



# JavaScript y React

## Clase 14

KOICA

IGU HANDONG GLOBAL  
UNIVERSITY



UNA

# OBJETIVOS DE LA CLASE 14

---

- Manejar Variables de estado de una componente mediante Hook (función useState)
- Manejar ES6 - Array
- Implementar los códigos de ejemplos propuestos en clase.



# ES6 - Array

---

- El objeto Array existe desde la primera versión de JavaScript, pero a lo largo del tiempo ha ido incorporando nuevas funcionalidades.
- Veremos los métodos que ha incorporado el objeto Array con la versión de ES6, recordando que todos los navegadores actuales los implementan en su totalidad.

# Método estático 'from'

---

El método estático 'from' retorna la referencia de un objeto de tipo Array que se crea a partir del dato que le pasamos:

```
<!DOCTYPE html>

<html>

<head>
  <title>Ejemplo de JavaScript</title>
  <meta charset="UTF-8">
</head>

<body>
  <script>
    const arreglo1 = [10, 20, 30];
    const arreglo2 = Array.from(arreglo1);
    arreglo1.fill(5);
    console.log(arreglo1); // [5,5,5]
    console.log(arreglo2); // [10,20,30]
  </script>
</body>

</html>
```

# Método estático 'from'

---

- Creamos un arreglo con 3 elementos:

```
const arreglo1 = [10, 20, 30];
```

- Luego llamamos al método 'from' del objeto 'Array', el cual nos retorna un nuevo arreglo con el mismo contenido de 'arreglo1'.

# Método estático 'from'

---

- Es importante recordar que los objetos en JavaScript (y los arreglos son objetos) almacenan la referencia del objeto y no los datos propiamente dichos como los hacen las variables de tipo primitiva. No es lo mismo hacer la asignación:

```
const arreglo2=arreglo1;
```

# Método estático 'from'

---

- Con la asignación previa sigue existiendo un solo objeto y luego dos referencias (arreglo1 y arreglo2) a dicho objeto.
- Si utilizamos la asignación tenemos como resultado:

```
<script>
  const arreglo1 = [10, 20, 30];
  const arreglo2 = arreglo1;
  arreglo1.fill(5);
  console.log(arreglo1); // [5,5,5]
  console.log(arreglo2); // [5,5,5]
</script>
```

# Método estático 'from'

---

- Como podemos comprobar si imprimimos el único arreglo ya sea con la referencia arreglo1 o arreglo2 luego se acceden al mismo contenido.
- Luego veremos que el método 'fill' cambia el contenido de un arreglo con el dato que le pasamos como parámetro.

```
<script>
  const arreglo1 = [10, 20, 30];
  const arreglo2 = arreglo1;
  arreglo1.fill(5);
  console.log(arreglo1); // [5,5,5]
  console.log(arreglo2); // [5,5,5]
</script>
```



# Método estático 'from'

---

- El método 'from' tiene opcionalmente un segundo parámetro al cual podemos pasar una función que reciba cada elemento del arreglo y retorne el dato a almacenar:

```
<!DOCTYPE html>
<html>

<head>
  <title>Ejemplo de JavaScript</title>
  <meta charset="UTF-8">
</head>

<body>
  <script>
    const arreglo1 = [10, 20, 30];
    const arreglo2 = Array.from(arreglo1,
    (elemento) => elemento * 2);
    console.log(arreglo2); // [20, 40, 60]
  </script>
</body>

</html>
```

# Método estático 'from'

---

- La función anónima retorna qué valor debe almacenar en el arreglo que se crea, en éste caso almacenamos los valores originarios del arreglo1 pero multiplicados por 2.
- Recordemos que las funciones flecha son ampliamente utilizadas a partir de ES6 ya que nos permiten generar un código muy conciso, utilizando la sintaxis anterior tenemos:

```
<script>
  const arreglo1 = [10, 20, 30];
  const arreglo2 = Array.from(arreglo1, function (elemento) { return elemento * 2 });
  console.log(arreglo2); // [20, 40, 60]
</script>
```

# Método estático 'of'

---

- El método estático 'of' retorna la referencia de un objeto de tipo Array a partir de una lista de valores que le pasamos a dicho método:

```
<script>

  const numeros = Array.of(1, 2, 3, 4);
  const texto = Array.of("hola", "mundo");
  const mezcla = Array.of(42, true, "JS", null);

  console.log(numeros); // [1, 2, 3, 4]
  console.log(texto);   // ["hola", "mundo"]
  console.log(mezcla);  // [42, true, "JS", null]

</script>
```

