para números de cualquier tipo: enteros o de punto flotante, los enteros están limitados por $\pm(2^{53}-1)$.
Bigint	Para números enteros de longitud arbitraria.
String	Para cadenas. Una cadena puede tener cero o más caracteres, no hay un tipo especial para un único carácter.
boolean	Para verdadero y falso: true/false.
Null	Para valores desconocidos – un tipo independiente que tiene un solo valor nulo: null
Undefined	Para valores no asignados – un tipo independiente que tiene un único valor indefinido: undefined
Symbol	Para identificadores únicos.
Object	Para estructuras de datos complejas.

Variables

Una variable es un “almacén con un nombre” para guardar datos. Existen 3 formas de declarar una variable:

1. let
2. const
3. var

Variables
let

Formas de declarar una variable con let

```
let message;
message = 'Hola!';
alert(message);
```

```
let user = 'John';
let age = 25;
let message = 'Hola';
```

```
let user = 'John', age = 25, message = 'Hola';
```

```
let user = 'John',
  age = 25,
  message = 'Hola';
```

Variables

const

Una variable de tipo const es inmutable, por lo cual solo se puede guardar una vez y sólo una vez un valor. Tales constantes se nombran utilizando letras mayúsculas y guiones bajos.

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";
```

```
const COLOR_RED = "#F00";
COLOR_RED = "#0F0";
```

Variables
var

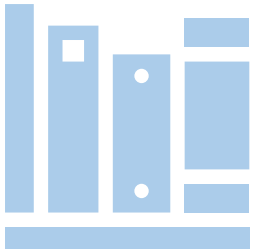
Las variables declaradas con var pueden tener a la función como entorno de visibilidad, o bien ser globales. Su visibilidad atraviesa los bloques.

```
if (true) {
  var test = true;
}

alert(test)
```

```
if (true) {
  let test = true;
}

alert(test);
```

Variables
objetos

Los objetos son usados para almacenar colecciones de varios datos y entidades más complejas asociados con un nombre clave. En JavaScript, los objetos penetran casi todos los aspectos del lenguaje.



Variables
objetos

Los objetos son usados para almacenar colecciones de varios datos y entidades más complejas asociados con un nombre clave. En JavaScript, los objetos penetran casi todos los aspectos del lenguaje.

```
// Sintaxis de "constructor de objetos"
let usuario = new Object();
// Sintaxis de "objeto literal"
let usuario = {};
```

Variables
objetos

Se puede declarar un objeto y agregar propiedades. Una propiedad tiene una clave (también conocida como “nombre” o “identificador”) antes de los dos puntos ":" y un valor a la derecha.

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};
```

Variables
objetos

Se puede declarar un objeto y agregar propiedades. Una propiedad tiene una clave (también conocida como “nombre” o “identificador”) antes de los dos puntos ":" y un valor a la derecha.

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};
```

```
// Obteniendo los valores
alert( user.name ); // John
alert( user.age ); // 30
```

Variables
objetos

1. Agregar una propiedad

```
user.isAdmin = false;
alert(user);
```

2. Eliminar una propiedad

```
delete user.name;
alert(user);
```

Variables
objetos

3. Acceder a una propiedad

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};


console.log(user.age);
console.log(user['name']);
```

Variables
Colecciones

```
// Crear el mapa
const map = new Map();
// Asignar un primer valor
map.set('nombre', 'Juan');
// Crear un objeto y asignarlo
// al mapa
const comidas = {
    dinner: 'Curry',
    lunch: 'Sandwich',
    breakfast: 'Eggs' };

const normalfoods = {};

map.set(normalfoods, comidas);
// Iterar sobre la coleccion
for (const [key, value] of map) {
  console.log(key, value);
}
// Eliminar el contenido
map.clear();
```

Los **mapas** son colecciones de claves y valores de cualquier tipo.

Variables
Colecciones

Métodos para listas de tipo Map

Método	Descripción
set(key, value)	Asigna un valor a una clave en el mapa.
get(key)	Devuelve el valor asociado con la clave proporcionada.
delete(key)	Elimina la entrada correspondiente a la clave especificada.
has(key)	Devuelve un booleano indicando si el mapa contiene una clave específica.
clear()	Elimina todas las entradas del mapa.
size	Propiedad que devuelve la cantidad de entradas en el mapa.
forEach(callbackFn, thisArg?)	Ejecuta una función proporcionada una vez por cada par clave-valor en el mapa.

Variables
Colecciones

Los **conjuntos** son listas ordenadas de valores que no contienen duplicados. En lugar de indexarse como lo hacen las matrices, se accede a los conjuntos mediante claves.

```
// Crear el conjunto
const planetas = new Set();
//Agregar valores al conjunto
planetas.add('Mercurio');
planetas.add('Tierra').add('Marte');
// Determinar si el conjunto posee
// algun elemento en particular
console.log(planetas.has('Tierra'));
// Eliminar un elemento del conjunto
planetas.delete('Marte');
console.log(planetas.has('Marte'));
// Iterar sobre el conjunto
for (const planet of planetas) {
  console.log(planet);
}
// Obtener el tamanio del conjunto
console.log(planetas.size);
// Eliminar el contenido
planetas.clear();
console.log(planetas.size);
```

Variables
Colecciones

Métodos para listas de tipo Set

Método	Descripción
add(value)	Añade un nuevo elemento al conjunto.
delete(value)	Elimina un elemento del conjunto según su valor.
has(value)	Devuelve un booleano indicando si el conjunto contiene un elemento específico.
clear()	Elimina todos los elementos del conjunto.
size	Propiedad que devuelve la cantidad de elementos en el conjunto.
forEach(callbackFn, thisArg?)	Ejecuta una función proporcionada una vez por cada valor en el conjunto.

2

Módulo 2

Arreglos

Arreglos

Es una estructura o colección para insertar datos de forma ordenanda.

```
let arreglo = new Array();
let arreglo = [];
```

```
let frutas = ["Apple", "Orange", "Plum"];

alert( frutas[0] ); // Apple
alert( frutas[1] ); // Orange
alert( frutas[2] ); // Plum
```

Arreglos

Podemos acceder a los métodos de manipulación de arreglos a través de:

```
// Metodos estaticos
Array.method()
// Metodos de instancia
Array.prototype.method()
```

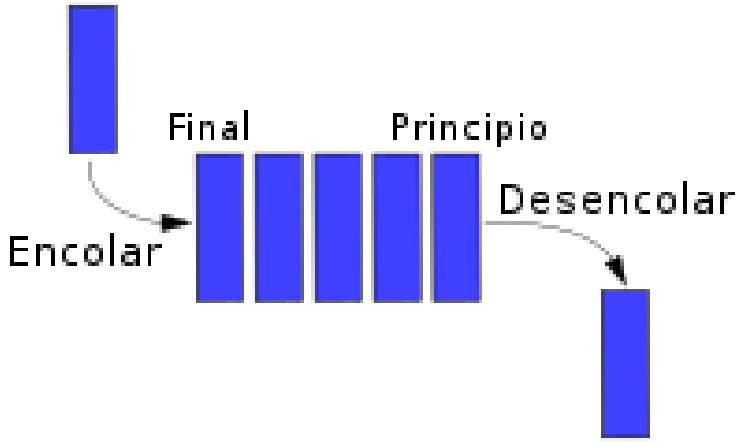
Arreglos

Principales métodos

Método	Descripción
push()	Añade uno o más elementos al final del arreglo.
pop()	Elimina el último elemento del arreglo.
shift()	Elimina el primer elemento del arreglo.
unshift()	Añade uno o más elementos al principio del arreglo.
splice()	Permite agregar o eliminar elementos de cualquier posición en el arreglo.
filter()	Retorna una porción del arreglo en base a una condición.
concat()	Combina dos o más arreglos.
forEach()	Itera sobre cada elemento del arreglo y ejecuta una función proporcionada.
map()	Crea un nuevo arreglo con los resultados de aplicar una función a cada elemento del arreglo.
find()	Retorna el primer elemento que cumple con una condición dada en la función proporcionada.
findIndex()	Retorna el índice del primer elemento que cumple con una condición dada en la función proporcionada.
keys()	Retorna un nuevo objeto Iterador que contiene las claves de índice del arreglo.
values()	Retorna un nuevo objeto Iterador que contiene los valores del arreglo.
fill()	Rellena todos los elementos de un arreglo con un valor estático.
from()	Crea un nuevo arreglo a partir de un objeto iterable o de un arreglo similar a un arreglo.

