CAPÍTULO 1

# JavaScript para React

## 1.1 Introdução

Algumas funcionalidades da linguagem JavaScript são usadas com frequência no desenvolvimento com React. Neste capítulo, faremos uma breve revisão dessas funcionalidades, quer explicando suas finalidades e exemplificando seu uso com exercícios práticos, quer apenas citando-as e deixando explicação mais detalhada por ocasião de seu uso em scripts no livro ou quando julgado conveniente.

Se você é experiente com a linguagem, particularmente com a ES6, sinta-se à vontade para pular este capítulo, contudo está convidado a recapitular seus conceitos e ter certeza de que está confortável com as funcionalidades listadas a seguir.

- JavaScript e ECMAScript
- Declaração de variáveis
- Operador ternário
- Arrow functions
- Operador spread
- Métodos `map`, `filter` e `find`
- Template Literals (Template String)
- Classes
- Destructuring assignment (Atribuição para uma desestruturação)
- Exports e Imports
- Sintaxe JSX

## 1.2 JavaScript e ECMAScript

JavaScript foi criada em 1995 por Brendan Eich, que à época trabalhava para a Netscape. Em 1996, a Netscape submeteu a linguagem à ECMA (European Computer Manufacturers Association – Associação dedicada à padronização de sistemas de informação), para ser padronizada para uso por fabricantes de navegadores. Em 1997 foi publicada a ECMA262 e, com base nela, a ECMAScript1, primeira versão ECMA da JavaScript.

| Ano | Versão da ECMA |
|---|---|
| 1998 | ECMAScript2 |
| 1999 | ECMAScript3 |
| 2008 | ECMAScript4 (abandonada) |
| 2009 | ECMAScript5 |
| 2011 | ECMAScript5.1 |
| 2015 | ECMAScript6 (ES6) |
| 2016 | ECMAScript7 (ES7) |
| 2017 | ECMAScript8 (ES8) |
| 2018 | ECMAScript9 (ES9) |

À época em que foi criada, o nome JavaScript era patente da Sun Microsytems (hoje Oracle). Quando da normatização da linguagem, ela foi renomeada para ECMASript. Assim, os dois termos se referem exatamente à mesma linguagem. O termo que nos acostumamos a usar é JavaScript, mas o oficial é ECMAScript. Nos dias atuais já se usa com muito mais frequência a forma abreviada ES*versão*.

### 1.2.1 Transpilers JavaScript

Transpilers são compiladores do tipo código-para-código capazes de entender o código escrito em uma linguagem e produzir o código equivalente em outra.

Transpilers JavaScript traduzem código não entendido, ou seja, não normatizado pela ECMA e, portanto, não suportado por dispositivos que entendem JavaScript. O mais conhecido e usado transpiler JavaScript é o Babel (`https://babeljs.io/`).

Os navegadores ainda não dão total suporte às versões ES6 e posteriores, mas já podemos usá-las graças ao uso de transpilers que as "traduzem" para a ES5, que é a versão bem suportada atualmente. Além da ES6, sintaxes tais como as da CoffeeScript, TypeScript e JSX, são exemplos das que necessitam de transpilers para serem suportadas pelos dispositivos. Mais informações você encontra em uma tabela de suporte da ES6 e transpilers em `https://bit.ly/2IxnDAh`.

## 1.3 Declaração de variáveis

Em JavaScript variáveis são containers destinados a armazenar dados. Antes da ES6, a palavra-chave para se declarar uma variável era `var`. A ECMAScript2015 (ES6) criou mais duas palavras-chaves para se declararem variáveis: `const` e `let`. Essas duas variáveis foram criadas para resolver problemas de alteração de reatribuição e de escopo, comuns quando se declara variáveis com uso de `var`.

✔ Atualmente as três palavras-chaves são válidas na linguagem, contudo recomenda-se usar `const` e `let` e evitar o uso de `var` para declarar variáveis.

A palavra-chave `const` (abreviatura de constante) destina-se a declarar variáveis que devem permanecer *fixas* no script. Uma vez declaradas, qualquer tentativa posterior de alterá-la ou reatribuí-la resulta em erro.

A palavra-chave `let` destina-se a declarar variáveis que não precisam permanecer *fixas*, mas cujo escopo limita-se ao local onde foram declaradas.

