```
React es una biblioteca de Javascript para crear interfaces de usuario. Se basa en el uso de componentes.

React se compone de dos librerías: React y ReactDOM

React es la librería con la que podemos definir nuestros componentes y su comportamiento.

ReactDOM es la librería encargada de gestionar el VDOM y las tareas asociadas al renderizado de estos componentes en el navegador.
```

1.6. Servidor *NodeJS*

Para instalar *NodeJS* en *Microsoft Windows* pincha aquí, para en *GNU/Linux Ubuntu* o *Debian* es necesario utilizar estos comandos:

```
curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Aunque en la mayoría de distribuciones funciona la instalación tradicional:

```
sudo apt-get install npm nodejs
```

Esta aplicación viene acompañada de **npm**, que es un gestor de paquetes que permitirá la inclusión de bibliotecas al proyecto que se esté desarrollando. Su uso es muy sencillo y funciona especificando el nombre del paquete que se necesita instalar, como se verá en lo sucesivo.

3. Componentes en `React`

Los componentes permiten separar programas en piezas independientes y reutilizables. Internamente, para *JavaScript*, un componente no es más que una función que recibe un objeto como parámetro, que se denomina `props`, y retorna objetos `HTML` que inserta en el `DOM` del navegador. Se utilizan para dividir la aplicación en trozos manejables, para indicar qué es lo que es necesario ver en pantalla y para hacer un redibujado selectivo, es decir, que cuando los datos cambien sólo se actualice el componente y no el resto de la aplicación.

Quizás, la parte más complicada de esta dinámica es la de dividir la aplicación web en componentes, aunque es una práctica que se adquiere con el tiempo. Un componente debe tratarse de forma independiente aunque debe interactuar con el resto que conforman la aplicación. Será necesario comunicarse entre sí y esta tarea se realiza a través de sus propiedades (`props`). Existen otros métodos de comunicación entre componentes, pero se tratarán en lo sucesivo.

Es importante tener en cuenta que el nombre de los componentes de `React` debe empezar con una mayúscula. De esta forma es como se diferencian a los elementos nativos de `HTML` de los componentes creados de forma pormenorizada. Si no se hace así, el preprocesador del código lo interpreta como si fuese una cadena (`string`) y no como si fuese una función (componente) y no lo dibujará (renderizará) de la manera esperada.

Un componente en `React` está compuesto por dos partes: la parte que contiene la **lógica de funcionamiento** (que se escribe en *JavaScript),* y la **parte que devuelve** (a través de `return`) que está escrita en **JSX** (que representa el código que el navegador mostrará en pantalla).

```
function ComponenteFeo(props) {
    // Lógica del componente en código JavaScript.
  return (
    // Vista del componente que será pintada en el navegador escrita en JSX.
  );
}
```

Un componente sólo puede dibujar (renderizar) una etiqueta cada vez, por lo que todo el contenido **JSX** debe estar encerrado dentro de una etiqueta. Por ejemplo:

```
function Undiv(props){
 return(
    <div>
       Estoy devolviendo sólo un div.
    </div>
    );
}
```

Si es necesario devolver dos etiquetas en la raíz del dibujado, se utilizan los `React Fragments` o encerrando todas las etiquetas en la raíz entre dos etiquetas vacías `<>`:

```
function Dosdivs(props) {
  return (
    <Fragment> // También se puede utilizar <>.
      <div>Estoy devolviendo el primer div.</div>
      <div>Y ahora el segundo div.</div>
    </Fragment> // También se puede utilizar </>.
```

```
  );
}
```

Una vez construido el componente, éste deberá ser llamado como si de una etiqueta `HTML` se tratarse, ya sea de forma autoconclusiva o con etiqueta de apertura y de clausura (la forma de dibujar estos componentes sí importa, como se verá posteriormemte):

```
<UnDiv/>
<DosDivs></DosDivs>
```

Cuando se ejecute este código en el `DOM` del navegador aparecerán las etiquetas `HTML` que devuelve cada uno (situadas en el `return`), es decir:

```
<div>Estoy devolviendo sólo un div.</div>
<div>Estoy devolviendo el primer div.</div>
<div>Y ahora el segundo div.</div>
```

3.1. Tipos de componentes

Existen dos tipos de componentes:

- **Componentes de clases** (*Class components*), utiliza la programación orientada a objetos (POO) para definir un componente. Para que una clase se considere componente en `React` debe heredar de la clase `React.Component` y debe contar con un método `render` el cual debe devolver elementos de `React` (otro componente, código **JSX** o datos, por ejemplo).