# Métodos 'keys', 'values' y 'entries'

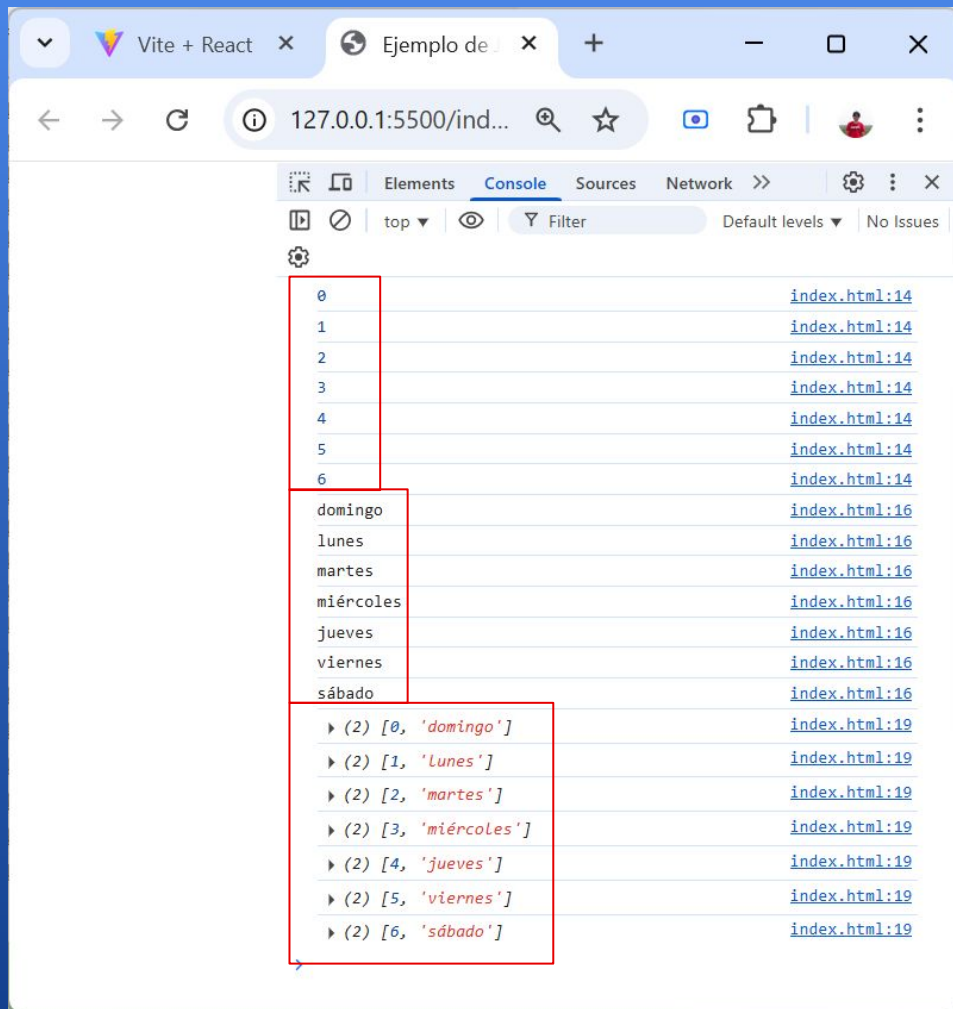
— — —

- El objeto Array implementa los métodos que permiten iterar sobre sus claves y valores:

```
<script>
    const lenguajes = ["domingo", "lunes", "martes",
        "miércoles",
        "jueves", "viernes", "sábado"];
    for (let indice of lenguajes.keys())
        console.log(indice); // 0 1 2 3 4 5 6
    for (let valor of lenguajes.values())
        console.log(valor); // "domingo" "lunes"
        "martes" "miércoles"
        // "jueves" "viernes" "sábado"
    for (let componente of lenguajes.entries())
        console.log(componente); // [0, "domingo"] [1,
        "lunes"] [2, "martes"]
                                   // [3, "miércoles"] [4,
        "jueves"]
                                   // [5, "viernes"] [6,
        "sábado"] [7, "domingo"]
</script>
```

# Métodos 'keys', 'values' y 'entries'

- Los datos que se recuperan en cada ciclo son:



# Métodos 'keys', 'values' y 'entries'

---

- En un solo 'for of' podemos recuperar cada componente y guardarlo en un Array el índice por un lado y el valor por otro:

```
for (let [indice,valor] of lenguajes.entries())  
  console.log(indice,valor);
```

# Método 'fill'

---

- El método 'fill' permite cambiar el contenido de todas o algunas componentes de un arreglo existente. En ningún caso permite redimensionarlo.
- Veamos un ejemplo y los efectos del método 'fill':

```
<script>
    const arreglo1 = [9, 9, 9, 9, 9, 9, 9];
    // fijamos el valor 3 a todas las componentes del arreglo
    arreglo1.fill(3);
    console.log(arreglo1); // [3, 3, 3, 3, 3, 3, 3]
    // fijamos el valor 0 desde la componente 3 del arreglo hasta el final
    arreglo1.fill(0, 3);
    console.log(arreglo1); // [3, 3, 3, 0, 0, 0, 0]
    // fijamos el valor 5 a todas las componentes
    arreglo1.fill(5);
    console.log(arreglo1); // [5, 5, 5, 5, 5, 5, 5]
    // fijamos el valor 100 desde la componente 3 hasta la componente 5 sin incluirla
    arreglo1.fill(100, 3, 5);
    console.log(arreglo1); // [5, 5, 5, 100, 100, 5, 5]
    // fijamos el valor 5 a todas las componentes
    arreglo1.fill(5);
    console.log(arreglo1); // [5, 5, 5, 5, 5, 5, 5]
    // fijamos el valor 100 en las dos últimas componentes
    arreglo1.fill(100, -2);
    console.log(arreglo1); // [5, 5, 5, 5, 5, 100, 100]
    // fijamos el valor 5 a todas las componentes
    arreglo1.fill(5);
    console.log(arreglo1); // [5, 5, 5, 5, 5, 5, 5]
    // fijamos el valor 100 en la antepenúltima y penúltima posición
    arreglo1.fill(100, -3, -1);
    console.log(arreglo1); // [5, 5, 5, 5, 100, 100, 5]
</script>
```



# Método 'fill'

---

- Si indicamos un solo valor se modifican todas las componentes del arreglo:

```
arreglo1.fill(3);
```