Se você ainda não se sente confortável com o uso das duas novas formas criadas pela ES6 e que serão amplamente usadas no desenvolvimento com React, leia os tutoriais disponíveis em `https://bit.ly/3oCuR5n` e `https://bit.ly/39QV6ky`, que descrevem as novas sintaxes para declarar variáveis.

## 1.4 Operador ternário

O operador ternário fornece uma sintaxe abreviada para a condicional `if–else` da linguagem JavaScript. Tal sintaxe abreviada é mostrada a seguir.

```
let resultado = (condicao) ? (retorno se verdadeira) : (retorno se falsa);
```

O valor de retorno é armazenado na variável `resultado`. A *`condicao`* é a cláusula `if` (falsa ou verdadeira) do `if–else`. A parte da sintaxe que declara: `? (`*`retorno se verdadeira`*`)` é o valor da variável `resultado` se a condição for verdadeira. A parte que declara: `: (`*`retorno se falsa`*`)` é o valor da variável `resultado` se a condição for falsa.

✔ É facultativo o uso do parêntese na condição e nos retornos mostrados na sintaxe. Se necessário, use-o para aumentar a legibilidade do código.

Observe uma cláusula `if-else` e sua equivalente escrita com operador ternário.

```
if (preco < 40) {
  return "Livro barato!";
} else {
  return "Livro não é barato!";
}
```

Por outro lado, escrita com sintaxe de operador ternário é mostrada a seguir.

```
let resultado = preco < 40 ? "Livro barato" : "Livro não é barato!";
```

A cláusula `elseif` também pode ser escrita com uso do operador ternário. Nesse caso, ela vai causar um aninhamento na sintaxe. Observe a seguir um exemplo de operador ternário aninhado escrito em multilinhas.

```
let preco = 50;
preco < 40
  ? console.log("Livro barato")
  : preco < 70
  ? console.log("Livro não é barato e nem caro!")
  : console.log("Livro é caro");
```

Esses scripts se encontram no CodeSandbox em `https://bit.ly/2JSs1dV` e os resultados podem ser visualizados no console daquela interface.

A partir daqui suponho que você conhece e sabe manipular, pelo menos, as funções básicas do CodeSandbox.

## 1.5 Arrow functions

A ES6 criou uma sintaxe simplificada para escrever funções JavaScript. Além da sintaxe, e, muito mais importante que ela, foram modificados vários comportamentos próprios das funções, com a finalidade de corrigir comportamentos estranhos e não tão fáceis de entender que ocorrem quando são utilizadas funções previstas nas versões anteriores à ES6. A ES6 criou uma nova maneira com sintaxe muito mais amigável e intuitiva de escrever e utilizar funções JavaScript.

Seguem alguns exemplos comparando a sintaxe segundo a ES5 e anteriores e a nova sintaxe prevista na ES6 (arrow function).

#### Exemplo 1

ES5

```
function saudacao() {
  return "Olá visitante"
};
console.log( saudacao() ); // Olá visitante
```

ES6

```
let saudacao = () => "Olá visitante";
console.log( saudacao() ); // Olá visitante
```

Foi abolida a palavra `function` e, quando o retorno da função se faz na mesma linha de código da seta `=>`, não há necessidade de uso da palavra `return` entre sinal de chaves, tal como mostrado a seguir.

ES6

```
let saudacao = () => (
  "Olá visitante"
  );
console.log( saudacao() ); // Olá visitante
```

Quando o retorno da função for na mesma linha de código depois da seta `=>`, use parênteses e não há necessidade de uso da palavra `return` tal como mostrado.

A função é atribuída a uma variável que passa a fazer referência a ela (no exemplo mostrado, a variável `saudacao`). A partir da seta ( `=>`), escreve-se o script a ser processado quando a função for invocada e o resultado desse script é o valor retornado pela função. Antes da seta estão os dados de entrada (ou parâmetros) da função e depois da seta, a saída da função (ou seu valor de retorno). Os exemplos a seguir mostram e esclarecem detalhes da sintaxe.

#### Exemplo 2

ES5

```
function dobrar(x) {
  return 2 * x;
}
console.log( dobrar(5) ); // 10
```

ES6

```
let dobrar = (x) => 2 * x;
console.log( dobrar(5) ); // 10
```

Nas funções que admitem um ou nenhum parâmetro, o sinal de parênteses envolvendo o parâmetro é opcional, e geralmente não é usado, mas você é livre para fazer sua escolha. Assim a sintaxe alternativa (sem parênteses) para a função é mostrada a seguir.

