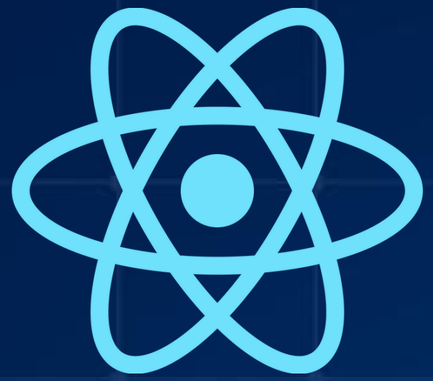
Secretaría de Desarrollo Económico Sustentable
Grandeza de México

Universidad Nacional Autónoma de México
Escuela Nacional de Estudios Superiores

IECAGto
Instituto Estatal de Capacitación



DESARROLLO WEB CON REACT

Temario

Fundamentos de desarrollo web

Javascript

Aplicaciones con React

Hooks y navegabilidad

Consumo de Web APIs

Herramientas en la nube

Sitios estáticos y SSR

Aplicaciones en tiempo real

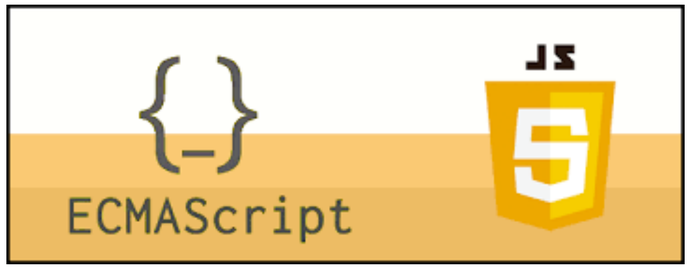
JavaScript

¿Qué es JavaScript?

JavaScript (JS) es un lenguaje de programación ligero, interpretado, o compilado justo-a-tiempo (just-in-time) con funciones de primera clase. Es más conocido como un lenguaje de scripting (secuencias de comandos) para páginas web, y es usado en muchos entornos fuera del navegador, tal como Node.js, Apache CouchDB y Adobe Acrobat JavaScript es un lenguaje de programación basada en prototipos, multiparadigma, de un solo hilo, dinámico, con soporte para programación orientada a objetos, imperativa y declarativa.

ECMA y ECMAScript

La *European Computer Manufacturers Association (ECMA)* es la asociación conformada por personas y fabricantes de tecnología encargada de llevar el rumbo de los cambios a JavaScript durante los años.



La primera versión de ECMAScript se publicó en 1997 y actualmente se han publicado 13 versiones, siendo la última ES2022 publicada en junio de 2022.

Los cambios a JavaScript son hechos por la comunidad, cualquier persona puede enviar una propuesta y el comité de ECMA decide y prioriza cuáles cambios se incluyen en cada especificación del lenguaje.

Declaración de variables

Anteriormente se utilizaba la palabra clave **var** para declarar cualquier tipo de variable.

Desde ES2015 se implementó el uso de las palabras reservadas **const** y **let** para declarar constantes (no pueden cambiar de valor una vez declaradas) y declarar variables dependientes del contexto.

```
const pizza = true;
pizza = false;
```

```
▶ Uncaught TypeError: Assignment to constant variable.
```

```
var topic = "JavaScript";

if (topic) {
  let topic = "React";
  console.log("block", topic); // React
}

console.log("global", topic); // JavaScript
```

Declaración de variables

Las variables declaradas con let pueden utilizarse en ciclos y mantener los valores de acuerdo al contexto de cada iteración.

```
const container = document.getElementById("container");
let div;
for (let i = 0; i < 5; i++) {
  div = document.createElement("div");
  div.onclick = function() {
    alert("This is box #: " + i);
  };
  container.appendChild(div);
}
```

Template strings

Es una nueva manera propuesta para la concatenación de cadenas de texto que permiten la inserción de variables dentro de estas evitando el uso excesivo de los la concatenación con el símbolo **«+»**. También permite la generación de cadenas de más de una línea.

```
console.log(lastName + ", " + firstName + " " + middleName);
```

```
console.log(`${lastName}, ${firstName} ${middleName}`);
```

```
const email = `
Hello ${firstName},

Thanks for ordering ${qty} tickets to ${event}.

Order Details
${firstName} ${middleName} ${lastName}
     ${qty} x $${price} = $${qty*price} to ${event}

You can pick your tickets up 30 minutes before
the show.

Thanks,

${ticketAgent}
`
```

Declaración de funciones

Las funciones permiten la ejecución de tareas repetitivas reutilizando el mismo código. Existen diferentes formas de declarar estas funciones, la primera de ellas es mediante **declaración**. Para declarar una función es necesario utilizar la palabra reservada ***function,*** seguida por el nombre de la función, y el cuerpo dentro de las llaves **{}**.

```
function logCompliment() {
  console.log("You're doing great!");
}

logCompliment();
```

Funciones como expresión

Otra opción para crear funciones es declararlas como **expresiones**, es decir, asignarlas a una variable de JavaScript.