  ```
  // Componentes de clase.
  class ComponenteFeo extends React.Component {
    render() {
      return (
        <div className='componente'>
          <h2>Mi primer componente.</h2>
          <div>Aunque un poco feo.</div>
        </div>
      );
    }
  }
  ```

  La principal ventaja de usar *class components* es que es posible utilizar el estado de la información para cambiar su estructura. Esta característica, que se estudiará con posterioridad, era exclusiva de este tipo de componentes hasta el advenimiento de los `Hooks` en la versión 16.8.0 de `React`. Esta adición permite prescindir por completo de las clases y utilizar funciones para el control del estado de la información, práctica altamente recomendable si se tiene en cuenta que *JavaScript* no fue diseñado con el uso de clases en mente.

- **Componentes funcionales** (*Functional Components*), es la forma más sencilla con la que se puede definir un componente en `React` ya que se utilizan funciones tal y como se haría en *JavaScript*. Progresivamente se abandona en uso de los componentes de clase en beneficio de los funcionales. Los primeros quedan relegados a proyectos en producción que necesiten ser actualizados (incluso así es posible crear componentes funcionales nuevos para ese proyecto).

Para que una función se convierta en *functional component* debe aceptar un sólo argumento (`props`) o ninguno y debe devolver código escrito en **JSX**.

```
// Componentes funcionales.
function ComponenteFeo(props) {
  return (
    <div className='componente'>
        <h2>Mi primer componente.</h2>
        <div>Tiene mejor pinta ya que es funcional.</div>
    </div>
  );
}
```

3.2. Propiedades en un componente (`props`)

Cuando se utiliza un componente es posible parametrizarlo a través de las propiedades que se especifican durante su declaración. El parámetro que recibe el componente es un objeto `JSON` que contendrá todos las propiedades que se le pasen a un componente y no hay límite en cuanto a la cantidad. Eso sí, las propiedades son sólo de lectura, es decir, los componentes son funciones puras (no modifican sus parámetros).

Así funciona un componente utilizando propiedades:

```
function Feo2(props) {
  return (
    <div className='componente'>
      <h2>{props.titulo}</h2>
      <div>{props.subtitulo}</div>
      <div>{props.nota}</div>
    </div>
  );
}
```

Y así su declaración dentro de otro componente:

```
<Feo2 titulo='El bueno, el malo…' subtitulo='...y el feo.' nota='90/100' />;
```

El resultado de este código sería:

```
<div className='componente'>
    <h2>El bueno, el malo...</h2>
    <div> ...y el feo.</div>
    <div>90/100</div>
</div>
```

Cada componente dispone de una propiedad especial que es `children`. Hay que recordar que un componente en `React` se maneja como si una etiqueta de `HTML` se tratase, por lo que `children` hará referencia al contenido introducido entre las etiquetas y no al pasado como propiedades. Se va a modificar el componente anterior para que el subtítulo sea pasado a través de `children`:

```
function Feo2(props) {
  return (
    <div className='componente'>
      <h2>{props.titulo}</h2>
      <div>{props.children}</div>
```

```
      <div>{props.nota}</div>
    </div>
  );
}
```

Y esta sería su construcción:

```
<Feo2 titulo='El bueno, el malo…' nota='90/100'>
  ...y el feo.
</Feo2>;
```

El resultado de este código es idéntico al anterior.

Entonces, ¿cuándo se usa uno y cuándo otro? Pues cuando sea necesario que la propiedad de un componente sea otro componente. Por ejemplo, se modifica el componente `<UnDiv>` para que recoja su `children` y le aplique un formato específico (a través de una clase):