ES5

```
let dobrar = x => 2 * x;
console.log( dobrar(5) ); // 10
```

ES6

```
let dobrar = (x) => {
  return 2 * x;
}
console.log( dobrar(5) ); // 10
```

⚠ Neste livro, adotaremos a sintaxe com uso de parênteses.

**Exemplo 3**

ES5

```
function avaliar(x, y, z) {
  if( x < 10) {
    return y + z;
  } else {
    return y * z;
  }
};
console.log(avaliar(18, 15, 5)) // 75
console.log(avaliar(2, 3, 11))  // 14
```

ES6

```
let avaliar = (x, y, z) => x < 10 ? y + z : y * z;
console.log(avaliar(18, 15, 5)); // 75
console.log(avaliar(2, 3, 11)); // 14
```

Nesse exemplo, usou-se o operador ternário em lugar da cláusula `if-else`, conforme estudado anteriormente.

Alternativamente você poderá escrever o operador ternário em linhas separadas. Nesse caso, a sintaxe multilinha para o operador é conforme mostrada a seguir.

```
let avaliar1 = (x, y, z) => (
  x < 10 // condição
  ? y + z // linha começa com ?
  : y * z; // linha começa com :
);
```

✔ A seta tem de ficar na mesma linha dos parâmetros, a condição em uma linha, e as linhas seguintes começando com `?` (interrogação) e `:` (dois pontos). Pule linhas se pretende aumentar a legibilidade do código. Observe que na sintaxe com o script escrito em linhas nesse exemplo o uso de parênteses depois da seta é *`obrigatório`*.

A sintaxe para autoinvocar a função – IIFE (Immediately Invoked Function Expression) – consiste em simplesmente envolvê-la em parênteses como mostrado no exemplo a seguir.

```
console.log( ( x => 2 * x )(40) ); // 80
```

Antes da ES6, o `this` da JavaScript era do escopo da função, de difícil compreensão, não raro mudava de escopo dentro da função e fonte de bugs no código. Arrow function alterou a maneira como `this` se vincula ao escopo, tornando-o mais intuitivo e de mais fácil entendimento. As alterações mais significativas são as relacionadas a seguir:

- Não mais existe `this` no escopo de arrow functions. Quando se usa arrow function `this` se refere ao escopo imediatamente acima do escopo da função. Caso o escopo da função seja outra arrow function, ele não é considerado para efeito de `this` e assim por diante até o escopo global.
- Pelo fato de que `this` pertence ao escopo do seu container, os métodos `call()`, `apply()` e `bind()` não funcionam em arrow functions.

Neste livro não vamos nos aprofundar no estudo do novo comportamento de `this`. Quando for julgado conveniente, faremos os devidos comentários.

✔ Recomendo a leitura do excelente tutorial, com vários exemplos práticos mostrando o comportamento de `this`: `https://bit.ly/3oDdx0d`.

Os scripts mostrados anteriormente e que ilustram arrow functions encontram-se no CodeSandbox em `https://bit.ly/3n3tdc0` e os resultados podem ser visualizados no console daquela interface.

## 1.6 Operador spread

O operador `spread` da ES6 destina-se a expandir um array, um objeto ou uma string. O símbolo para esse operador é constituído por três pontos (…).

Vejamos alguns exemplos e sua aplicação prática, considerando as seguintes constantes que armazenam arrays:

```
const livros1 = ["CSS3", "JavaScript", "PHP"];
const livros2 = ["HTML5", "React"];
const precos = [70, 30, 90, 100, 10];
```

#### Exemplo 1

```
let livros = livros1 + livros2;
```

O resultado não é um array com os cinco livros como era de se esperar, e sim uma string como mostrada a seguir.

```
CSS3,JavaScript,PHPHTML5,React
```

#### Exemplo 2

```
let livros = [livros1] + [livros2];
```

O resultado não é um array com os cinco livros como era de se esperar, e sim um array de dois arrays como mostrado a seguir.

```
[ ["CSS3", "JavaScript", "PHP"], ["HTML5", "React"] ]
```

#### Exemplo 3

```
let livros = [...livros1, ...livros2];
```

Agora sim! O uso do operador `spread` produz o resultado que se esperava nos dois exemplos anteriores, um array com os cinco livros como mostrado a seguir.

```
[ "CSS3", "JavaScript", "PHP", "HTML5", "React" ]
```

#### Exemplo 4

```
let precoMaximo = Math.max(precos);
```

Aqui passamos um array de números como parâmetro da função `Math.max()`. Isso não é permitido em ES6. O resultado é uma mensagem informando que o parâmetro `precos` não é um número, `NaN`.