# Método 'fill'

— — —

- Podemos indicar desde qué posición se comienza a modificar:

```
// fijamos el valor 0 desde la componente 3 del arreglo  
// hasta el final  
arreglo1.fill(0, 3);
```

# Método 'fill'

---

- También podemos indicar desde y hasta qué componente se deben modificar:

```
// fijamos el valor 100 desde la componente 3 hasta  
// la componente 5 sin incluirla  
arreglo1.fill(100, 3, 5);
```

# Método 'fill'

— — —

- Finalmente podemos indicar con valores negativos comenzar desde el final del arreglo:

```
// fijamos el valor 100 en las dos últimas componentes
arreglo1.fill(100, -2);
console.log(arreglo1); // [5, 5, 5, 5, 5, 100, 100]

// fijamos el valor 100 en la antepenúltima y penúltima
// posición
arreglo1.fill(100, -3, -1);
console.log(arreglo1); // [5, 5, 5, 5, 100, 100, 5]
```

# Método 'copyWithin'

---

- El método 'copyWithin' permite copiar algunas componentes de un arreglo en otra parte del mismo arreglo. En ningún caso permite redimensionarlo.
- Veamos un ejemplo y los efectos del método 'copyWithin':

# Método 'copyWithin'

---

- El método 'copyWithin' es muy eficiente cuando debemos copiar un bloque de un arreglo a otra parte dentro del mismo arreglo.

```
<script>
  let arreglo1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];
  // Copiar a partir de la posición cero del arreglo
  // los elementos comprendidos entre las posiciones
  // 5 y 8
  arreglo1.copyWithin(0, 5, 8);
  console.log(arreglo1); // [6, 7, 8, 4, 5, 6, 7, 8, 9]
  arreglo1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];
  // Copiar a partir de la posición cero del arreglo
  // los elementos comprendidos desde la posición 5
  // hasta el final
  arreglo1.copyWithin(0, 5);
  console.log(arreglo1); // [6, 7, 8, 9, 5, 6, 7, 8, 9]
  arreglo1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];
  // Copiar a partir de la posición cero los dos últimos
  // elementos del arreglo
  arreglo1.copyWithin(0, -2);
  console.log(arreglo1); // [8, 9, 3, 4, 5, 6, 7, 8, 9]
</script>
```

# Búsquedas de valores e índices en un arreglo: indexOf, lastIndexOf e include

---

- Disponemos de una serie de métodos para recuperar el índice donde se encuentra un elemento en un arreglo. También mediante el método 'include' podemos identificar si un determinado valor está contenido en el Array.
- El siguiente ejemplo muestra distintas variantes de los métodos:

# Búsquedas de valores e índices en un arreglo: indexOf, lastIndexOf e include

---

```
<script>

  let arreglo1 = [10, 20, 30, 40, 50, 60, 10, 20, 30];
  // Indice donde se almacena el valor 30
  console.log(arreglo1.indexOf(30)); // 2
  // Indice donde se almacena un valor inexistente
  console.log(arreglo1.indexOf(100)); // -1
  // Indice donde se almacena el valor 20
  // buscando desde el final
  console.log(arreglo1.lastIndexOf(20)); // 7
  // Indice donde se almacena el valor 10
  // comenzando la búsqueda en la posición 5
  console.log(arreglo1.indexOf(10, 5)); // 6
  // Existe el valor 50 en el arreglo?
  console.log(arreglo1.includes(50)); // true
  // Existe el valor 100 en el arreglo?
  console.log(arreglo1.includes(100)); // false

</script>
```



# Búsquedas de valores e índices en un arreglo: indexOf, lastIndexOf e include

— — —

- Al método includes también podemos pasar un segundo parámetro que indica a partir de qué posición dentro del arreglo comenzar a buscar:

```
// Existe el valor 100 en el arreglo a partir de  
// la posición 2?  
console.log(arreglo1.includes(100, 2));
```

# Métodos 'find' y 'findIndex'

---

- Estos dos métodos nos permiten enviar una función anónima donde definimos el algoritmo de la búsqueda:

```
<script>
    const personas = [
        {
            nombre: "oscar",
            edad: 72
        },
        {
            nombre: "maría",
            edad: 22
        }
    ];
    let per = personas.find((persona, indice) => persona.nombre == "maría");
    if (per != undefined)
        console.log(per);
    else
        console.log("No se encuentra maría en el vector de objetos");
    let indice = personas.findIndex((persona, indice) => persona.nombre == "maría");
    if (indice != -1)
        console.log("Se encuentra en la posición " + indice + " del arreglo.");
    else
        console.log("No se encuentra maria en el vector de objetos");
</script>
```

# Métodos 'find' y 'findIndex'

---

- El método find retorna el valor 'undefined' si no retorna true el algoritmo dispuesto en la función anónima. En el caso que se encuentra el valor buscado luego retorna el valor de la posición donde se lo ubicó.
- Finalmente el método findIndex si la función anónima retorna true luego el método retorna la posición donde se encontró el valor a buscar.