Arreglos

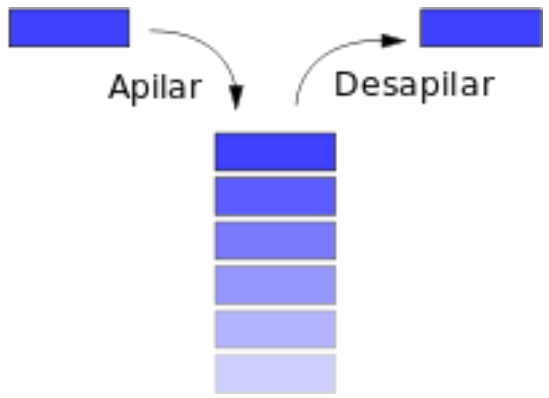
Push/Shift




```
// Creamos un arreglo inicial
let miArreglo = [1, 2, 3, 4, 5];

// Utilizamos el método 'shift' para
// eliminar el primer elemento del arreglo
let primerElemento = miArreglo.shift();

// Utilizamos el método 'push' para
// agregar elementos al final del arreglo
miArreglo.push(6, 7);

console.log("Arreglo final:", miArreglo);
```

Arreglos
Push/Pop

```
// Creamos un arreglo inicial
let miArreglo = ["manzana", "banana", "cereza"];

// Utilizamos el método 'push' para
// agregar un elemento al final del arreglo
miArreglo.push("durazno");

// Utilizamos el método 'pop' para
// eliminar el último elemento del arreglo
let ultimoElemento = miArreglo.pop();

// Salida final
console.log("Arreglo final:", miArreglo);
```



Arreglos

Map

El método map crea un nuevo arreglo que contiene los valores resultantes de aplicar la operación a cada elemento del arreglo original.

```
// Creamos un arreglo de números
let numeros = [1, 2, 3, 4, 5];

// Utilizamos el método 'map' para realizar
// una operación en cada elemento del arreglo
let duplicados = numeros.map(function(numero) {
  return numero * 2;
});
```

Arreglos
Filter

El método filter en JavaScript se utiliza para crear un nuevo arreglo con todos los elementos que cumplan cierta condición especificada en una función de filtrado.

```
// Creamos un arreglo de números
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Utilizamos el método 'filter' para obtener
// los números pares del arreglo
let numerosPares = numeros.filter(function(numero) {
  return numero % 2 === 0;
});

console.log("Números pares:", numerosPares);
```

Arreglos

find/findIndex

Retorna el primer elemento que cumple con una condición dada en la función proporcionada.

```
// Definir un arreglo de objetos
let personas = [
  { nombre: 'Juan', edad: 25 },
  { nombre: 'María', edad: 30 },
  { nombre: 'Ana', edad: 22 },
  { nombre: 'Carlos', edad: 28 }
];

// Utilizar la función find para encontrar la primera
// persona que cumple con cierta condición
let personaEncontrada = personas.find(function(persona) {
  return persona.edad > 25;
});

// Imprimir el resultado
console.log(personaEncontrada);
```

Arreglos
from

Array.from() toma el objeto iterable (en este caso, una cadena de texto) y crea un nuevo arreglo donde cada elemento del arreglo es un carácter de la cadena original.

```
// Crear un objeto iterable
let iterable = 'Hola';

// Usar Array.from() para crear un nuevo arreglo
let nuevoArreglo = Array.from(iterable);

// Resultado: ['H', 'o', 'l', 'a']
console.log(nuevoArreglo);
```

Arreglos
from

Se puede usar la función para transformar cada elemento durante la creación del nuevo arreglo.

```
// Crear un arreglo de números
let numeros = [1, 2, 3, 4];

// Usar Array.from() con una función de mapeo
// para duplicar cada número
let nuevoArreglo = Array.from(numeros, num => num * 2);

// Resultado: [2, 4, 6, 8]
console.log(nuevoArreglo);
```

Arreglos

Spread operator

En JavaScript, el operador de propagación (spread operator) es un operador que se introdujo en ECMAScript 6 (también conocido como ES6) y se utiliza para descomponer elementos en varias ubicaciones. Este operador se representa con tres puntos consecutivos (...).

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
// Resultado: [1, 2, 3, 4, 5, 6]
const combinedArray = [...array1, ...array2];
```

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
// Resultado: { a: 1, b: 2, c: 3, d: 4 }
const combinedObject = { ...obj1, ...obj2 };
```

3

Módulo 3

Cadenas

Cadenas
métodos

Método	Descripción
startsWith(prefix)	Retorna true si la cadena comienza con el prefijo dado, de lo contrario, retorna false.
endsWith(suffix)	Retorna true si la cadena termina con el sufijo dado, de lo contrario, retorna false.
includes(substring)	Retorna true si la cadena contiene la subcadena, de lo contrario, retorna false.
repeat(count)	Retorna una nueva cadena que consiste en la cadena original repetida count veces.
raw(strings, ...values)	Permite crear cadenas de texto sin procesar (raw strings) utilizando literales de plantilla.
padStart(length, pad)	Rellena la cadena al principio con un carácter específico hasta alcanzar la longitud deseada.
padEnd(length, pad)	Rellena la cadena al final con un carácter específico hasta alcanzar la longitud deseada.

Cadenas

startsWith/endsWith

Método startsWith

Retorna true si la cadena comienza con el prefijo dado, de lo contrario, retorna false.

```
// Metodo startsWith
let frase = "Hola, ¿cómo estás?";
let comienzaConHola = frase.startsWith("Hola");
console.log(comienzaConHola); // Resultado: true
```

Método endsWith

Retorna true si la cadena termina con el sufijo dado, de lo contrario, retorna false.

```
let nombreArchivo = "documento.pdf";
let esPDF = nombreArchivo.endsWith(".pdf");
console.log(esPDF); // Resultado: true
```

Cadenas

includes

Retorna true si la cadena contiene la subcadena, de lo contrario, retorna false

```
let mensaje = "La reunión será a las 10 de la mañana";
let contieneHora = mensaje.includes("10");
console.log(contieneHora); // Resultado: true
```

Cadenas

padStarts/padEnds

Método padStarts

Rellena la cadena al principio con un carácter específico hasta alcanzar la longitud deseada.

```
let numero = "42";
let numeroConCeros = numero.padStart(5, "0");
// Resultado: "00042"
console.log(numeroConCeros);
```

Método padEnds

Rellena la cadena al final con un carácter específico hasta alcanzar la longitud deseada.

```
let precio = "25";
let precioConPuntoCero = precio.padEnd(5, ".00");
// Resultado: "25.00"
console.log(precioConPuntoCero);
```

Cadenas
repeat/raw

Método repeat

Retorna una nueva cadena que consiste en la cadena original repetida count veces

```
let asteriscos = "*".repeat(5);
// Resultado: "*****"
console.log(asteriscos);
```

Método raw

Permite crear cadenas de texto sin procesar (raw strings) utilizando literales de plantilla

```
let nombre = "Juan";
let mensaje = String.raw`Hola, \n${nombre}!`;
// Resultado: "Hola, \\nJuan!"
console.log(mensaje);
```

4

Módulo 4

Métodos numéricos

Métodos numéricos

Método	Descripción
Number.isInteger(value)	Determina si el valor proporcionado es un número entero.
isNaN(value)	Determina si el valor proporcionado no es un número (NaN).
isFinite(value)	Determina si el valor proporcionado es un número finito.
Number.isSafeInteger(value)	Determina si el valor proporcionado es un número entero seguro en términos de representación precisa.
parseInt(string, radix)	Convierte una cadena en un número entero. El segundo parámetro especifica la base del sistema numérico (por ejemplo, 10 para decimal).
parseFloat(string)	Convierte una cadena en un número de punto flotante.