#### Exemplo 5

```
let precoMaximo1 = Math.max(...precos);
```

Agora sim! O uso do operador `spread` produz o resultado que se esperava, o número 100, que é o valor máximo constante do array de preços.

A possibilidade de se passar um array como parâmetro de uma função é uma das mais poderosas funcionalidades do operador `spread`.

Esses exemplos encontram-se no CodeSandBox em `https://bit.ly/37V6eKK`.

## 1.7 Métodos map(), filter() e find()

Esses três métodos são muito usados em React e é essencial que você se sinta totalmente confortável com seu uso. Basicamente eles manipulam objetos e se destinam a percorrer os itens de um objeto (inicial) iterável e criar um objeto

(novo) com itens que resultam da manipulação dos itens do objeto (inicial). Esses métodos não alteram o objeto (inicial), e sim criam um objeto (novo).

A título de observação, pois não entraremos em detalhes sobre isso, esclareço que, além dos parâmetros que cada um desses métodos admite, é possível também definir um objeto para servir de referência como `this` na função callback. Ficaremos restritos a dois dos três parâmetros desses métodos.

### 1.7.1 Método map()

Esse método admite três parâmetros. O primeiro, obrigatório, é uma função callback, o segundo, opcional, o índice do item, e o terceiro, o objeto original. Esse método percorre cada item de um objeto iterável (por exemplo: um array) e cria um novo objeto no qual cada item é o retorno da função callback aplicada sobre cada item do objeto original. Os exemplos mostrados a seguir esclarecem o que dissemos.

**Exemplo 1**

```
const livros = ["CSS3", "HTML5", "JavaScript", "React", "PHP"];
  let livros1 = livros.map( (livro) => "Livro " + livro );
  let livros2 = livros.map((livro, index) => "Livro" + index + " " + livro);
console.log(livros);
console.log(livros1);
console.log(livros2);
```

No exemplo mostrado, usamos a função `map()` para criar dois novos arrays, `livros1` e `livros2`, que foram mapeados do array original `livros`. Os exemplos são esclarecedores por si só e dispensam mais explicações. Observe que no segundo exemplo usamos o índice do array original, capturando-o como o parâmetro `index` da função callback.

**Exemplo 2**

```
const livrosA =
[
  {titulo: "Construindo Sites com HTML", autor: "Maurício Samy Silva"},
  {titulo: "Web Scraping com Python", autor: "Ryan Mitchell"},
  {titulo: "CSS3", autor: "Maurício Samy Silva"}
];
let livrosX = livrosA.map((livro) => "Livro: " + livro.titulo);
let livrosY = livrosA.map((livro) => "Autor: " + livro.autor);
```

```
console.log(livrosA);
console.log(livrosX);
console.log(livrosY);
```

No exemplo mostrado, criamos dois novos arrays, `livrosX` e `livrosY`, que foram mapeados do array de objetos original `livrosA`. Os exemplos são esclarecedores por si só e dispensam mais explicações.

Esses exemplos encontram-se no Code Sandbox em `https://bit.ly/39SJ2iN` e os resultados podem ser visualizados no console daquela interface.

### 1.7.2 Método filter()

Esse método admite três parâmetros. O primeiro, obrigatório, é uma função callback, o segundo, opcional, o índice do item, e o terceiro, o objeto original. Esse método percorre cada item de um objeto iterável (por exemplo: um array) e cria um novo objeto no qual cada item satisfaz uma condição de filtragem expressa na função callback aplicada sobre cada item do objeto original. Os exemplos mostrados a seguir esclarecem o que dissemos.