Hay que tener en cuenta que las funciones **declaradas** pueden ser invocadas incluso si su declaración se coloca después, a diferencia de las funciones como expresión que arrojan un error.

```
const logCompliment = function() {
  console.log("You're doing great!");
};

logCompliment();
```

```
// Invoking the function before it's declared
hey();
// Function Declaration
function hey() {
  alert("hey!");
}

// Invoking the function before it's declared
hey();
// Function Expression
const hey = function() {
  alert("hey!");
};

TypeError: hey is not a function
```

Parámetros de función

Al igual que en otros lenguajes, las funciones pueden recibir parámetros nombrados (de cualquier tipo), así como regresar un valor. Los parámetros de función pueden tener valores por defecto para casos en los que no se agregue el atributo al llamar la función.

```
const createCompliment = function(firstName, message) {
  return `${firstName}: ${message}`;
};

createCompliment("Molly", "You're so cool");
```

```
function logActivity(name = "Shane McConkey", activity = "skiing") {
  console.log(`${name} loves ${activity}`);
}
```

Funciones arrow

Es una manera de crear funciones sin utilizar la palabra reservada **function**, e incluso se puede omitir la declaración de la palabra **return** si la función solo cuenta con una expresión. Cuando una función arrow solo acepta un parámetro también se pueden eliminar los paréntesis.

```
const lordify = function(firstName) {
  return `${firstName} of Canterbury`;
};
```

```
const lordify = firstName => `${firstName} of Canterbury`;
```

Funciones arrow

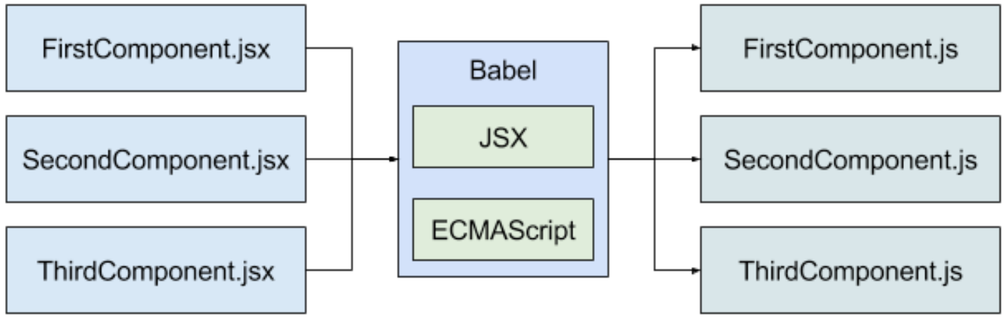
Las funciones arrow permiten proteger el **contexto** sobre el cual se están ejecutando, es decir, dependiendo de dónde sean declaradas pueden tener diferentes valores para **this.**

```
const tahoe = {
  mountains: ["Freel", "Rose", "Tallac", "Rubicon", "Silver"],
  print: function(delay = 1000) {
      setTimeout(() => {
        console.log(this.mountains.join(", "));
      }, delay);
    }
  };

  tahoe.print(); // Freel, Rose, Tallac, Rubicon, Silver
```

Compilación en JavaScript

Conforme se lanzan actualizaciones del lenguaje, es posible que no todos los navegadores lo soporten, por lo que se han creado herramientas para traducir estas características en código compatible a nivel general, este proceso es llamado **compilación**.

En la actualidad existen herramientas para la compilación de JavaScript como **Babel**.



Desestructuración de objetos

La **asignación desestructurada** permite generar variables a partir del contenido de un objeto y obtener solo los valores necesarios. Estas nuevas variables pueden ser declaradas con las palabras reservadas **const** y **let**.