Métodos numéricos

isInteger

Método isInteger

Determina si el valor proporcionado es un número entero.

```
let entero = 42;
let esEntero = Number.isInteger(entero);
// Resultado: true
console.log(esEntero);
```

Métodos numéricos

NaN

En JavaScript, el valor NaN representa "Not a Number" (No es un número). Es un valor especial que se utiliza para indicar que una operación aritmética ha producido un resultado que no es un número válido

```
console.log(0 / 0)
console.log(Math.sqrt(-1))
console.log(NaN + 5)
```

Métodos numéricos

isNaN

Método isNaN

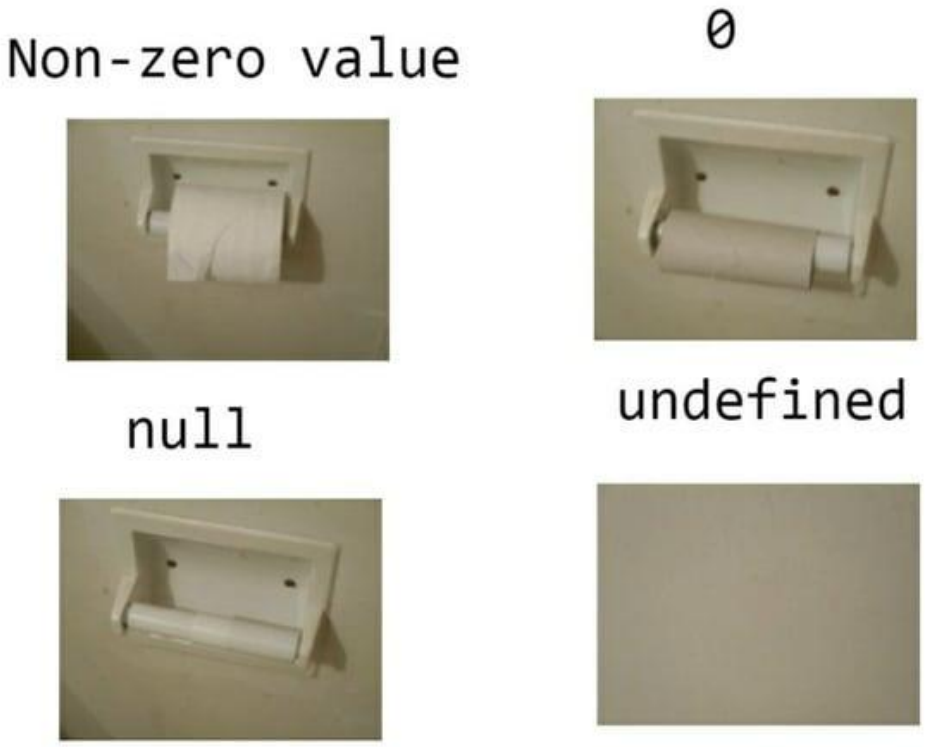
Determina si el valor proporcionado no es un número (NaN).

```
let noEsNumero = "Hola";
let esNaN = isNaN(noEsNumero);
// Resultado: true
console.log(esNaN);
```

Métodos numéricos

Valores nulos

- **Non-zero (Distinto de cero):** Este término se refiere a cualquier valor numérico que no sea igual a cero.
- **0:** Es simplemente el valor numérico cero.
- **null:** Es un valor especial en JavaScript que indica la ausencia de un objeto o valor.
- **undefined:** Es un valor que tiene una variable cuando no se le ha asignado un valor.



Métodos numéricos

En JavaScript, un valor se considera finito si es un número finito, es decir, si es un número que no es Infinity ni -Infinity. Los valores finitos incluyen enteros, decimales y fracciones que no representan infinito.

```
// Valor Finito positivo
var entero = 42;
// Valor Infinity positivo
var infinitoPositivo = Infinity;
// Valor Infinity negativo
var infinitoNegativo = -Infinity;
// Esto da como resultado Infinity
var resultadoOperacionNoFinita = 1 / 0;
// Esto da como resultado NaN
var resultadoOperacionNoFinita = 0 / 0;
```

Métodos numéricos

isFinite

Método isFinite

Determina si el valor proporcionado es un número finito.

```
let numeroInfinito = Infinity;
let esFinanciero = isFinite(numeroInfinito);
// Resultado: false
console.log(esFinanciero);
```

Métodos numéricos

Entero seguro

Un entero seguro en JavaScript es aquel que se encuentra en el rango -2^53 hasta 2^53, inclusive.

```
var enteroSeguro = Number.MAX_SAFE_INTEGER;
var noEsEnteroSeguro = Number.MAX_SAFE_INTEGER + 1;

console.log(Number.isSafeInteger(enteroSeguro));  // true
console.log(Number.isSafeInteger(noEsEnteroSeguro));  // false
```

Métodos numéricos

isSafeInteger

Método isSafeInteger

Determina si el valor proporcionado es un número entero seguro en términos de representación precisa

```
// Es el máximo entero seguro en JavaScript
let enteroSeguro = 9007199254740991;
let esEnteroSeguro = Number.isSafeInteger(enteroSeguro);
// Resultado: true
console.log(esEnteroSeguro);
```

Métodos numéricos

parseInt/parseFloat

Método parseInt

Convierte una cadena en un número entero. El segundo parámetro especifica la base del sistema numérico (por ejemplo, 10 para decimal).

```
let cadenaHexadecimal = "1a";
let enteroDesdeHex = parseInt(cadenaHexadecimal, 16);
console.log(enteroDesdeHex); // Resultado: 26
```

Método parseFloat

Convierte una cadena en un número de punto flotante.

```
let cadenaDecimal = "3.14";
let decimal = parseFloat(cadenaDecimal);
console.log(decimal); // Resultado: 3.14
```

5

Módulo 5

Funciones

Funciones

Las funciones son los principales “bloques de construcción” del programa. Permiten que el código se llame muchas veces sin repetición.

```
function name(parameter1, parameter2, ... parameterN) {
    // body
}
```

```
function showMessage() {
    alert( '¡Hola a todos!' );
}

showMessage();
showMessage();
```

Funciones

Si una variable con el mismo nombre se declara dentro de la función, le *hace sombra* a la externa.

```
let usuario = 'John';

function showMessage() {
  let usuario = "Bob"; // declara variable local

  let mensaje = 'Hello, ' + usuario;
  // ¿Cual es la salida?
  alert(mensaje);
}

// la función crea y utiliza su propia variable local usuario
showMessage();

// ¿Cual es la salida?
alert( usuario );
```

Funciones

Se pueden pasar N numero de parámetros a una función.

```
// parámetros: from, text
function showMessage(from, text) {
    alert(from + ': ' + text);
}
// Llamar la funcion
showMessage('Ann', '¡Hola!');
```

Funciones

Si una función es llamada, pero no se le proporciona un argumento, su valor correspondiente se convierte en undefined. Podemos especificar un valor llamado “predeterminado” o “por defecto” (es el valor que se usa si el argumento fue omitido) en la declaración de función usando =

```
function showMessage(from, text = "sin texto") {
    alert( from + ": " + text );
}

showMessage("Ann");
```