#### Exemplo 1

```
const livros =
[
  {titulo: "Construindo Sites com HTML", autor: "Maurício Samy Silva"},
 {titulo: "Web Scraping com Python", autor: "Ryan Mitchell"},
  {titulo: "CSS3", autor: "Maurício Samy Silva"}
];
let livros1 = livros.filter((livro) => livro.titulo === "CSS3");
let livros2 = livros.filter((livro) => livro.autor === "Maurício Samy Silva");
let livros3 = livros.filter((livro) => livro.titulo.includes("com"));
console.log(livros1);
console.log(livros2);
console.log(livros3);
```

Esses exemplos encontram-se no CodeSandbox em `https://bit.ly/3n4npj0` e os resultados podem ser visualizados no console daquela interface.

### 1.7.3 Método find()

Esse método admite três parâmetros. O primeiro, obrigatório, é uma função callback, o segundo, opcional, o índice do item, e o terceiro, o objeto original.

Esse método percorre cada item de um objeto iterável (por exemplo: um array) e cria um novo objeto no qual cada item satisfaz a condição expressa no retorno da função callback aplicada sobre cada item do objeto original. O exemplo mostrado a seguir esclarece o que dissemos.

**Exemplo 1**

```
const livros =
  [
    {id: 1, titulo: "Construindo Sites com HTML"},
    {id: 2, titulo: "Web Scraping com Python"},
    {id: 3, titulo: "CSS3"}
  ];
let livros1 = livros.find((livro) => livro.id === 3);

console.log(livros1); // {id: 3, titulo: "CSS3"}
console.log(livros1.titulo); // CSS3
```

Esses exemplos encontram-se no CodeSandBox em `https://bit.ly/3n6uBeO` e os resultados podem ser visualizados no console daquela interface.

## 1.8 Template Literals

Template Literals é uma funcionalidade criada pela ES6 com a finalidade de incrementar e ampliar as funcionalidades de Template Strings que existia nas versões anteriores da ES.

Trata-se de uma nova sintaxe para se declararem strings, a qual forneceu à linguagem um construtor semelhante aos existentes em outras linguagens e amplamente populares. Antes da ES6, a sintaxe para se declarar uma string impunha o uso de aspas simples ou duplas envolvendo a string. Concatenar string impunha o uso de sinal de adição ou o método `concat()`. Pular linhas ou tabular strings impunha o uso de caracteres especiais escapados e assim por diante.

A nova sintaxe prevê o uso do sinal de crase ( ` ) para envolver a string. Adeus aspas e boas-vindas à crase, pois ela tornou a declaração de strings muito mais poderosa. Adeus caracteres de escape para espaçamentos e multilinhas, pois os espaçamentos e mudança de linha inseridos no código são respeitados na renderização.

Observe os exemplos a seguir e as respectivas saídas.

**Exemplo 1**

```
const livro = `Livro 'React' do "Maujor"`;
console.log(livro);
//Saída: Livro 'React' do "Maujor"

const livro1 =`Livro 'React'
do "Maujor"`;
console.log(livro1);
// Saída: Livro 'React' do "Maujor"

const marcacao =
`<!DOCTYPE html>
<html lang="pt-br">
  <head>
    ...
  </head>
  <body>
    ...
  </body>
</html>`;
console.log(marcacao);
//Saída:
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    ...
  </head>
  <body>
    ...
  </body>
</html>
```

Template literals permite interpolação de variáveis e/ou expressões dentro de uma string. Observe o exemplo a seguir.

**Exemplo 2**

```
const titulo =`Livro React do "Maujor"`;
const preco = 80;
```

```
const mensagem =
  `O preço normal do ${titulo} é de R$${preco},00
  Na promoção o preço cai para R$${preco * 0.8},00`;
console.log(mensagem);
//Saída:
O preço normal do Livro React do "Maujor" é de R$80,00
Na promoção o preço cai para R$64,00
```

Observe que a sintaxe para interpolar em uma string é `${ }`. Dentro do sinal de chaves pode-se inserir o nome de uma variável ou qualquer expressão válida na linguagem JavaScript.

Esses exemplos encontram-se no CodeSandBox em `https://bit.ly/371FQzl`.

Outra funcionalidade de Template Literals é a possibilidade de se criarem templates com uso de funções. Trata-se de uma funcionalidade avançada que não será abordada neste livro. Fica a nota apenas à título de informação.

## 1.9 Classes

Até a ES6, a ausência de classes na linguagem JavaScript era motivo de estranheza para muitos desenvolvedores e até mesmo para desconsiderarem o uso da linguagem. Contudo existe em JavaScript a possibilidade de se criarem scripts que simulem classes usando-se funções. São scripts bem estranhos e verbosos e à primeira vista difíceis de se aceitar e entender, principalmente para aqueles acostumados com linguagens que usam classes "verdadeiras".

