> *Ficha Prática Nº9 (React – Estados e Interação com utilizador)*

Nas ultimas aulas práticas, foram introduzidos dois conceitos essenciais à criação de aplicações em React: **componentes** e ***props*** (propriedades). Esta ficha, introduz outro conceito fundamental, aquando criação de uma aplicação em React, designado como *states* – **estados**, que serão aplicados no jogo de memória. De facto, uma das grandes tarefas de um programador *front-end* em React, consiste em implementar uma boa *gestão de estados da aplicação*. Dependendo das características da aplicação, esta gestão varia, acabando por ter uma complexidade elevada em aplicações de maior dimensão o que leva, muitas vezes, a recorrer a bibliotecas externas que ajudem e facilitem essa tarefa.

Com referido anteriormente, a implementação de componentes em React pode ser efetuada recorrendo a **componentes de classe** ou **componentes funcionais**. Aquando do lançamento da primeira versão do React, os componentes funcionais, eram bastante limitados, designados de *stateless components*, e como tal, para a criação de aplicações e componentes mais complexos, que envolvessem estados e reutilização da lógica do componente, entre outras características, era necessário recorrer a componentes de classes. No entanto, o surgimento dos **Hooks**, integrados a partir da versão 16.8 do React em 2018, possibilitou que componentes funcionais pudessem incluir o conceito de **estados**, gestão do ciclo de vida, e outras funcionalidades, que até então, só eram possíveis com recurso a componentes de classe, passando assim a ser possível implementar uma aplicação complexa, na sua totalidade, recorrendo apenas a componentes funcionais. O que essencialmente distingue estes dois métodos é a sua sintaxe, mais propriamente, a forma de declaração de um componente, a passagem de propriedades, a manipulação de estados e gestão do ciclo de vida do componente. Apesar dos hooks apresentarem algumas vantagens como simplificação de escrita e compreensão de código, quando comparado com componentes de classe, ou mesmo no contexto de deixar de ser, de todo, necessário o uso da palavra-chave *this*, que tem comportamento diferente em JavaScript quando comparado com outras linguagens e assim reduzir a confusão que esta possa causar quando aplicada em React; continua a ser possível a implementar componentes de classe. A escolha é uma opção de implementação, tendo em consideração as características e complexidade da aplicação ou mesmo preferências de cada programador. Portanto, os Hooks possibilitam funcionalidades adicionais a componentes funcionais e não substituem, pelo menos até ao momento, componentes de classe, que ainda apresentam algumas características que não podem ser implementadas apenas com componentes funcionais.

> *Hooks - useState*

Os **Hooks** são funções JavaScript reutilizáveis com duas regras importantes a ter em consideração quando se pretende utilizar numa aplicação React:

- → **devem ser declarados no topo** do componente e não em funções regulares de JavaScript;
- → **não devem ser** usados em condições, ciclos ou funções "aninhadas".

O React já disponibiliza um conjunto de ***built-in* Hooks**, no entanto, é possível também criar os próprios Hooks (https://reactjs.org/docs/hooks-reference.html. No contexto desta ficha, será usado o Hook **useState**, que, como o próprio nome indica, permite criar variáveis de estado em componentes funcionais. Para além deste, existem outros Hooks, com características e funções diferentes, sendo facilmente distinguíveis pois também iniciam com o prefixo ***use***, estando, no entanto, fora do âmbito desta ficha.

O conceito de **estado** é um conceito chave em React. Um estado permite armazenar valores de propriedades que pertencem ao componente e quando esses objetos, ou quando o estado desse objeto se altera, informa que deverá ser reavaliado de forma a agendar e desencadear uma novamente renderização. Assim, quando se pretende associar algo visível na aplicação a uma variável de forma a que quando se altera essa variável, isso seja refletido também na aplicação, obrigando a uma novamente renderização, é necessário recorrer a uma **variável de estado**. O Hook **useState** retorna um *array* de dois valores. O primeiro elemento do array refere-se ao valor da **variável de estado** em si e o segundo elemento é uma **referência da função** que permite alterar o valor do estado, dessa variável. Ao criar a variável de estado com o useState, pode-se usar o conceito de desestruturação de *arrays,* existente a partir do ES6, que permite armazenar os valores obtidos pelo *useState* diretamente para variáveis, como se apresenta no código seguinte:

```
const [estado, setEstado] = useState(estadoInicial);
```

Assim, a variável `estado` é inicializada com o valor especificado em *estadoInicial* e a função `setEstado` é o nome dado à função que permite alterar o valor do estado dessa variável. Seguindo a convenção, é comum especificar a função com o prefixo *set*, seguido do nome da variável. Quando ocorre o primeiro render, o valor da variável de estado usado para a aplicação é o estado especificado como argumento, aquando a sua criação. Quando o valor da variável de estado se altera é então agendada uma nova renderização, de forma a que essa alteração se reflita na interface (UI) da aplicação. Note que, as variáveis regulares não irão atualizar o UI. Para usar o Hook useState é necessário efetuar o *named import* do useState da biblioteca do React, como apresentado abaixo, caso contrário, não será possível a sua utilização.