# Métodos 'find' y 'findIndex'

— — —

- Estamos utilizando funciones flecha, pero nada nos impide de utilizar la sintaxis de funciones anónimas tradicionales:

```
let per = personas.find(function (persona, indice) { return persona.nombre == "maría" });
if (per != undefined)
    console.log(per);
else
    console.log("No se encuentra maría en el vector de objetos");
let indice = personas.findIndex(function (persona, indice) { return persona.nombre == "maría" });
if (indice != -1)
    console.log("Se encuentra en la posición " + indice + " del arreglo.");
else
    console.log("No se encuentra maría en el vector de objetos");
```

# Métodos every y some

---

- Al método 'every' debemos pasar una función anónima que procesa cada elemento del arreglo. Si todos los elementos cumplen la condición impuesta dentro de la función, luego el método 'every' retorna true.
- En cambio con el método 'some' con que un elemento cumpla la condición impuesta luego el método retorna true.
- every (todos) some (alguno)

```
<script>

  const personas = [
    {
      nombre: "carlos",
      edad: 12
    },
    {
      nombre: "maría",
      edad: 22
    }
  ];

  // Todas las personas tienen una edad menor a 100?
  console.log(personas.every((elemento, indice, arreglo) => elemento.edad < 100)); // true
  // Todas las personas son mayor de edad?
  console.log(personas.every((elemento, indice, arreglo) => elemento.edad >= 18)); // false
  // Todas las personas tienen un nombre con 6 o menos caracteres?
  console.log(personas.every((elemento, indice, arreglo) => elemento.nombre.length <= 6)); // true
  // Alguna persona tiene 100 años de edad?
  console.log(personas.some((elemento, indice, arreglo) => elemento.edad == 100)); // false
  // Alguna persona es menor de edad?
  console.log(personas.some((elemento, indice, arreglo) => elemento.edad <= 18)); // true
</script>
```

# Método filter

---

- El método 'filter' retorna un arreglo con todos los elementos del arreglo original que cumplen una condición que definimos en la función anónima que le pasamos:



```
<script>
  const personas = [
    {
      nombre: "pedro",
      edad: 34
    },
    {
      nombre: "ana",
      edad: 54
    },
    {
      nombre: "carlos",
      edad: 12
    }
  ];
  // Recuperar todas las personas mayores de edad
  const personasmayores = personas.filter((elemento, indice, arreglo) => elemento.edad >= 18);
  console.log(personasmayores); // [{nombre: "pedro",edad: 34},{nombre: "ana",edad: 54}]
</script>
```

# Método filter

— — —

- Tener en cuenta que la función anónima debe retornar true si queremos que el elemento se almacene en el arreglo generado. Si bien no tiene sentido, si queremos que se guarden todos los elementos luego podemos codificar:

```
const personasmayores = personas.filter((elemento, indice, arreglo) => true);
```

- Recordar que si no utilizamos funciones flecha el algoritmo será:

```
const personasmayores = personas.filter(function (elemento, indice, arreglo) { return true });
```

# Método map

---

- El método 'map' similar al método 'filter' retorna un arreglo cuyos elementos resultan de las operaciones que se efectúan dentro de la función anónima que le pasamos como parámetro:

```
<script>
  const arreglo1 = [1, 2, 3, 4, 5];
  // Generar un nuevo arreglo con los elementos del arreglo1 elevados al cuadrado
  const arreglo2 = arreglo1.map((elemento, indice, arreglo) => elemento ** 2);
  console.log(arreglo2); //[1, 4, 9, 16, 25]
</script>
```

- El arreglo generado siempre es del mismo tamaño que el arreglo original.

# Métodos reduce y reduceRight

---

- El método 'reduce' recibe como primer parámetro una función cuyo primer parámetro es un valor que se arrastra entre cada proceso de un elemento del arreglo, el segundo parámetro de reduce es el valor inicial con el que inicia el acumulador (si no pasamos el segundo parámetro se inicia con el primer elemento del arreglo):

```
<script>

const arreglo1 = [1, 2, 73, 3, 4, 120, 5, 18];
// Acumular todos los elementos que tienen un solo dígito
const cantidad = arreglo1.reduce((acumulador, elemento, indice, arreglo) => {
  if (elemento < 10)
    acumulador += elemento;
  return acumulador;
}, 0);
console.log(cantidad); // 15

</script>
```

# Métodos reduce y reduceRight

- El método 'reduceRight' es similar a 'reduce' con la diferencia que visita cada elemento del arreglo partiendo del final.

```
<script>
  const arreglo1 = [1, 2, 73, 3, 4, 120, 5, 18];
  // Generar un nuevo arreglo con los elementos invertidos
  const arreglo2 = arreglo1.reduceRight((acumulador, elemento, indice, arreglo) => {
    acumulador.push(elemento);
    return acumulador;
  }, []);
  console.log(arreglo2); // [18, 5, 120, 4, 3, 73, 2, 1]
</script>
```

# Métodos reduce y reduceRight

- El valor inicial es un arreglo vacío que le pasamos al método reduceRight:

```
[]
```

- Y dentro de la función anónima agregamos cada elemento del vector visitando sus componentes a partir del final:

```
acumulador.push(elemento);
```

# Problema

---

- Modificar el problema anterior (proyecto 004) para que se muestren 5 valores aleatorios. Almacenar los 5 valores en un vector y éste en la variable de estado.

```
import { useState } from "react";

function App() {

  function generarAleatorios() {

    const vec = new Array(5)

    for (let x = 0; x < vec.length; x++)

      vec[x] = Math.trunc(Math.random() * 10)

    setNumeros(vec)

  }

  const [numeros, setNumeros] = useState([0, 0, 0, 0, 0]);

  return (

    <div>

      <p>Números aleatorios:</p>

      {numeros.map(num => (<p>{num}</p>))}

      <button onClick={generarAleatorios}>Generar números aleatorios</button>

    </div>

  );

}

export default App;
```

# Variables de estado de una componente mediante Hook (función useState)

— — —

- A la función useState podemos pasar tanto tipos de datos primitivos como vectores:

```
const [numeros, setNumeros] = useState([0,0,0,0,0]);
```