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};

const { bread, meat } = sandwich;

console.log(bread, meat); // dutch crunch tuna
```

Desestructuración de objetos

La **asignación desestructurada** permite generar variables a partir del contenido de un objeto y obtener solo los valores necesarios. Estas nuevas variables pueden ser declaradas con las palabras reservadas **const** y **let**. Este proceso también puede ser aplicado a los atributos pasados a una función.

```
const sandwich = {
  bread: "dutch crunch",
  meat: "tuna",
  cheese: "swiss",
  toppings: ["lettuce", "tomato", "mustard"]
};

const { bread, meat } = sandwich;

console.log(bread, meat); // dutch crunch tuna
```

```
const lordify = ({ firstname }) => {
  console.log(`${firstname} of Canterbury`);
};

const regularPerson = {
  firstname: "Bill",
  lastname: "Wilson"
};

lordify(regularPerson); // Bill of Canterbury
```

Desestructuración de arreglos

Al igual que los objetos, los arreglos también pueden ser desestructurados, obteniendo los valores de acuerdo a la posición del arreglo en que se encuentran. Si deseamos obtener elementos de forma salteada se pueden colocar comas entre espacios vacíos.

```
const [firstAnimal] = ["Horse", "Mouse", "Cat"];

console.log(firstAnimal); // Horse
```

```
const [, , thirdAnimal] = ["Horse", "Mouse", "Cat"];

console.log(thirdAnimal); // Cat
```

Operador de propagación

Este operador (...) permite realizar diferentes tareas, tales como:

- Combinar contenidos de dos o más arreglos
- Crear copias de arreglos/objetos
- Obtener los elementos restantes de un arreglo.
- Obtener parámetros de una función como un arreglo.

```
const peaks = ["Tallac", "Ralston", "Rose"];
const canyons = ["Ward", "Blackwood"];
const tahoe = [...peaks, ...canyons];

console.log(tahoe.join(", ")); // Tallac, Ralston, Rose, Ward, Blackwood
```

```
const lakes = ["Donner", "Marlette", "Fallen Leaf", "Cascade"];

const [first, ...others] = lakes;

console.log(others.join(", ")); // Marlette, Fallen Leaf, Cascade
```

Fetch y Promesas

La función **fetch** permite realizar peticiones a APIs de forma simplificada en forma de una **promesa**. Las promesas son objetos que representan el estado de una operación asíncrona, y para completar su ejecución es necesario encadenar un método **.then()**.

La función de **then** es ejecutar código después de que la función asíncrona termina de forma exitosa, el valor regresado por dicha función es pasado como parámetro de la función en el cuerpo del then.

También existen las funciones **catch()** y **finally()**.

```
fetch("https://api.randomuser.me/?nat=US&results=1").then(res =>
  console.log(res.json())
);
```

Async/Await

Las **funciones asíncronas** son otro método utilizado en el manejo de promesas. Su principal característica es que evita el encadenamiento de funciones y se puede indicar de forma más explícita si se desea esperar a la finalización del código utilizando la palabra reservada **await**.

Para declarar una función asíncrona, se debe especificar la palabra **async** antes de la declaración de la función. Solo las funciones asíncronas permiten la ejecución de otras funciones con **await**.

```
const getFakePerson = async () => {
  let res = await fetch("https://api.randomuser.me/?nat=US&results=1");
  let { results } = res.json();
  console.log(results);
};

getFakePerson();
```

Construcción de promesas

Construir promesas permite definir múltiples escenarios de éxito y errores. Cuando se define una promesa se pasan los las funciones **resolves** y **rejects** como parámetros de la función interna para determinar el momento en que deseamos que la promesa devuelva una respuesta de éxito o error.

```
const getPeople = count =>
  new Promise((resolves, rejects) => {
    const api = `https://api.randomuser.me/?nat=US&results=${count}`;
    const request = new XMLHttpRequest();
    request.open("GET", api);
    request.onload = () =>
      request.status === 200
        ? resolves(JSON.parse(request.response).results)
        : reject(Error(request.statusText));
    request.onerror = err => rejects(err);
    request.send();
  });
```

Clases

Las clases proveen una sintaxis similar a la de lenguajes orientados a objetos tradicionales para el manejo de objetos en lugar de hacer uso de funciones como era hecho anteriormente.
Actualmente las clases no son utilizadas por React, sin embargo, se hacía uso de ellas en versiones antiguas de la biblioteca.