O que a ES6 trouxe de novidade foi criar uma sintaxe mais intuitiva e simples, incrementar as funcionalidades dos verbosos scripts de antes e chamar de classe, quando na verdade nada mais são do que um mecanismo para criação de objetos, como sempre foram na linguagem. Nesse sentido, classes nada mais são do que construtores personalizados usados para definir tipos de referências personalizados. Tipos de referência em JavaScript são a coisa mais próxima a classes.

Não faz parte do escopo deste livro aprofundar o estudo de classes. Quando necessário, faremos os devidos comentários. Faço esse registro porque, para se criarem componentes (abordados na Seção 2.3.1), podemos usar funções ou classes.

A título de informação e por razões didáticas criaremos um componente do tipo classe ao projetar a aplicação de uma tabela de livros no Capítulo 3. Leitura complementar sobre classes JavaScript em `https://mzl.la/3ozmsQj`.

## 1.10 Atribuição via desestruturação (Destructuring assignment)

Uma das melhorias introduzidas pela ES6 foi a chamada *atribuição via desestruturação*. Trata-se de um recurso que simplifica o acesso aos dados contidos em arrays ou objetos, criando-se variáveis que armazenam os dados.

Atribuição via desestruturação é a sintaxe para uma expressão JavaScript que possibilita a extração de dados de arrays ou propriedades de objetos com uso de variáveis distintas. Para um array ou objeto com *n* itens, cria-se um conjunto de *n* variáveis.

A explicação é bem simplista, porém suficiente para começar o seu estudo, contudo a desestruturação é uma sintaxe com recursos bem mais poderosos. O exemplo a seguir esclarece a sintaxe. Mais informações em `https://mzl.la/39SF7Cx`.

⚠ A desestruturação não modifica o array ou objeto original. Ela copia seus itens e os atribui a variáveis.

Observe dois exemplos de desestruturação.

#### Exemplo 1 – Desestruturação de um array

Sem desestruturação – antes da ES6

```
const livros = ["CSS3", "HTML5", "JavaScript", "React"];
console.log(livros[0]);  // CSS3
console.log(livros[3]);  // React
```

Com desestruturação – a partir da ES6

```
const livros = ["CSS3", "HTML5", "JavaScript", "React"];
let [css, html5, js, react] = livros; // aqui a desestruturação
console.log(css);    // CSS3
console.log(react);  // React
```

#### Exemplo 2 – Desestruturação de um objeto

Sem desestruturação – antes da ES6

```
const livros = [
  { titulo: "React", autor: "Maurício Samy Silva" },
  { titulo: "Node Essencial", autor: "MRicardo R. Lecheta" },
  { titulo: "UX Desing", autor: "Will Grant" }
];
console.log(livros[0].titulo);  // React
console.log(livros[2].autor);   // Will Grant
```

Com desestruturação – a partir da ES6

```
const livros = [
  { titulo: "React", autor: "Maurício Samy Silva" },
  { titulo: "Node Essencial", autor: "MRicardo R. Lecheta" },
  { titulo: "UX Desing", autor: "Will Grant" }
];
let [lUm, lDois, lTres] = livros;
console.log(lUm.titulo);  // React
console.log(lTres.autor); // Will Grant
```

Esses exemplos encontram-se no CodeSanbox em `https://bit.ly/3lXCodf`.

## 1.11 Import e export

A ES6 introduziu na linguagem JavaScript as diretivas `import` e `export`, destinadas a importar arquivos de scripts e a definir arquivos de scripts a serem exportáveis.

React faz uso exaustivo dessas diretivas, assim vamos mostrar um exemplo que dá uma boa ideia de como funcionam arquivos que importam scripts. Arquivos importados em uma página contêm códigos para serem executados em um arquivo JavaScript.

Com certeza você já sabe que para usar uma biblioteca, framework ou genericamente um script JavaScript dentro de uma página HTML é preciso que se use a tag `script` com o atributo `src` apontando para o endereço onde se encontra hospedado o script a ser usado na página conforme mostrado no exemplo a seguir.

```
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
```

A diretiva `import` possibilita que você use um script dentro de outro script, ou seja, importe um script para dentro de outro script. E use esse outro script em uma página HTML. Esse mecanismo é largamente usado em React.