# Variables de estado de una componente mediante Hook (función useState)

---

- En la función generarAleatorios() procedemos a crear un vector de 5 elementos y guardar los 5 valores aleatorios. Seguidamente llamamos a la función 'setNumeros':

```
function generarAleatorios() {  
  
    const vec=new Array(5)  
  
    for(let x=0; x<vec.length; x++)  
  
        vec[x]=Math.trunc(Math.random()*10)  
  
    setNumeros(vec)  
  
}
```

# Variables de estado de una componente mediante Hook (función useState)

---

Para mostrar todos los elementos del vector llamamos al método map de la clase Array y le pasamos una función flecha que retorna para cada elemento del vector un valor encerrado entre las marcas 'p':

```
{numeros.map(num => (<p>{num}</p>)) }
```

# Problema

— — —

- Crear un nuevo proyecto llamado proyecto005.
- Almacenar en el estado de la componente el siguiente vector con artículos:

```
[{  
  codigo: 1,  
  descripcion: 'papas',  
  precio: 12.52  
},{  
  codigo: 2,  
  descripcion: 'naranjas',  
  precio: 21  
},{  
  codigo: 3,  
  descripcion: 'peras',  
  precio: 18.20  
}]
```

```
import { useState } from "react";

function App() {
  function eliminarUltimaFila() {
    if (articulos.length > 0) {
      const temp=Array.from(articulos)
      temp.pop()
      setArticulos(temp)
    }
  }

  const [articulos, setArticulos] = useState([
    {
      codigo: 1,
      descripcion: 'papas',
      precio: 12.52
    }, {
      codigo: 2,
      descripcion: 'naranjas',
      precio: 21
    }, {
      codigo: 3,
      descripcion: 'peras',
      precio: 18.20
    }
  ]);
}
```

```
return (  
  <div>  
    <table border="1">  
      <thead><tr><th>Código</th><th>Descripción</th><th>Precio</th></tr></thead>  
      <tbody>  
        {articulos.map(art => {  
          return (  
            <tr key={art.codigo}>  
              <td>  
                {art.codigo}  
              </td>  
              <td>  
                {art.descripcion}  
              </td>  
              <td>  
                {art.precio}  
              </td>  
            </tr>  
          )  
        })}  
      </tbody>  
    </table>  
  </div>  
)
```

```
</tbody>
  </table>
  <button onClick={eliminarUltimaFila}>Eliminar última fila</button>
</div>
);}
export default App;
```

# Variables de estado de una componente mediante Hook (función useState)

---

Llamamos a la función useState para crear la variable de estado y le pasamos el arreglo de objetos

```
const [articulos, setArticulos] = useState([{\n  codigo: 1,\n  descripcion: 'papas',\n  precio: 12.52\n}, {\n  codigo: 2,\n  descripcion: 'naranjas',\n  precio: 21\n}, {\n  codigo: 3,\n  descripcion: 'peras',\n  precio: 18.20\n}]);
```

# Variables de estado de una componente mediante Hook (función useState)

Para mostrar los datos del vector nuevamente empleamos la llamada al método 'map' y le pasamos una función anónima. Dentro de la función anónima generamos cada fila de la tabla con los datos de un producto:

```
<table border="1">
  <thead><tr><th> Código</th><th> Descripción</th><th> Precio</th>
  </tr></thead>
  <tbody>
    {articulos.map(art => {
      return (
        <tr key={art.codigo}>
          <td>
            {art.codigo}
          </td>
          <td>
            {art.descripcion}
          </td>
          <td>
            {art.precio}
          </td>
        </tr>
      )
    }) }
  </tbody>
</table>
```



# Variables de estado de una componente mediante Hook (función useState)

---

- Como podemos comprobar la fila de títulos de la tabla la hacemos previo a llamar a map.
- Cuando se presiona el botón llamamos a la función 'eliminarUltimaFila'. Primero comprobamos si el vector tiene elementos accediendo al atributo 'length', en caso afirmativo procedemos a crear una copia del vector original llamando al método from y almacenando en una variable temporal el nuevo vector, luego eliminamos el último elemento llamando al método pop y finalmente actualizamos la variable de estado llamando a la función 'setArticulos' (hay que tener en cuenta que debemos crear siempre un nuevo vector con la copia del original y pasar dicho valor):

# Variables de estado de una componente mediante Hook (función useState)

---

```
function eliminarUltimaFila() {  
  if (articulos.length > 0) {  
    const temp=Array.from(articulos)  
    temp.pop()  
    setArticulos(temp)  
  }  
}
```

# Problema

---

- Modificar el ejercicio anterior para que aparezca un botón en cada fila de la tabla y permita borrar dicho artículo.

```
[{  
  codigo: 1,  
  descripcion: 'papas',  
  precio: 12.52  
},{  
  codigo: 2,  
  descripcion: 'naranjas',  
  precio: 21  
},{  
  codigo: 3,  
  descripcion: 'peras',  
  precio: 18.20  
}]
```

```
import { useState } from "react";

function App() {

  function borrar(cod) {
    const temp = articulos.filter((art)=>art.codigo !== cod);
    setArticulos(temp)
  }

  const [articulos, setArticulos] = useState([
    {
      codigo: 1,
      descripcion: 'papas',
      precio: 12.52
    }, {
      codigo: 2,
      descripcion: 'naranjas',
      precio: 21
    }, {
      codigo: 3,
      descripcion: 'peras',
      precio: 18.20
    }
  ]);
```

```
return (
  <div>
    <table border="1">
      <thead><tr><th>Código</th><th>Descripción</th><th>Precio</th><th>Borra?</th></tr></thead>
      <tbody>
        {articulos.map(art => {
          return (
            <tr key={art.codigo}>
              <td>
                {art.codigo}
              </td>
              <td>
                {art.descripcion}
              </td>
              <td>
                {art.precio}
              </td>
              <td>
                <button onClick={() => borrar(art.codigo)}>Borrar</button>
              </td>
            </tr>
          )
        }) }
```

```
        </tbody>
      </table>
    </div>
  );
}

export default App;
```