```
function Vacation(destination, length) {
  this.destination = destination;
  this.length = length;
}

Vacation.prototype.print = function() {
  console.log(this.destination + " | " + this.length + " days");
};

const maui = new Vacation("Maui", 7);

maui.print(); // Maui | 7 days
```

```
class Vacation {
  constructor(destination, length) {
    this.destination = destination;
    this.length = length;
  }

  print() {
    console.log(`${this.destination} will take ${this.length} days.`);
  }
}
```

Módulos de ES6

A partir de la especificación ES6, se incluyeron **«módulos»** en JavaScript, estos consisten en p**artes de código reutilizables** que pueden ser incorporados en otros archivos de JS sin causar colisión de variables.

Los elementos de JavaScript pueden ser exportados como **múltiples objetos dentro de un archivo** o como **una sola variable a nivel general.**

```
export const print=(message) => log(message, new Date())

export const log=(message, timestamp) =>
  console.log(`${timestamp.toString()}: ${message}`)
```

```
export default new Expedition("Mt. Freel", 2, ["water", "snack"]);
```

Inmutabilidad

La inmutabilidad es un concepto de programación funcional que propone que una estructura de datos nunca debe cambiar. El concepto de inmutabilidad en la programación se aplica cuando, para obtener el resultado a partir de una estructura, se genera una copia del objeto en lugar de modificarlo directamente. El siguiente ejemplo muestra una función que muta a un objeto y no cumple el principio de inmutabilidad.

```
let color_lawn = {
  title: "lawn",
  color: "#00FF00",
  rating: 0
};
```

```
function rateColor(color, rating) {
  color.rating = rating;
  return color;
}

console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 5
```

Inmutabilidad

Al pasar objetos como parámetros de una función, estamos compartiendo la referencia a la función misma, por lo que es importante entender que si modificamos algo de la estructura del parámetro lo estamos haciendo de manera global. Los siguientes ejemplos muestran dos propuestas para ejecutar la misma función sin mutar el objeto **color** y creando una copia en su lugar.

```
const rateColor = function(color, rating) {
  return Object.assign({}, color, { rating: rating });
};

console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 0
```

```
const rateColor = (color, rating) => ({
  ...color,
  rating
});
```

Funciones puras

Una **función pura** es una función que devuelve un valor basado en los parámetros recibidos, no causa efectos secundarios, no asigna variables globales ni cambia el estado de la aplicación. Trata a los argumentos como datos inmutables.

En el ejemplo de la izquierda podemos ver una función impura, y a la derecha una función pura:

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
};

function selfEducate() {
  frederick.canRead = true;
  frederick.canWrite = true;
  return frederick;
}

selfEducate();
console.log(frederick);
```

```
const frederick = {
  name: "Frederick Douglass",
  canRead: false,
  canWrite: false
};

const selfEducate = person => ({
  ...person,
  canRead: true,
  canWrite: true
});

console.log(selfEducate(frederick));
console.log(frederick);
```

Transformación de datos - filter()

JavaScript ya cuenta con funciones específicas para realizar transformación de datos siguiendo el principio de inmutabilidad en arreglos y objetos. La función filter produce un nuevo arreglo a partir de un predicado que devuelve los valores **true** o **false** para decidir si el elemento formará parte del nuevo arreglo.

```
const wSchools = schools.filter(school => school[0] === "W");

console.log(wSchools);
// ["Washington & Liberty", "Wakefield"]
```

Transformación de datos - map()

La función **map** permite generar un nuevo arreglo a partir de otro existente con el mismo número de elementos. Recibe como parámetro una función que se ejecuta una vez para cada elemento del arreglo, y el valor de retorno puede ser cualquier objeto que se desee.

```
const highSchools = schools.map(school => `${school} High School`);

console.log(highSchools.join("\n"));

// Yorktown High School
// Washington & Liberty High School
// Wakefield High School

console.log(schools.join("\n"));

// Yorktown
// Washington & Liberty
// Wakefield
```

Transformación de datos - reduce()

La función **reduce** permite obtener un valor a partir de la ejecución de una función por cada elemento del arreglo, guardando su valor anterior para la siguiente iteración.

```
const ages = [21, 18, 42, 40, 64, 63, 34];

const maxAge = ages.reduce((max, age) => {
  console.log(`${age} > ${max} = ${age > max}`);
  if (age > max) {
    return age;
  } else {
    return max;
  }
}, 0);

console.log("maxAge", maxAge);
```