```
function Undiv(props) {
  return(
    <div className="un-estilo">
      {props.children}
    </div>
 );
}
```

Ahora tan sólo hay que pasar al componente (a través de su etiqueta de apertura y clausura) el texto que requiera el formato de la clase `un-estilo`:

```
<Feo2 titulo='El bueno, el malo…' nota='90/100'>
    <UnDiv>...y el feo.</UnDiv>
</Feo2>;
```

Este código creará las siguientes etiquetas `HTML` en el `DOM` del navegador:

```
<div className='componente'>
  <h2>El bueno, el malo...</h2>
  <div>
    <div className='un-estilo'>...y el feo.</div>
  </div>
  <div>90/100</div>
</div>;
```

De esta forma es posible construir estructuras complejas a partir de pequeñas porciones de código encapsuladas en componentes.

Otra de las novedades de ***ECMAScript6*** que facilita la escritura de código es que es posible asignar los valores de un objeto `JSON` a variables de un modo rápido y en una sola línea. A esto se le denomina **desestructuración** de objetos y se tratará con algo más de profundidad en lo sucesivo.

Por ejemplo:

```
const persona = {
  nombre: "Feo",
  apellido1: "De Verdad",
  apellido2: "De La Buena"
}
const {nombre, apellido1} = persona;
console.log(nombre) // Muestra Feo.
console.log(apellido1); // Muestra De Verdad.
```

De este modo no es necesario usar la nomenclatura de punto para el acceso a valores de un objeto (ni tampoco los corchetes) y siempre funcionará si las variables de destino se llaman igual que las propiedades del objeto.

En los sucesivo se profundizará en el manejo de esta técnica, pero es buena idea familiarizarse con ella desde el inicio, aunque sea de forma somera, ya que ahorra escribir mucho código.

4. *JavaScript* XML

Para escribir componentes se utiliza la sintaxis **JSX** (*JavaScript* `XML`). Esta es una extensión a *JavaScript* que permite escribir `HTML` sin necesidad de representarlo como un `string` (encerrándolo entre comillas). En realidad es un preprocesador de código que, al ser dibujado (render), se traduce en código `HTML`. Si no existiera este "lenguaje", habría que invocar en cada redibujado de un componente al método del objeto `React`:

```
React.createElement(type, [props], [...children]);
```

De ese modo, para dibujar el siguiente código:

```
function App() {
  return (
    <h1>Hola Mundo</h1>
 );
}
```

habría que escribirlo del siguiente modo:

```
function App() {
  return React.createElement("h1", null, "Hola Mundo");
}
```

JSX facilita la escritura a través de un preprocesador que se encarga de hacer esta conversión.
Quizás este ejemplo no parezca muy práctico, pero si es necesario crear estructuras más complejas utilizando etiquetas `HTML` anidadas:

```
function App() {
  return (
    <div>
      <h1>Bienvenido Feo</h1>
      <button>Unirse a la comunidad</button>
    </div>
  );
}
```

Su traducción a *JavaScript* sería:

```
function App() {
  return React.createElement(
    "div",
    null,
    React.createElement("h1", null, "Bienvenido Feo"),
    React.createElement("button", null, "Unirse a la comunidad")
  );
}
```

Las etiquetas `<h1>` y `<button>` también son transformadas a una llamada a la función `React.createElement(...)` y son pasados como parámetros en la llamada a la función que crea el `<div>` que los contiene.

Como se aprecia en estos sencillos ejemplos, sin la utilización de **JSX** el código se vuelve pesado de escribir y difícil de mantener, por lo que aprender **JSX** es un tiempo que retorna su inversión en un breve lapso de tiempo.

4.1. Expresiones *JavaScript* dentro de JSX

Se pueden incrustar expresiones dentro de **JSX** encerrándolas entre llaves:

```
// Línea de código situada en la parte JavaScript del componente.
const nombre = 'Feo Muy Feo';
// Línea de código situada en la parte JSX (return) del componente.
<h1>¡Adiós, {nombre}!</h1>;
```

y esto al dibujarlo en `HTML` será:

```
<h1>¡Adiós, Feo Muy Feo!</h1>
```

4.2. Dibujado condicional

Otra expresión bastante útil es la que permite dibujar una sección de una aplicación dependiendo de una condición. Un problema con este tipo de expresiones es que no permite implementar una lógica de tipo `if-else` nativa de *JavaScript*. En el caso de ser necesario este tipo de lógicas es posible utilizar el operador ternario:

```
function Viernes(props) {
  let esViernes = false;
  return(
    <div>
      <h1> Bienvenido </h1>
      <h1>{esViernes ? "¡Es viernes!" : "Otro día más..."}</h1>
    </div>
  );
}
```

que al dibujarlo quedaría:

```
<div>
  <h1> Bienvenido </h1>
  <h2>Otro día más...</h2>
</div>
```

Otra limitación importante es que no es posible utilizar estructuras iterativas tipo `while` o `for`. En su lugar se debe utilizar el método `map` de los objetos iterables (principalmente `JSON` y `arrays`). Así, para imprimir el `array`

```
feos = ["Hambre", "Guerra", "Peste", "Muerte"];
```

se haría del siguiente modo:

```
return(
 feos.map((feo) => {
    return <div>{feo}</div>;
 });
);
```

4.3. Valores Ignorados

Las etiquetas que contengan valores booleanos (ya sean `true` o `false`), `null` y `undefined` serán dibujados sin ningún contenido en su interior.

```
<p>{true}</p>
<p>{false}</p>
<p>{true && true}</p>
<p>{null}</p>
<p>{undefined}</p>
<p></p>
```

Todos los ejemplos anteriores transpilan (traducen) a un párrafo vacío.

Si bien es posible escribir aplicaciones en `React` sin utilizar **JSX**, el uso de esta extensión permite agilizar la escritura del código. Las mejoras que ofrece utilizarlo no sólo se quedan en la simplicidad y claridad del código, sino que también permite más productividad y rapidez en el desarrollo de aplicaciones con `React`.

Aunque **JSX** es bastante simple de entender, tiene particularidades que es necesario conocer:

- hay ciertos atributos en `HTML` que son los mismos que palabras reservadas de *JavaScript* por lo que hay que cambiarlos. Por ejemplo `class` por `className` o `for` por `htmlFor`,
- hay que tener en cuenta que la sintaxis de **JSX** es más cercana a la sintaxis de *JavaScript* que de `HTML` y comparten la misma convención al momento de nombrar funciones o variables. Recuerda usar `camelCase`,
- todas las etiquetas se deben cerrar ya sea con una etiqueta de cierre `<div></div>` o por auto cerrado `<img />`,
- sólo permite devolver una única etiqueta, por lo que todas ellas deben estar envueltas dentro de una (aunque esto es más una limitación de componente que de **JSX**):

  ```
  <div>
    <h1>¡Hola Feo!</h1>
    <div>desde Petrer</div>
  </div>
  ```

- se usan las llaves (`{ }`) para insertar expresiones de *JavaScript* en código **JSX**,
- se desaconseja utilizar el estilo en línea a través de la etiqueta `style`. **JSX** no interpreta de forma directa código `CSS`, sino que habrá que utilizar C*amel* C*ase* para los atributos compuestos de `CSS`, como por ejemplo `backgroundColor`. Además este código debe estar encerrado entre dobles llaves `{{ }}`. Evita utilizar estilo en línea ya que siempre debe manejarse a través de las clases.

Teniendo en cuenta estas particularidades, aprender esta extensión de *JavaScript* no resulta una tarea difícil, como se verá en la siguiente práctica.

Aunque todo esto parezca muy limitante, hay que recordar que se está trabajando con el contenido dentro del `return` del componente. Fuera de éste es posible emplear cualquier código en *JavaScript*.