# Variables de estado de una componente mediante Hook (función useState)

---

Cuando generamos ahora la tabla en la última celda debemos disponer un botón que llame a una función y le pase como parámetro el código de artículo a borrar:

```
<button onClick={() => borrar(art.codigo)}>Borrar</button>
```

# Variables de estado de una componente mediante Hook (función useState)

---

Cuando llamamos a una función debemos plantear una función anónima que se le asigna al evento 'onClick'. La función 'borrar' recibe como parámetro el código de artículo a borrar:

```
function borrar(cod) {  
    const temp = articulos.filter((art)=>art.codigo !== cod);  
    setArticulos(temp)  
}
```



# Variables de estado de una componente mediante Hook (función useState)

---

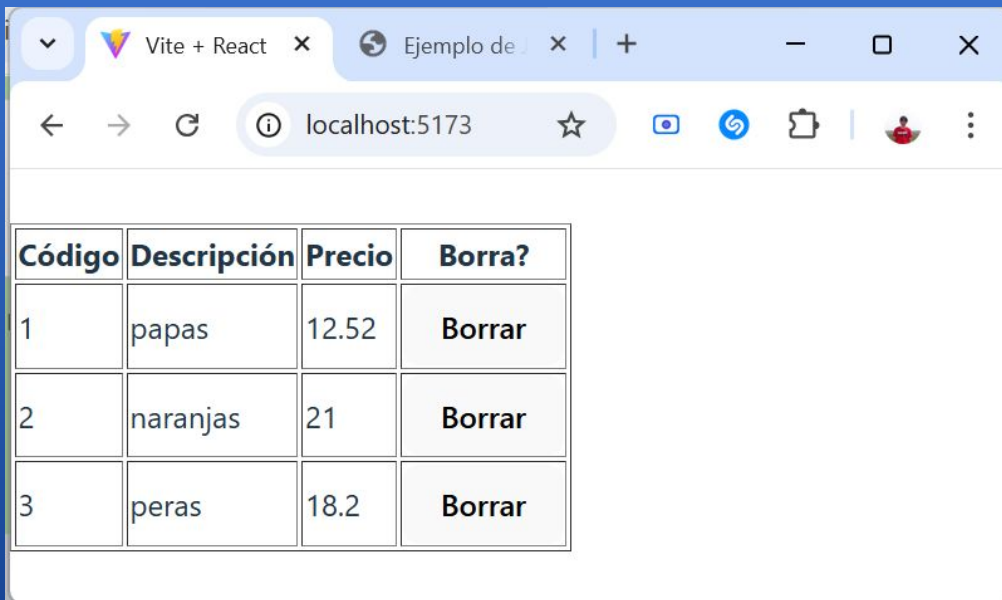
Para borrar un determinado elemento del vector utilizamos el método `filter` que genera otro vector con todas las componentes que cumplen la condición que le pasamos en la función anónima.

Finalmente actualizamos el estado para que se redibuje la página.

Tenemos como resultado:

# Variables de estado de una componente mediante Hook (función useState)

---



The screenshot shows a web browser window with two tabs: 'Vite + React' and 'Ejemplo de...'. The address bar shows 'localhost:5173'. The main content area displays a table with the following data:

Código	Descripción	Precio	Borra?
1	papas	12.52	Borrar
2	naranjas	21	Borrar
3	peras	18.2	Borrar

# Acotaciones

---

Las variables de estado pueden almacenar datos de tipo:

- String.
- Boolean
- Number
- Array
- Object
- Undefined

Los valores iniciales del estado se ejecutan solo una vez y es cuando se carga la componente.

# Componentes

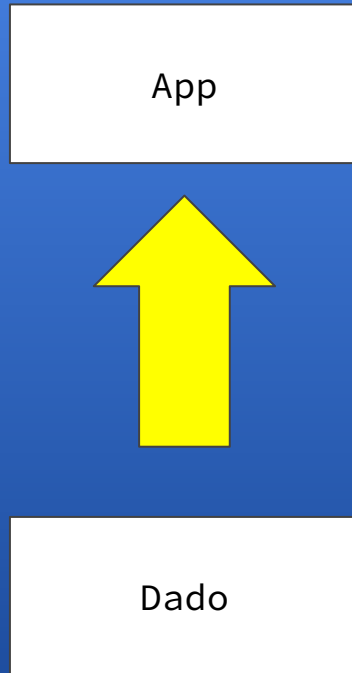
---

- Las componentes son una característica fundamental de React. Nos permiten dividir la aplicación en trozos de nuestra interfaz visual con objetivos bien definidos: 'menú de opciones', 'formulario de búsqueda', 'ventanas de mensajes', 'tablas de datos' etc.
- La división de nuestra aplicación en componentes nos ayudan a reutilizar dicho código en proyectos futuros.
- Hasta ahora cada proyecto que hemos desarrollado ha implementado una sola componente funcional llamada 'App' (hacemos referencia a que se trata de una componente funcional, ya que versiones antiguas de React proponían por defecto una componente de clase)