Funciones

En JavaScript, una función no es una estructura del lenguaje, sino un tipo de valor especial. Existe otra sintaxis para crear una función que se llama una Expresión de Función.

```
let saludar = function() {
    alert( "Hola" );
};

typeof saludar;
```

Funciones

```
function saludar() {    // a) Crear la funcion
    alert( "Hola" );
}

let copia = saludar;    // b) Hacer una copia

copia(); // Hola        // c) Ejecutar la funcion copia
saludar(); // Hola      // d) Ejecutar la funcion original
```

Funciones callback

Una función de callback es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción, permitiendo así que el código en la función callback se ejecute en un momento específico.

```
function saludar(nombre) {
    alert('Hola ' + nombre);
}

function procesarEntradaUsuario(callback) {
    var nombre = prompt('Por favor ingresa tu nombre.');
    callback(nombre);
}

procesarEntradaUsuario(saludar);
```

Funciones flecha

Una función de flecha (también conocida como "arrow function" en inglés) es una característica introducida en ECMAScript 6 (ES6) de JavaScript. Proporciona una sintaxis más concisa y simplificada para crear funciones en comparación con las funciones tradicionales.

```
(parametro1, parametro2, ...) => {
    // Cuerpo de la función
    // Puede contener una o varias instrucciones
    return resultado;
}
```

Funciones flecha

Una función de flecha (también conocida como "arrow function" en inglés) es una característica introducida en ECMAScript 6 (ES6) de JavaScript. Proporciona una sintaxis más concisa y simplificada para crear funciones en comparación con las funciones tradicionales.

```
// Funcion 1
// Recibir parametros. Funcion normal
const sumar1 = function(num1, num2) {
    return num1 + num2;
}

// Funcion 2 Recibir parametros
// Recibir parametros. Funcion flecha
const sumar2 = (num1, num2) => {
    return num1 + num2;
}

// Funcion 3 Recibir parametros
// Recibir parametros. Funcion flecha
const sumar3 = (num1, num2) => num1 + num2;
```

Funciones flecha

Una función de flecha (también conocida como "arrow function" en inglés) es una característica introducida en ECMAScript 6 (ES6) de JavaScript. Proporciona una sintaxis más concisa y simplificada para crear funciones en comparación con las funciones tradicionales.

```
// Funcion 4 Un solo parametro
const imprimir = nombre => console.log(nombre);

// Funcion 5 Sin parametros
const imprimirDocument = () => { console.log(document) }

// Funcion 6 Retornar objeto
const retornarObjeto = (id, nombre) => ({ id: id, nombre: nombre})
```

6

Módulo 6

Destructuración

Destructuring

La sintaxis de asignación de desestructuración es una expresión de JavaScript que hace posible descomprimir valores de matrices o propiedades de objetos en distintas variables.

```
// Objeto user
let user = {
    // En la clave "name" se almacena el valor "John"
    name: "John",
    // En la clave "age" se almacena el valor 30
    age: 30
};

const {nombre, edad} = producto;
```

Destructuring

La sintaxis de asignación de desestructuración es una expresión de JavaScript que hace posible descomprimir valores de matrices o propiedades de objetos en distintas variables.

```
let [variable1, variable2, ..., variableN] = arreglo;
```

```
let numeros = [1, 2, 3];

let [a, b, c] = numeros;

console.log(a); // Resultado: 1
console.log(b); // Resultado: 2
console.log(c); // Resultado: 3
```

Destructuring

También podemos asignar propiedades a variables con cualquier nombre.

```
const myObject = {
    one: 'a',
    two: 'b',
    three: 'c'
};

// ES6 destructuring example
const { one: first, two: second, three: third } = myObject;

// Ahora se puede usar las variables first, second y third
console.log(first);  // 'a'
console.log(second); // 'b'
console.log(third);  // 'c'
```

Destructuring

También se puede hacer referencia a objetos anidados más complejos.

```
const meta = {
    title: 'Destructuracion',
    authors: [{
            firstname: 'Lot',
            lastname: 'Alvarez'
        }
    ],
    publisher: {
        name: 'SitePoint',
        url: 'http://www.pagina.com/'
    }
};

const {
    title: doc,
    authors: [{ firstname: name }],
    publisher: { url: web }
} = meta;
```

Destructuring

¿Qué produce el siguiente código?

```
var a = 1, b = 2;

// Destructuracion de intercambio
[a, b] = [b, a];

console.log(a)
console.log(b)
```

Destructuring

La función **printInfo** toma un solo parámetro que es un objeto. En lugar de pasar el objeto directamente como un argumento, se está utilizando la destructuring en la propia declaración de la función para extraer las propiedades específicas del objeto **(name, age, city)** y luego imprimirlas.

```
// Definición de una función que toma un objeto como parámetro
function printInfo({ name, age, city }) {
    console.log(`Nombre: ${name}`);
    console.log(`Edad: ${age}`);
    console.log(`Ciudad: ${city}`);
}

// Objeto que será pasado como argumento a la función
const person = {
    name: 'Juan',
    age: 25,
    city: 'Mexico'
};

// Llamada a la función con destructuración del objeto
printInfo(person);
```

Destructuring

```
// Definición de una función que devuelve un
// objeto con múltiples valores
function getPersonInfo() {
    const name = 'Ana';
    const age = 30;
    const city = 'Barcelona';

    // Devolver un objeto con destructuración
    return { name, age, city };
}

// Llamada a la función y destructuración de
// los valores devueltos
const { name, age, city } = getPersonInfo();

// Imprimir los valores obtenidos
console.log(`Nombre: ${name}`);
console.log(`Edad: ${age}`);
console.log(`Ciudad: ${city}`);
```

La función **getPersonInfo** devuelve un objeto con las propiedades name, age y city. Luego, al llamar a la función y asignar el resultado a una variable utilizando destructuración, se puede acceder fácilmente a esos valores individualmente.

Destructuring

La función **forEach** se utiliza para iterar sobre cada objeto en el array libros. En el cuerpo de la función de iteración, la destructuración se utiliza para extraer las propiedades titulo, autor e isbn de cada objeto, y luego se imprimen esos valores.

```
// Array de objetos que representan libros
const libros = [
    { titulo: 'JavaScript: The Good Parts', autor: 'Douglas Crockford', isbn: '978-0596517748' },
    { titulo: 'Clean Code', autor: 'Robert C. Martin', isbn: '978-0132350884' },
    { titulo: 'The Pragmatic Programmer', autor: 'Andrew Hunt', isbn: '978-0201616224' }
  ];

  // Iterar sobre el array de libros utilizando destructuración
  libros.forEach(({ titulo, autor, isbn }) => {
    console.log('--- Libro ---');
    console.log(`Título: ${titulo}`);
    console.log(`Autor: ${autor}`);
    console.log(`ISBN: ${isbn}`);
    console.log('\n');
  });
```

7

Módulo 7

Símbolos

Símbolos

En JavaScript, los símbolos son un tipo de dato primitivo introducido en ECMAScript 2015 (también conocido como ES6). Los símbolos se utilizan para crear identificadores únicos. Cada símbolo creado es único, lo que significa que no hay dos símbolos que sean iguales, incluso si tienen el mismo nombre.

```
const simbolo1 = Symbol("simbolo 1")
const simbolo2 = Symbol(123)

// los símbolos son únicos
Symbol() === Symbol() // false
// incluso con la misma descripción
Symbol('a') === Symbol('a') // false
// comparacion de simbolos en variables
simbolo1 === simbolo2 // false
```

Símbolos

```
// Declaracion de un objeto y
// una constante de tipo Simbolo
let user = {};
let email = Symbol();

// Agregar dos propiedades al objeto
user.name = 'Juan';
user.age = 30;
// Agregar una tercera propiedad
// de tipo simbolo
user[email] = 'juan@ejemplo.com';

// <-- Array [ "name", "age" ]
Object.keys(user);
// <-- Array [ "name", "age"]
Object.getOwnPropertyNames(user);
```

Hay un par de ventajas al hacerlo:

1. Las claves basadas en símbolos nunca chocarán, a diferencia de las claves de cadena, que pueden entrar en conflicto con las claves de propiedades o métodos existentes de un objeto.