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

> Eventos

A **manipulação de eventos** em React é muito semelhante à manipulação de eventos em elementos DOM, com algumas diferenças na sintaxe:

- → Deve-se usar *camelCase* na especificação do evento;
- → Em JSX não é passada uma string, mas sim a função que manipula o evento;
- → Não é possível retornar falso, para evitar o comportamento padrão em React. É necessário invocar de forma explicita do *preventDefault*.

Quando se usa React, geralmente não é necessário invocar *addEventListener* para adicionar *listeners* a um elemento DOM depois deste ser criado. Em vez disso, apenas se especifica um "ouvinte" quando o elemento for renderizado inicialmente. A função de manipulação de eventos, bem como outras funções devem ser passadas como *props* para os componentes filhos. Quando se pretende passar um parâmetro para um manipulador de evento ou um *callback* pode-se recorrer a uma *arrow function* no qual envolve o manipulador de eventos e passar parâmetros, ou então recorrendo a atributos data.

> Exemplos Práticos – Variável de Estado e Evento

Abaixo apresenta-se um simples exemplo de um componente que contém uma variável de estado de nome `visível` de forma a apresentar toda a descrição associada ao logotipo apresentado ou, caso contrario, esconder parte dele. Este comportamento existe quase se clica no texto **Ler mais** / **Reduzir**, respetivamente. Neste simples exemplo, tanto a variável de estado como o evento estão especificados no componente. As figuras apresentam o exemplo pretendido bem como o código para que este comportamento exista.

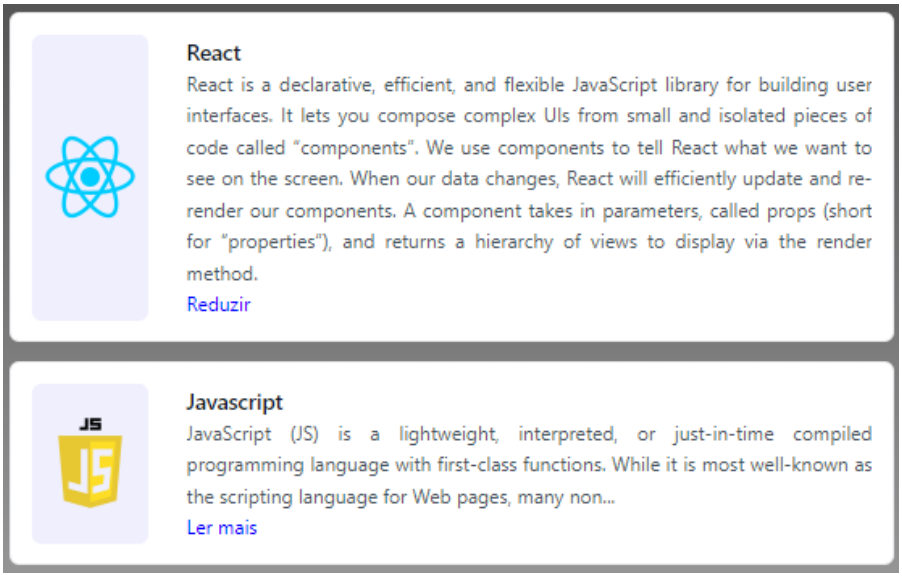

Figura 1 - Exemplo de uso de variavel de estado e evento click

```
import React, { useState } from "react";
import "./exemplo.css";
function InfoComponent(props) {

  const [visible, setVisible] = useState(true);

  const texto = props.children;
  const maxLength = 200;

  function getText() {
    let text;
    if (texto.length <= maxLength) {
      text = texto;
    } else if (visible) {
      text = `${texto.substr(0, maxLength).trim()}...`;
    } else text = texto;
    return text;
  }
  return (
    <>
      <div className="wrapper">
        <div className="logo">
          <img src={props.src} alt={props.title} />
        </div>
        <div className="text">
          <h2>{props.title}</h2>
          <p>
              <p>{getText()}</p>
            { visible
                ? ( <a onClick={() => setVisible(true)}> Reduzir </a> )
                : ( <a onClick={() => setVisible(false)}>Ler mais </a>  )
               }
          </p>
        </div>
      </div>
    </>
```

```
<InfoComponent title="React" src="./assets/images/react.png">React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”. We use components to tell React what we want to see on the screen. When our data changes, React will efficiently update and re-render our components. A component takes in parameters, called props (short for “properties”), and returns a hierarchy of views to display via the render method.</InfoComponent>
```

No exemplo seguinte, foram efetuadas alterações de forma a que a variável de estado esteja no componente pai, de forma a que, ao **expandir** o texto, expande todos, assim como, ao ***reduzir***, contrai todos.