# Problema

— — —

- Crear un nuevo proyecto llamado proyecto006
- Confeccionar una aplicación que muestre tres dados.
- Plantear una componente funcional llamada 'Dado' que tenga una propiedad llamada 'valor' a la cual le llega el dato a mostrar.
- Nuestra componente principal seguirá siendo 'App' y dentro de la misma definiremos 3 componentes de tipo 'Dado':



# Componentes

---

- Veamos todos los pasos que debemos dar para poder implementar este proyecto empleando 2 componentes:
- Primero creamos en la carpeta src dos archivos llamados 'Dado.jsx' y 'Dado.css'

```

  v proyecto006
    > node_modules
    > public
    v src
      > assets
      # App.css
      ⚛ App.jsx
      # Dado.css
      ⚛ Dado.jsx
      # index.css
      ⚛ main.jsx
      .gitignore
      eslint.config.js
      <> index.html
      {} package-lock.json
      {} package.json

```

A screenshot of a file explorer window showing the project structure. The 'proyecto006' folder is expanded, showing 'node\_modules', 'public', and 'src'. The 'src' folder is further expanded, showing 'assets', 'App.css', 'App.jsx', 'Dado.css', 'Dado.jsx', 'index.css', 'main.jsx', '.gitignore', 'eslint.config.js', 'index.html', 'package-lock.json', and 'package.json'. The 'Dado.jsx' file is highlighted with a blue selection bar.

# Componentes

— — —

- Codificamos la componente 'Dado' en el archivo 'Dado.jsx':

```
import './Dado.css'

function Dado(props) {
  return (
    <div className="dado-recuadro">{props.valor}</div>
  );
}

export default Dado
```

# Componentes

---

- Mostramos el dato almacenado en la propiedad 'valor' que llegará desde la componente 'App' mediante el parámetro props:

```
<div className="dato-recuadro">{props.valor}</div>
```



# Componentes

---

- Utilizamos estilos para que el dado aparezca mejor en pantalla. Los estilos los guardamos en el archivo 'Dado.css' y lo debemos importar para utilizarlo en el div:

```
import './Dado.css'
```

# Componentes

---

- Como importaremos la clase 'Dado' luego en la clase 'App' debemos exportarla:

```
export default Dado
```

# Componentes

---

- También codificamos el archivo 'Dado.css':

```
.dado-recuadro {  
  display: flex;  
  width: 2rem;  
  height: 2rem;  
  background-color: black;  
  color: white;  
  justify-content: center;  
  align-items: center;  
  margin: 1rem;  
  border-radius: 4px;  
}
```

# Componentes

— — —

- Ahora codificamos la función 'App':

```
import Dado from "../Dado";

function App() {
  const valor1 = Math.trunc(Math.random()*6)+1
  const valor2 = Math.trunc(Math.random()*6)+1
  const valor3 = Math.trunc(Math.random()*6)+1
  return (
    <div>
      <Dado valor={valor1} />
      <Dado valor={valor2} />
      <Dado valor={valor3} />
    </div>
  );
}

export default App;
```

# Componentes

---

- Lo primero que debemos hacer para poder utilizar la componente Dado es importarla:

```
import Dado from './Dado';
```

# Componentes

---

- Lo nuevo es que en el bloque JSX definimos tres etiquetas Dado y pasamos la propiedad valor con distintos valores aleatorios previamente generados:

```
const valor1 = Math.trunc(Math.random()*6)+1

const valor2 = Math.trunc(Math.random()*6)+1

const valor3 = Math.trunc(Math.random()*6)+1

return (

  <div>

    <Dado valor={valor1} />

    <Dado valor={valor2} />

    <Dado valor={valor3} />

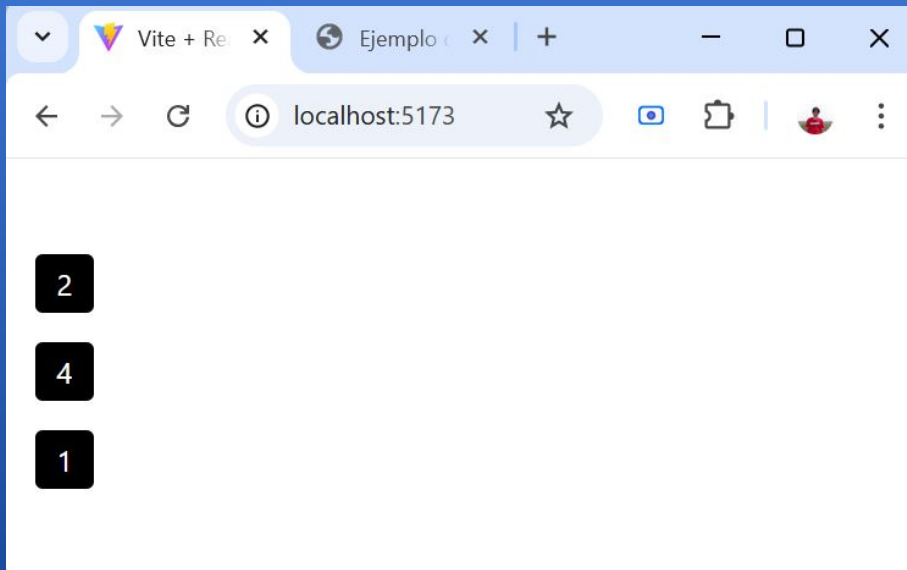
  </div>

);
```