2. No se enumerarán en los bucles for... in y funciones como keys() o getOwnPropertyNames() los ignorarán.

Símbolos

Normalmente todos los Symbols son diferentes aunque tengan el mismo nombre. Pero algunas veces se puede necesitar que symbols con el mismo nombre sean la misma entidad. Para lograr esto, existe un global symbol registry. Ahí se puede crear symbols y accesarlos después, lo cual garantiza que cada vez que se acceda a la clave con el mismo nombre, esta devuelva exactamente el mismo symbol.

```
// Leer desde el registro global
// si el símbolo no existe, se crea
let id = Symbol.for("id");

// leer nuevamente
let idAgain = Symbol.for("id");

// Comparar los simbolos
// true
console.log( id === idAgain );
```

Símbolos

Existen varios symbols del sistema que JavaScript utiliza internamente, y que podemos usar para ajustar varios aspectos de nuestros objetos.

ECMAScript® 2024 Language Specification (tc39.es)

Símbolos

Un método que devuelve el iterador predeterminado para un objeto, el cual se puede personalizar para un comportamiento personalizado.

```
// Arreglo normal
const arrayDeStrings = ['uno', 'dos', 'tres'];

// Obtener el iterador del array
const iterador = arrayDeStrings[Symbol.iterator]();

// Iterar sobre el array usando el iterador
let resultado = iterador.next();
while (!resultado.done) {
  console.log(resultado.value);
  resultado = iterador.next();
}
```

Símbolos

Un método que devuelve el iterador predeterminado para un objeto, el cual se puede personalizar para un comportamiento personalizado.

```
// Definir una función generadora que crea objetos
// iterables con datos parametrizados
function crearIterable(datos) {
  return {
    datos: datos,
    indice: 0,
    [Symbol.iterator]() {
      return {
        next: () => {
          if (this.indice < this.datos.length) {
            return {
              value: this.datos[this.indice++],
              done: false
            };
          } else {
            return { done: true };
          }
        }
      };
    }
  };
}
```

```
// Utilizar el objeto iterable con
// diferentes conjuntos de datos
const iterable1 = crearIterable(
  ['uno', 'dos', 'tres']);

// Iterar sobre el primer iterable
for (const elemento of iterable1) {
  console.log(elemento);
}
```

8

Módulo 8

Generadores e iteradores

Iteradores

Este enfoque es conocido como **"bucle for tradicional"** y es útil cuando necesitas un control más preciso sobre el índice de la iteración.

```
var arreglo = [73, 6, 90, 19, 15];

for (var i = 0; i < arreglo.length; i++) {
    console.log(arreglo[i]);
}
```

Iteradores

La instrucción for...in itera sobre todas las propiedades enumerables de un objeto ignorando las propiedades codificadas por símbolos.

```
const object = { a: 1, b: 2, c: 3 };

for (const property in object) {
  console.log(`${property}: ${object[property]}`);
}
```

Iteradores

El bucle for .. of proporciona una forma más concisa de iterar sobre los elementos de una colección iterable, como un array.

¿Qué produce la siguiente salida?

```
const object = { a: 1, b: 2, c: 3 };

for (const property of object) {
  console.log(`${property}: ${object[property]}`);
}
```

Iteradores

El bucle for .. of proporciona una forma más concisa de iterar sobre los elementos de una colección iterable, como un array. ¿Por qué produce error el ejemplo anterior?

```
const object = [1, 3, 5, 7, 9];

for (const property of object) {
  console.log(property);
}
```

Iteradores

Para que un objeto sea considerado iterable en JavaScript, debe implementar el protocolo de iteración.

```
const objetoIterable = {
  clave1: 'valor1',
  clave2: 'valor2',
  clave3: 'valor3'
}
```

¿Un objeto en Javascript no es un objeto iterable?

Iteradores

El protocolo de iteración en JavaScript implica la presencia de un método especial llamado Symbol.iterator. Este método debe devolver un objeto iterador con un método next() que proporciona los valores sucesivos de la iteración.

```
// Arreglos
const array = [1, 2, 3];
// Cadenas
const cadena = 'Hola';
// Mapas
const mapa = new Map([
  ['clave1', 'valor1'],
  ['clave2', 'valor2'],
]);
// Conjuntos
const conjunto = new Set([1, 2, 3]);
```

Iteradores

```
const objetoIterable = {
  clave1: 'valor1',
  clave2: 'valor2',
  clave3: 'valor3',
  [Symbol.iterator]: function () {
    const claves = Object.keys(this);
    let index = 0;
    return {
      next: () => {
        if (index < claves.length) {
          const clave = claves[index++];
          return {
            value: [clave, this[clave]],
            done: false };
        } else {
          return { done: true };
        }
      },
    };
  },
};
```

En este ejemplo, el objeto objetoIterable implementa el método Symbol.iterator, que devuelve un iterador personalizado. Este iterador proporciona pares clave-valor durante la iteración.

Generators

Los generadores en JavaScript son una característica poderosa y flexible que permite la creación de funciones iterables. Los generadores se introdujeron con ECMAScript 6 (ES6) y están diseñados para simplificar la creación y gestión de secuencias de datos.

Generators

La principal diferencia entre una función normal y un generador es que las funciones generadoras pueden pausar y reanudar su ejecución en múltiples puntos, permitiendo una iteración paso a paso.

Generators

```
function* generadorEjemplo() {
  yield 1;
  yield 2;
  yield 3;
}

// Crear un generador
const miGenerador = generadorEjemplo();

// Iterar sobre los valores generados
// Imprime 1
console.log(miGenerador.next().value);
// Imprime 2
console.log(miGenerador.next().value);
// Imprime 3
console.log(miGenerador.next().value);
// Imprime undefined (ya que no hay más valores)
console.log(miGenerador.next().value);
```

La sintaxis de una función generadora utiliza la palabra clave function* en lugar de function. Dentro de la función generadora, puedes utilizar la palabra clave yield para pausar la ejecución y devolver un valor.

Ejercicio 1

Dado el siguiente árbol DOM

```
<div id="root">
    <p>Parrafo 1</p>
    <div>
        <p>Parrafo 2</p>
        <p>Parrafo 3</p>
    </div>
    <p>Parrafo 4</p>
</div>
```

1. ¿Cómo se puede recorrerlo con una función clásica?
2. ¿Cómo se puede recorrerlo con una función generadora?

9

Módulo 9

Programación Orientada
a Objetos

Programación Orientada a Objetos

El comportamiento del programa es llevado a cabo por objetos, entidades que representan elementos del problema a resolver y tienen atributos y comportamiento.

Clase

Modelo donde se establecen las características comunes de un grupo de objetos.