No exemplo mostrado a seguir criamos uma página HTML que usa o script `main.js`, que por sua vez importa o script `utils.js`. O script `utils.js` define duas funções (`estudar()` e `elogiar()`) que são declaradas exportáveis.

Observe os códigos mostrados a seguir e você entenderá o mecanismo de importar e exportar da ES6.

**utils.js – arquivo com funções a exportar**

```
const estudar = (texto) => { console.log(texto); }; ❶
const elogiar = (elogio) => { console.log(elogio);}; ❶
```

```
export default estudar; ❷
export { elogiar }; ❷
```

❶ Função a ser exportada. É válido reunir várias funções a serem exportadas em um só arquivo.

❷ Normalmente no final do arquivo (opcionalmente no início) declara-se a diretiva `export`. Observe que existem dois tipos de exportação para as funções constantes do arquivo, um é o `default` e outro é com uso de chaves `{ }`. A exportação com uso de `default`, em geral, é para a função principal, ou única, existente no arquivo (módulo). Havendo mais funções no módulo, usa-se a exportação com uso de chaves `{ }`. Nesse caso, declara-se o nome das funções separadas por vírgula; por exemplo: `{elogiar, criticar, reclamar}`.

**main.js – arquivo que importa as funções de utils.js**

```
import estudar from "./utils.js"; ❶
import { elogiar } from "./utils.js"; ❷

const btn1 = document.querySelector("#btn1"); ❸
btn1.addEventListener("click", () => {
  estudar("Vamos estudar React.");
});
const btn2 = document.querySelector("#btn2"); ❸
btn2.addEventListener("click", () => {
  elogiar("O livro React do Maujor é muito bom!");
});
```

❶ Importa a função default.

❷ Importa a função não default.

❸ O clique em botão existente na página `index.html` comprova o funcionamento da importação.

**index.html – arquivo usa o script main.js**

```
<body>
  <h1>Exemplos</h1>
  <p><code>( import / export )</code></p>
  <button id="btn1">Estudar</button>
  <button id="btn2">Elogiar</button>

  <script type="module" src="./js/main.js"></script>
</body>
```

Esse exemplo encontra-se no CodeSandbox em `https://bit.ly/3n2CmC9`. Visite o exemplo, abra o console daquela interface, clique nos botões e observe a mensagem no console.

## 1.12 Sintaxe JSX

JSX é a abreviatura para JavaScript XML, uma tecnologia criada com a finalidade de simplificar e facilitar a escrita de códigos JavaScript e amplamente usada em React. Trata-se de uma sintaxe declarativa que descreve, com uso de JavaScript, HTML e CSS, a estrutura, o comportamento e a apresentação de um componente.

Observe um exemplo de criação de marcação HTML com uso de sintaxe JavaScript puro.

```
let elemento = document.createElement("h1");
let titulo = document.createTextNode("Livro React do Maujor");
const resultado = elemento.appendChild(titulo);

Resultado: <h1>LivroReact do Maujor</h1>
```

O mesmo exemplo anterior escrito com sintaxe JSX:

```
let resultado = <h1>Livro React do Maujor</h1>;
```

Nesse exemplo, uma linha de sintaxe JSX produz o mesmo efeito que três linhas de JavaScript puro.

Observe a sintaxe JSX para inserção de atributos.

```
let minhaId = "topo";
const elemento = <div id = { minhaId }>TOPO</div>;

Resultado: <div id="topo">TOPO</div>
```

Arquivos contendo sintaxe JSX devem ser gravados com extensão `.js`, ou, opcionalmente, `.jsx`, pois são arquivos JavaScript. Desde que navegadores não entendem a sintaxe, é necessário um transpiler (ver Seção 1.2.1).

React usa o transpiler Babel para fazer a "tradução" (transpilação) de JSX para JavaScript, mas você não precisa se preocupar, pois Babel já vem configurado e pronto para uso na biblioteca React. Se você não usar o comando `react-create-app` (estudaremos esse comando adiante) para criar a estrutura inicial da sua aplicação, deverá configurar manualmente um pacote chamado Webpack para fazer funcionar Babel. Aprenda React primeiro e depois, quando, e se necessário, pense em configurações manuais.

JSX é uma sintaxe XML, portanto todas as tags devem ser fechadas seja com a tag de fechamento específica, seja com uso de barra como em `<br />`, `<img />`, `<div />`, `<p />` ou `<p></p>`.