# Componentes

---

- Podemos ver la potencia que puede tener la declaración de componentes y luego crear tantas como necesitemos.
- Si ejecutamos ahora la aplicación desde la línea de comandos de Node.js 'npm run dev' tenemos como resultado:



# Problema

---

- En el problema anterior si necesitamos sortear otros tres dados estamos obligados a recargar la página. Modificaremos la aplicación y agregaremos un botón para que cada vez que se presione nos actualice únicamente los valores de los tres dados sin tener que recargar en forma completa la página.
- Solo debemos efectuar cambios en la función App:



# Problema

```
import Dado from "../Dado";
import { useState } from "react";

function App() {
  function generarValor() {
    return Math.trunc(Math.random() * 6) + 1
  }

  function tirar() {
    setNumero1(generarValor())
    setNumero2(generarValor())
    setNumero3(generarValor())
  }

  const [numero1, setNumero1] = useState(generarValor())
  const [numero2, setNumero2] = useState(generarValor())
  const [numero3, setNumero3] = useState(generarValor())

  return (
    <div>
      <Dado valor={numero1} />
      <Dado valor={numero2} />
      <Dado valor={numero3} />
      <button onClick={tirar}>Tirar</button>
    </div>
  );
}

export default App;
```

# Problema

---

- Guardamos el estado de tres valores aleatorios en `numero1`, `numero2` y `numero3`. Además recordemos que inicializamos tres variables con las funciones que se llamarán cuando necesitemos cambiar los valores de las variables de estado:

```
const [numero1, setNumero1] = useState(generarValor())  
const [numero2, setNumero2] = useState(generarValor())  
const [numero3, setNumero3] = useState(generarValor())
```

# Problema

— — —

- En el bloque JSX recuperamos el 'estado' de los tres valores a pasar a cada componente 'Dado':

```
return (  
  <div>  
    <Dado valor={numero1} />  
    <Dado valor={numero2} />  
    <Dado valor={numero3} />  
    <button onClick={tirar}>Tirar</button>  
  </div>  
);
```

# Problema

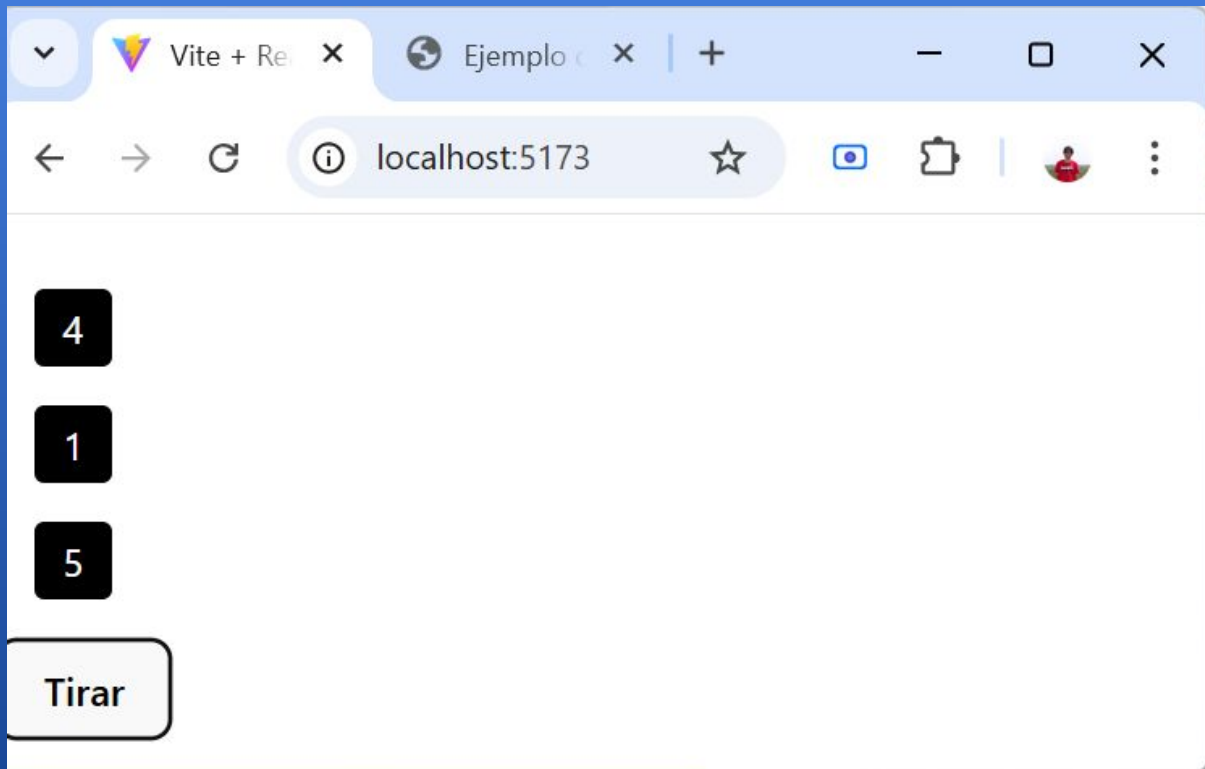
---

- La función tirar() modifica el 'estado' llamando a la función setNumero de cada variable de estado, esto hace que se actualicen en pantalla los nuevos valores de cada dado sin tener que recargar toda la página:

```
function tirar() {  
  
    setNumero1(generarValor())  
  
    setNumero2(generarValor())  
  
    setNumero3(generarValor())  
  
}
```

# Problema

- En el navegador tenemos como resultado:



# EJERCICIOS ADICIONALES PROPUESTOS

— — —





**¡MUCHAS  
GRACIAS!**