Instancia

Es un ejemplar o un objeto que se ha creado a partir de la clase o molde

OBJETO →

Atributos de Clase

Los atributos, también llamados datos o variables miembro son porciones de información que un objeto posee o conoce de sí mismo.

```
class Carro {
    color;
    modelo;
    linea;
}
```

Métodos de clase o instancia

Un método es una abstracción de una operación que puede hacer o realizarse con un objeto. Una clase puede declarar cualquier número de métodos que lleven a cabo operaciones de lo más variado con los objetos. Los métodos de instancia operan sobre las variables de instancia de los objetos pero también tienen acceso a las variables de clase.

```
class Carro {
    color;

    cambiarColor() {

    }
}
```

Métodos set y get

Los métodos get y set, son métodos que se asocian a un atributo de la clase y que se usan en las clases para mostrar (get) o modificar (set) el valor de un atributo. El nombre del método siempre será get o set y a continuación el nombre del atributo.

```
class Carro {
    // Atributo color
    _color;

    // Getter para obtener el color
    get color() {
        return this._color;
    }

    // Setter para establecer el color
    set color(nuevoColor) {
        this._color = nuevoColor;
    }
}
```

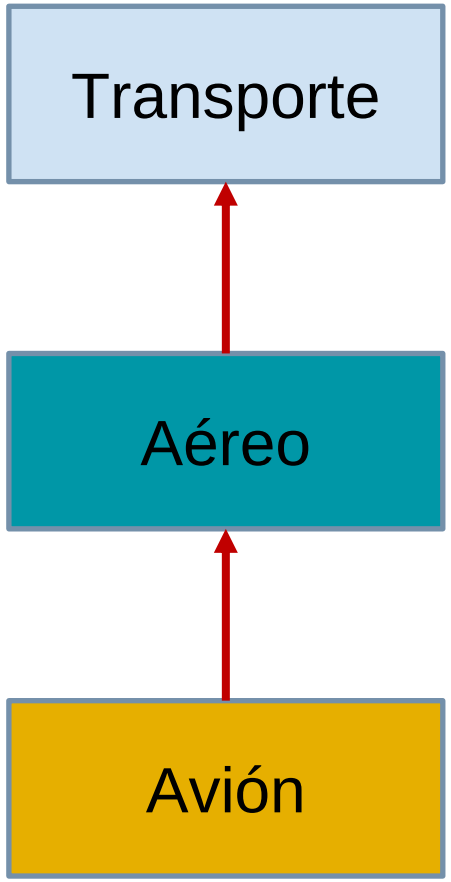
Obtener el estado de un objeto

En muchas ocasiones necesitamos obtener el estado actual de un objeto, para lo cual podemos crear un método personalizado o podemos sobre escribir el método toString()

```
class Carro {
    // Atributo color
    color;

    // Método toString para obtener la
    // representación en cadena del objeto
    toString() {
        return `Color: ${this.color}`;
    }
}
```

Herencia

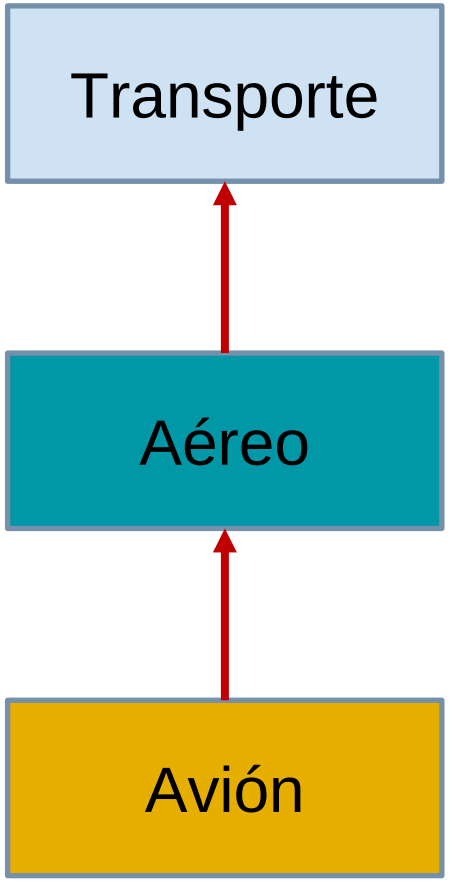
Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase.



Herencia

Superclase

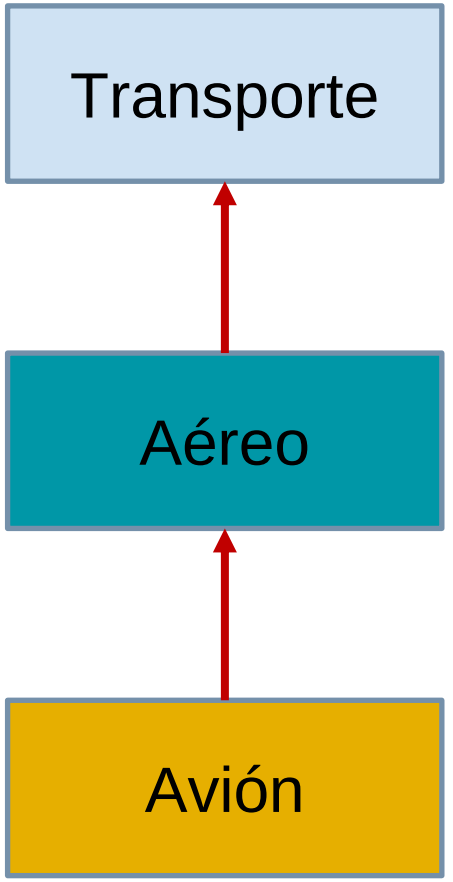
La clase cuyas características se heredan se conoce como superclase (o una clase base o una clase principal).



Herencia

Subclase

La clase que hereda la otra clase se conoce como subclase (o una clase derivada, clase extendida o clase hija). La subclase puede agregar sus propios campos y métodos además de los campos y métodos de la superclase.

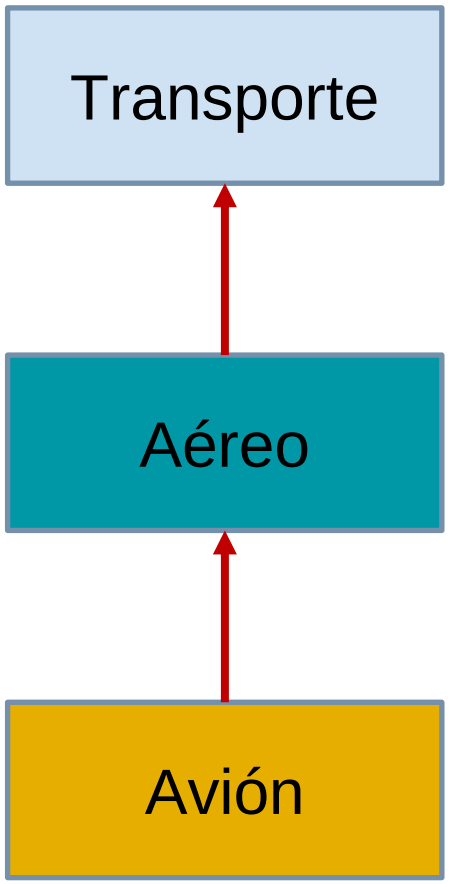


Herencia

```
class Transporte {

}
```

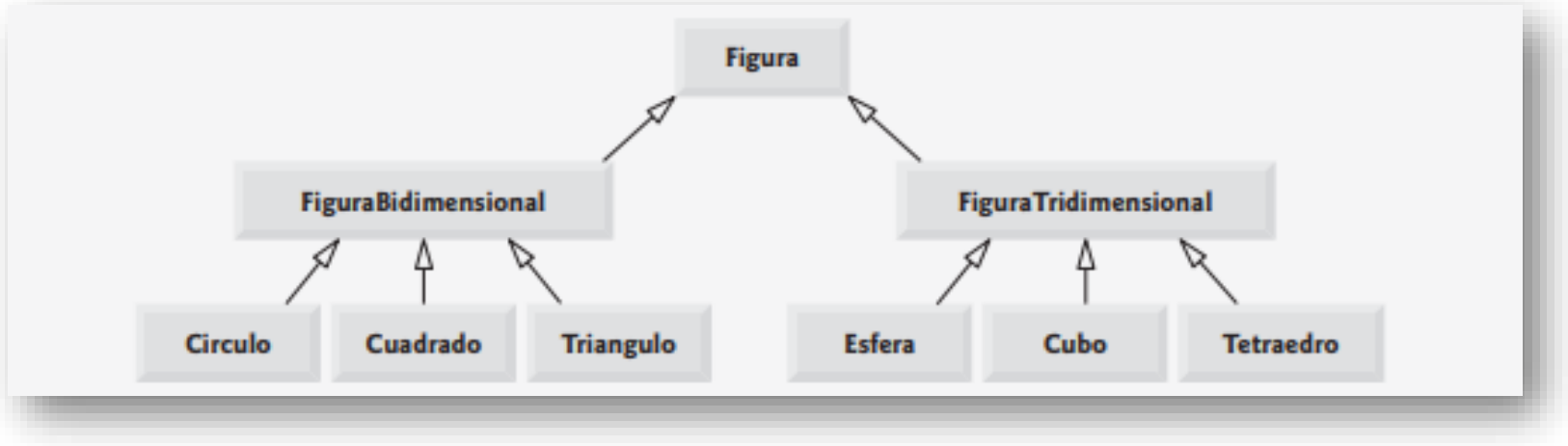
```
class Aereo extends Transporte {

}
```

```
class Avion extends Aereo {

}
```