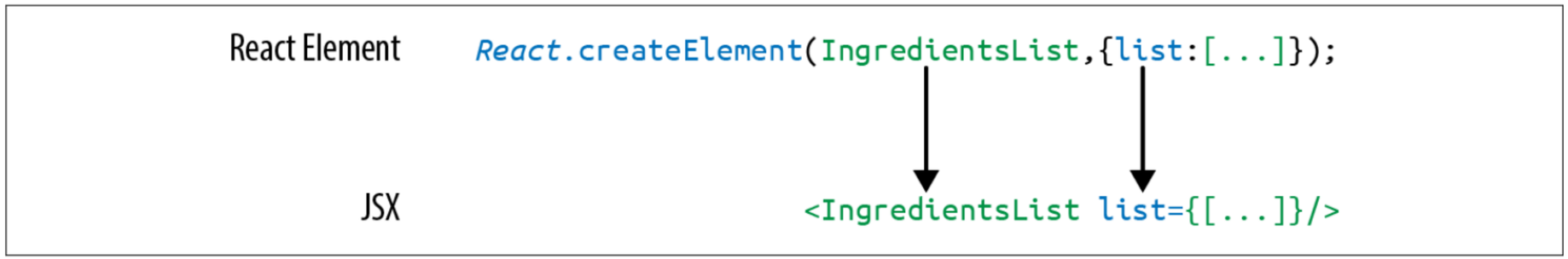
Funciones de Alto Orden

Las funciones de alto orden (HOC, *High Order Function*) son funciones que pueden manipular la ejecución de otras funciones, es decir, pueden recibir funciones como parámetros, e incluso devolver funciones como respuesta.

```
const invokeIf = (condition, fnTrue, fnFalse) =>
  condition ? fnTrue() : fnFalse();

const showWelcome = () => console.log("Welcome!!!");

const showUnauthorized = () => console.log("Unauthorized!!!");

invokeIf(true, showWelcome, showUnauthorized); // "Welcome!!!"
invokeIf(false, showWelcome, showUnauthorized); // "Unauthorized!!!"
```

JavaScipt XML (JSX)

JSX es una extensión de JavaScript que nos permite definir elementos de React usando una notación basada en etiquetas directamente en nuestro código **JavaScript**. Evita el uso excesivo del método **createElement**. Su parecido con HTML lo hace una herramienta útil a la hora de crear interfaces.



Características de JSX

Componentes anidados: Los componentes creados pueden añadirse como hijos de otros componentes y pueden ser añadidos más de una vez.

```
<IngredientsList>
  <Ingredient />
  <Ingredient />
  <Ingredient />
</IngredientsList>
```

Expresiones de JavaScript: Se pueden añadir expresiones de JavaScript incluyéndolas dentro de llaves **{}** para los atributos de cada elemento y en el cuerpo del componente.

```
<h1>{title}</h1>
```

```
<input type="checkbox" defaultChecked={false} />
```

Características de JSX

Uso de className: A diferencia de HTML, no se puede utilizar la etiqueta class debido a que es una palabra reservada de JS; en su lugar se usa **className**.

```
<IngredientsList>
  <Ingredient />
  <Ingredient />
  <Ingredient />
</IngredientsList>
```

Mapeo de arreglos: Los arreglos se pueden mapear en forma de elementos de **JSX** directamente.

```
<ul>
  {props.ingredients.map((ingredient, i) => (
    <li key="{i}">{ingredient}</li>
  ))}
</ul>
```

Integración con Babel

JSX no puede ser interpretado directamente por ningún navegador web en la actualidad, por lo que es necesario utilizar un **compilador de JS** para traducir el contenido JSX a algo que los navegadores puedan entender. La manera más sencilla de integrarlo es mediante el CDN de Babel.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Examples</title>
  </head>
  <body>
    <div id="root"></div>

    <!-- React Library & React DOM -->
    <script
  src="https://unpkg.com/react@16.8.6/umd/react.development.js">
  </script>
    <script
  src="https://unpkg.com/react-dom@16.8.6/umd/react-dom.development.js">
  </script>
    <script
  src="https://unpkg.com/@babel/standalone/babel.min.js">
  </script>

    <script type="text/babel">
      // JSX code here. Or link to separate JavaScript file that contains JSX.
    </script>
  </body>
</html>
```

Ejercicio de práctica - JSX

Usando el ejercicio de las recetas anterior. Modificarlo para evitar usar createElement y en su lugar utilizar JSX para el renderizado de los elementos.

Secretaría de Desarrollo Económico Sustentable

Grandeza de México

Escuela Nacional de Estudios Superiores

IECAGto

Instituto Estatal de Capacitación

¡GRACIAS!