Nomes de atributos devem seguir a sintaxe camelCase como em `onClick`, `onSubmit` e `colSpan`.

Os atributos HTML `class` e `for` devem ser escritos `className` e `htmlFor`, pois todas as palavras reservadas da JavaScript, tais como `class` e `for`, em JSX têm sua sintaxe modificada. Para um artigo sobre a sintaxe JSX dos atributos HTML, consulte `https://bit.ly/33Ye9pi`.

Pode-se inserir qualquer expressão JavaScript na sintaxe JSX, desde que envolvidas em sinal de abre e fecha chaves `{ aqui expressão JavaScript }`.

Um erro comum cometido por iniciantes em JSX é a inserção de statements (declarações) JavaScript entre os sinais de abre e fecha chaves. Somente expressões e não declarações JavaScript funcionam na sintaxe JSX. Expressões produzem um valor (por exemplo: x + 4) e declarações executam uma ação (por exemplo: a condicional `if-else`).

Estilização inline é perfeitamente normal em React, embora em sintaxe JSX a aplicação de estilos seja diferente daquela usada em HTML. Você deve criar um objeto contendo as declarações de estilos e aplicá-lo com uso do atributo `style`. Em sintaxe JSX são válidas as duas formas mostradas a seguir.

Objeto contendo estilização inserido diretamente inline.

```
<p style = { { color: "red", fontSize: "28px", textTransform: "uppercase" } }>
  Texto estilizado inline
</p>
```

Objeto contendo estilização definido e aplicado inline por referência.

```
let estilosInline = {
color: "red",
fontSize: "28px",
textTransform: "uppercase"
}

<p style = { estilosInline }>Texto estilizado inline</p>
```

Propriedades CSS com nomes separados com hífen devem ser escritas com sintaxe camelCase (por ex. `backgroundColor` e `textTransform`), tal como em JavaScript.

Valores CSS são strings. Use aspas na sintaxe do objeto.

Estilos inline não funcionam se aplicados para media queries, animações, pseudoclasses e pseudoelementos.

Em campos de formulário, valor (valor entrado no campo) e valor default (valor quando o campo é carregado) têm sintaxe diferente da HTML. Observe a seguir.

Sintaxe HTML:

```
<textarea>Deixe seu comentário</textarea>
<select>
<option value="css" selected>Livro CSS</option>
```

Sintaxe JSX:

```
<textarea defaultValue = { Deixe seu comentário} />
<select defautValue="css">
<option value="css">Livro CSS</option>
```

Em JSX, ao contrário da JavaScript, o evento `onChange` é reconhecido e disparado nos campos de formulário `radio`, `select`, `checkbox` ao serem selecionados e `input` e `textarea` ao se digitar um caractere neles.

Para dificultar ataques XSS ao script, JSX escapa os caracteres nas expressões. Atenção ao usar entidades HTML em strings inseridas em expressões. Observe o exemplo a seguir.

Caractere não inserido em expressão renderiza normalmente.

```
<p>Maujor &reg;</p>
Resultado: Maujor ®
```

Caractere inserido em expressão resulta em erro.

```
<p>{ 'Maujor &reg;' }</p>
Resultado: Erro de parseamento
```

Para inserir caracteres em *`expressões`*, use a codificação Unicode para o caractere. Consulte uma tabela de correspondência de codificação de caracteres em `https://bit.ly/39R8khc`.

```
<p>{ 'Maujor \u00AE' }</p>
Resultado: Maujor ®
```

Em JSX espaços em branco, em qualquer quantidade, sejam horizontais ou verticais (pulos de linha) são reduzidos a um espaço. Se os espaços em branco estiverem no início ou no final da linha, serão ignorados. Para obter espaços em branco na horizontal, use o caractere ` ` ou seu equivalente Unicode `\u00A0` se inserido em expressões.

Para inserir comentários no código, use as sintaxes mostradas a seguir. A primeira para blocos de comentários e a segunda para comentários em linha.

```
{ /*
Aqui comentário
Aqui mais comentário
*/ }
```

ou

```
{
// Aqui comentário
// Aqui mais comentário
}
```

Observe que, ao contrário de JavaScript puro, em JSX é necessário que se envolva o comentário entre sinal de chaves `{ }`.