Ejercicio 1

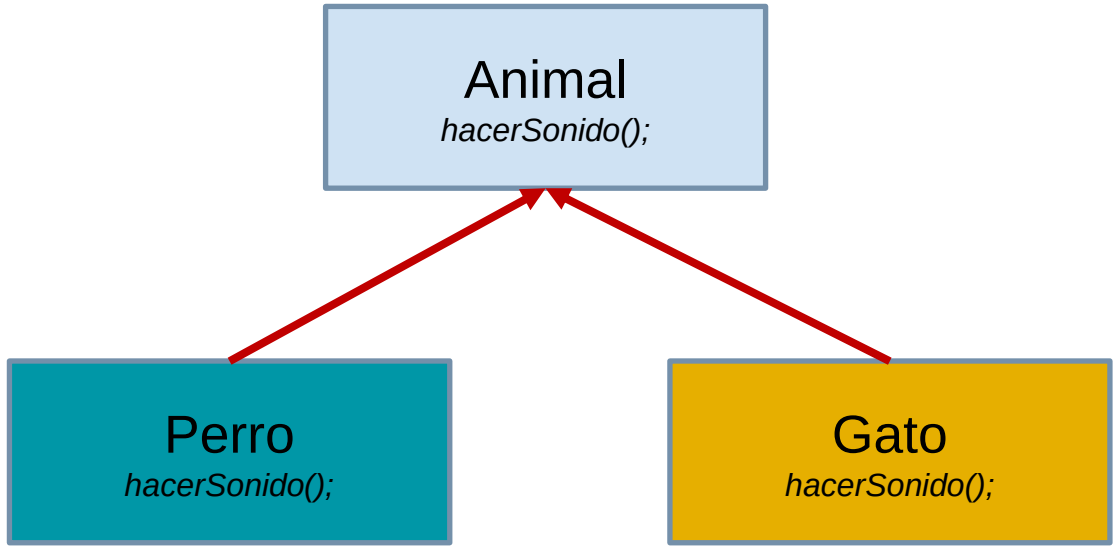
Implementar el siguiente diagrama de clases con Herencia



Polimorfismo

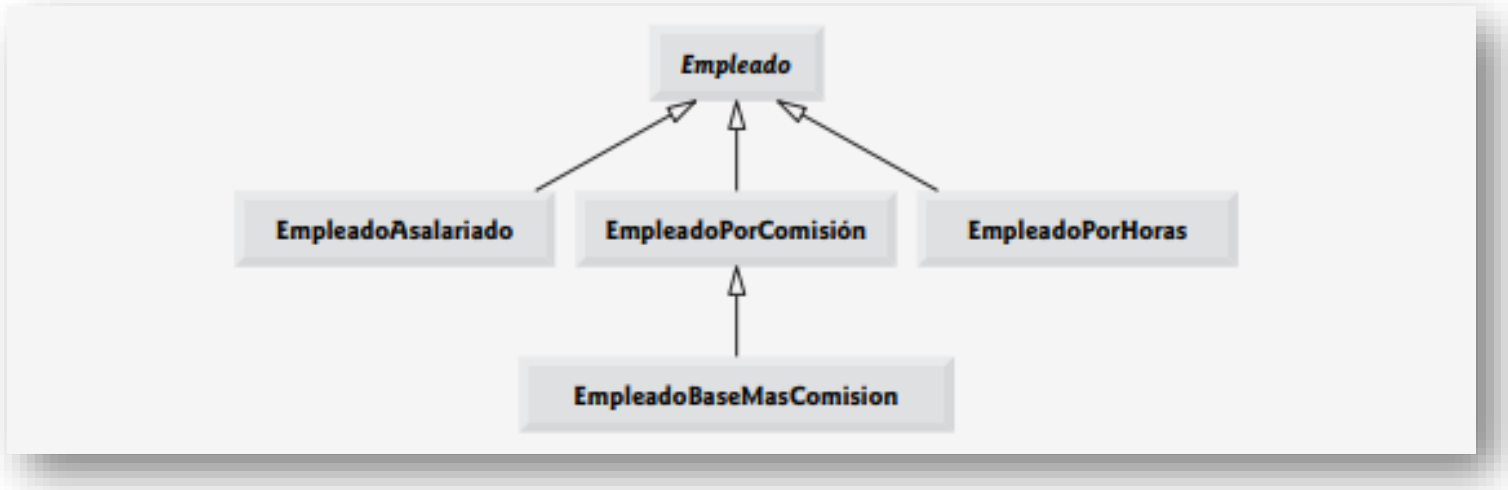
En programación orientada a objetos, polimorfismo es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros.

Es una característica de la programación orientada a objetos que permite llamar a métodos con igual nombre pero que pertenecen a clases distintas.



Ejercicio 1

Implementar el siguiente diagrama de clases



Cada clase debe contener:

1. Atributos
2. Métodos setter y getter
3. Constructor(es)

Ejercicio 1

Cada clase tiene las siguientes especificaciones:

	ingresos	toString
Empleado	abstract	*primerNombre apellidoPaterno* número de seguro social: *NSS*
Empleado-Asalariado	salarioSemanal	empleado asalariado: *primerNombre apellidoPaterno* número de seguro social: *NSS* salario semanal: *salarioSemanal*

EmpleadoPor-Horas	if horas <= 40
	  sueldo * horas
	else if horas > 40
	  40 * sueldo +
	  (horas - 40) *
	  sueldo * 1.5
		empleado por horas: *primerNombre apellidoPaterno*
		número de seguro social: *NSS*
		sueldo por horas: *sueldo*; horas trabajadas: *horas*

EmpleadoPor-Comisión	tarifaComisión * ventasBrutas	empleado por comisión: *primerNombre apellidoPaterno* número de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comisión: *tarifaComisión*
Empleado-BaseMas-Comision	(tarifaComision * ventasBrutas) + salarioBase	empleado por comisión con salario base: *primerNombre apellidoPaterno* número de seguro social: *NSS* ventas brutas: *ventasBrutas*; tarifa de comisión: *tarifaComision*; salario base: *salarioBase*

10

Módulo 10

Soicitudes de red

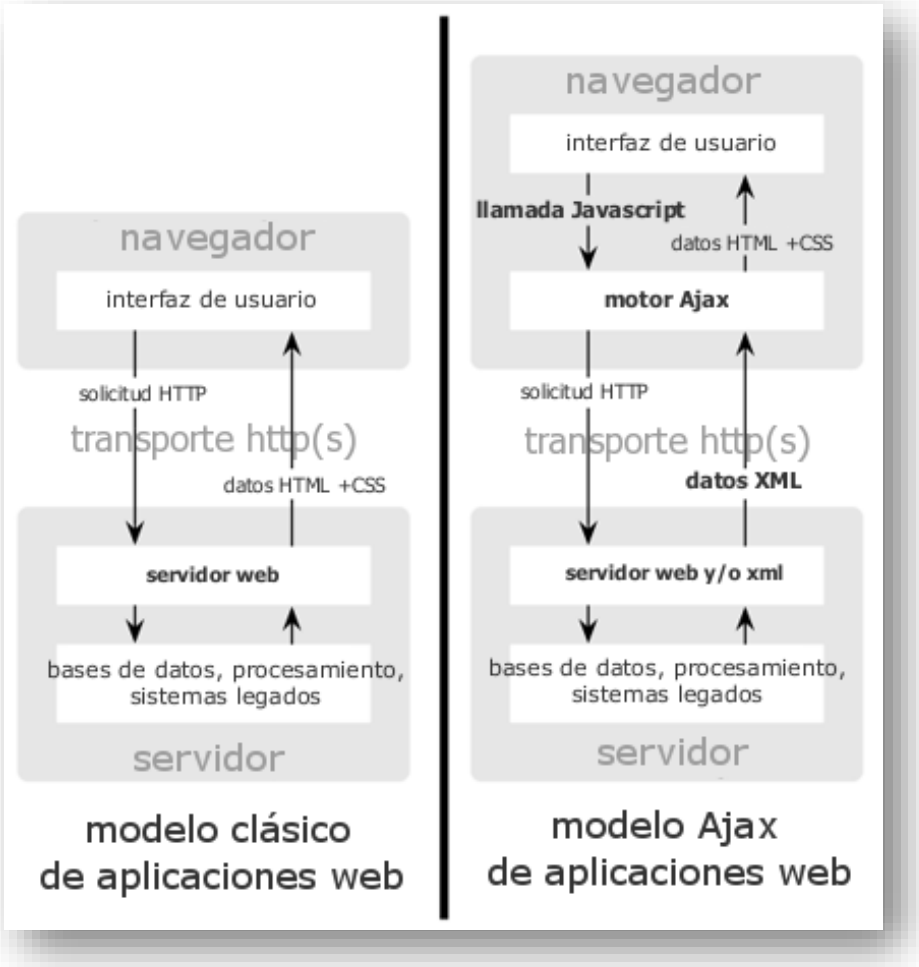
Destructuring

JavaScript puede enviar peticiones de red al servidor y cargar nueva información siempre que se necesite. Por ejemplo, se puede utilizar una petición de red para:

- Crear una orden.
- Cargar información de usuario.
- Recibir las últimas actualizaciones desde un servidor.

Solicitudes de red
AJAX

AJAX permite que una página web que ya ha sido cargada solicite nueva información al servidor.



Solicitudes de red

AJAX

AJAX permite que una página web que ya ha sido cargada solicite nueva información al servidor.

```
const xhr = new XMLHttpRequest();
const url = 'https://api.example.com/data';

xhr.open('GET', url, true);

xhr.onreadystatechange = function() {
  if (xhr.status === 200) {
    const response = JSON.parse(xhr.responseText);
    // Hacer algo con la respuesta recibida
  } else if (xhr.status !== 200) {
    // Manejar errores de la solicitud
  }
};

xhr.send();
```

Solicitudes de red
Promise

Una Promise (promesa) en JavaScript es un objeto que representa la eventual finalización (éxito o fracaso) de una operación asíncrona y la entrega de su resultado.

```
function fetchData() {
    return new Promise((resolve, reject) => {
      // Simulando una operación asincrónica
      setTimeout(() => {
        const data = 'Datos obtenidos de forma exitosa';
        if (data) {
            // La promesa se cumple
            resolve(data);
        } else {
            // La promesa se rechaza
            reject('Error al obtener los datos');
        }
      }, 2000); // Espera de 2 segundos
    });
}
```

Solicitudes de red
Promise

Estados de una promise:

1. Pendiente (pending): Estado inicial. Se crea una promesa y la operación está en progreso.

2. Cumplida (fulfilled): La promesa se resuelve correctamente y se obtiene el resultado esperado.

3. Rechazada (rejected): La promesa falla y se obtiene un motivo de error.

Solicitudes de red
Promise

Las promesas tienen dos partes principales: el productor (también conocido como "executor") y el consumidor. El productor es la función que realiza la operación asíncrona y eventualmente resuelve o rechaza la promesa.

```
// Crear una promesa
let miPromesa = new Promise((resolve, reject) => {
    // Simular una operación asíncrona
    setTimeout(() => {
        // Cambiar a false para simular un fallo
        let exito = true;

        if (exito) {
            resolve("¡Operación completada con éxito!");
        } else {
            reject("¡La operación ha fallado!");
        }

    }, 2000); // Simular un retardo de 2 segundos
});
```

El consumidor utiliza los métodos then y catch para manejar la resolución o el rechazo de la promesa.

```
// Consumir la promesa
miPromesa
  .then((resultado) => {
    // Se ejecuta si la promesa se resuelve
    console.log(resultado);
  })
  .catch((error) => {
    // Se ejecuta si la promesa es rechazada
    console.error(error);
  });
```

Solicitudes de red
Promise

Solicitudes de red
Promise

El método fetch() no es soportado por navegadores antiguos pero es perfectamente soportado por los navegadores actuales y modernos.

```
let promise = fetch(url, [options])
```

- **url:** representa la dirección URL a la que deseamos acceder.
- **options:** representa los parámetros opcionales, como puede ser un método o los encabezados de nuestra petición, etc.

Solicitudes de red

Promise

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    return response.json();
})
  .then(data => {
    // Hacer algo con los datos obtenidos
})
  .catch(error => {
    // Manejar el error de la promesa
});
```

Solicitudes de red
Comunicación asíncrona

La llamada telefónica es comunicación sincrónica, porque el mensaje se recibe al mismo tiempo que se manda.

Un mensaje de texto es comunicación asincrónica, pues es una acción que se ejecuta en un momento, pero solo finaliza cuando la otra parte lee el texto.

Asíncrona en javascript

Lo que hacemos en este lenguaje de programación es programar acciones que se ejecutarán en caso de que otra acción suceda. Sin embargo, no podemos controlar cuándo, ni quiera si sucederá la acción.

¿Para qué sirve?

La programación orientada a eventos es aquella que prepara la ejecución de código en función de los eventos que pueden ocurrir. Es decir, preparamos el código por si un evento sucede.

Async/Await

Async y await en JavaScript son dos palabras clave que nos permiten transformar un código asíncrono para que parezca ser síncrono.

```
async function fetchData() {
    try {
      const response = await fetch(
        'https://api.example.com/data');

      if (!response.ok) {
        throw new Error('Error en la solicitud');
      }
      const data = await response.json();
      // Hacer algo con los datos obtenidos
      return data;
    } catch (error) {
      // Manejar el error
      console.error(error);
      throw error;
    }
}
```

Async/Await

- La palabra clave async se utiliza en una función para envolver el contenido de la función en una promesa.
- La palabra clave await en JavaScript nos permite definir una sección de la función a la cual el resto del código debe esperar.

```
async function fetchData() {
    try {
      const response = await fetch(
        'https://api.example.com/data');

      if (!response.ok) {
        throw new Error('Error en la solicitud');
      }
      const data = await response.json();
      // Hacer algo con los datos obtenidos
      return data;
    } catch (error) {
      // Manejar el error
      console.error(error);
      throw error;
    }
}
```

Solicitudes de red
Comunicación asíncrona

Async/Await

Mandando a llamar la función fetchData con async/await

```
// Llamada a la función fetchData utilizando
//then y catch
fetchData()
.then(data => {
  // Manejar los datos obtenidos
})
.catch(error => {
  // Manejar el error
});
```