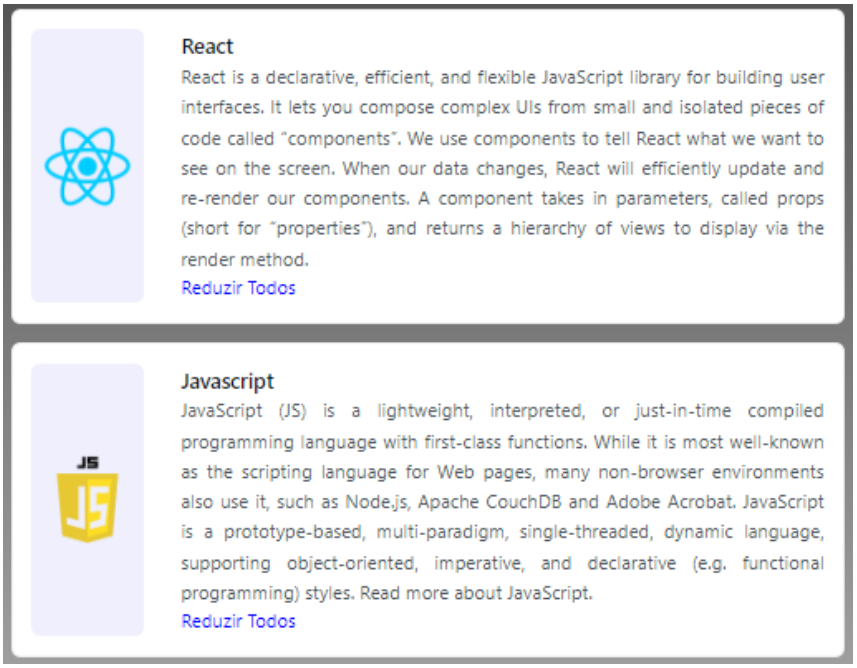

Figura 2 - Uso de variável de estado no elemento Pai

```
function AppExemplo() {

     const [allTextVisible, setAllTextVisible] = useState(false);

     const handleAllTextVisible = () => {
          if (allTextVisible) {
               console.log(allTextVisible)
               setAllTextVisible(false);
          } else {
               console.log(allTextVisible)
               setAllTextVisible(true);
          }
     };
     return (
          <div className="container">
               <InfoComponent title="React" src="./assets/images/react.png"
                              allTextVisible={allTextVisible}
                              onAllTextVisible={handleAllTextVisible}>

      React is a declarative,    ...
```

```
import React, { useState } from "react";
import "./exemplo.css";
function InfoComponent(props) {
  const { title, src, allVisible, onAllTextVisible } = props;
  const texto = props.children;
  const maxLength = 200;

  function getText() {  ...  }
  return (
    <>
      <div className="wrapper">
        <div className="logo">
          <img src={src} alt={title} />
        </div>
        <div className="text">
          <h2>{title}</h2>
          <p>{getText()}</p>
          {!allVisible ? (
            <a onClick={onAllTextVisible}> Expandir Todos </a>
          ) : (
            <a onClick={onAllTextVisible}> Reduzir Todos </a>
          )}
        </div>
      </div>
    </>
  );
}
export default InfoComponent;
```

› *Preparação do ambiente*

a. Efetue o download e descompacte o ficheiro **ficha9.zip** disponível no *inforestudante*.

NOTA: Os alunos que desejarem devem continuar a resolução da ficha 8, iniciada na última aula.

b. Inicie o *Visual Studio Code* e abra **a pasta no *workspace***.

c. No terminal, digite os seguintes comandos:

- → **`npm install`** (apenas quem usa ficha9.zip)
- → **`npm start`** (para iniciar a aplicação)

d. Visualize a página no browser, endereço http://localhost:3000/, que deverá ter o aspeto da figura 3.

Figura 3 – Estado Inicial da Aplicação

Parte II – Exercício: Variável de Estado e Eventos

1> Tendo em consideração o que foi apresentado anteriormente, efetue alterações no componente App e no componente ControPanel, de forma a que ao selecionar um nível, o botão **Iniciar Jogo** fique ativado e desativado, quando o nível voltar à posição "Selecione…".

Além disso, ao clicar em **Iniciar Jogo**, deverão ser apresentados os outros elementos que compõem o painel de controlo, como apresentação do tempo de jogo e pontuação (ainda que sem estarem realmente a funcionar), assim como o texto "Parar Jogo" no botão.

As figuras abaixo apresentam o comportamento desejado. ***Note que, se recorreu ao código fornecido em ficha9.zip as peças de jogo não serão apresentadas*** **(implemente essa parte no fim desta ficha, consultando a ficha prática 8).**

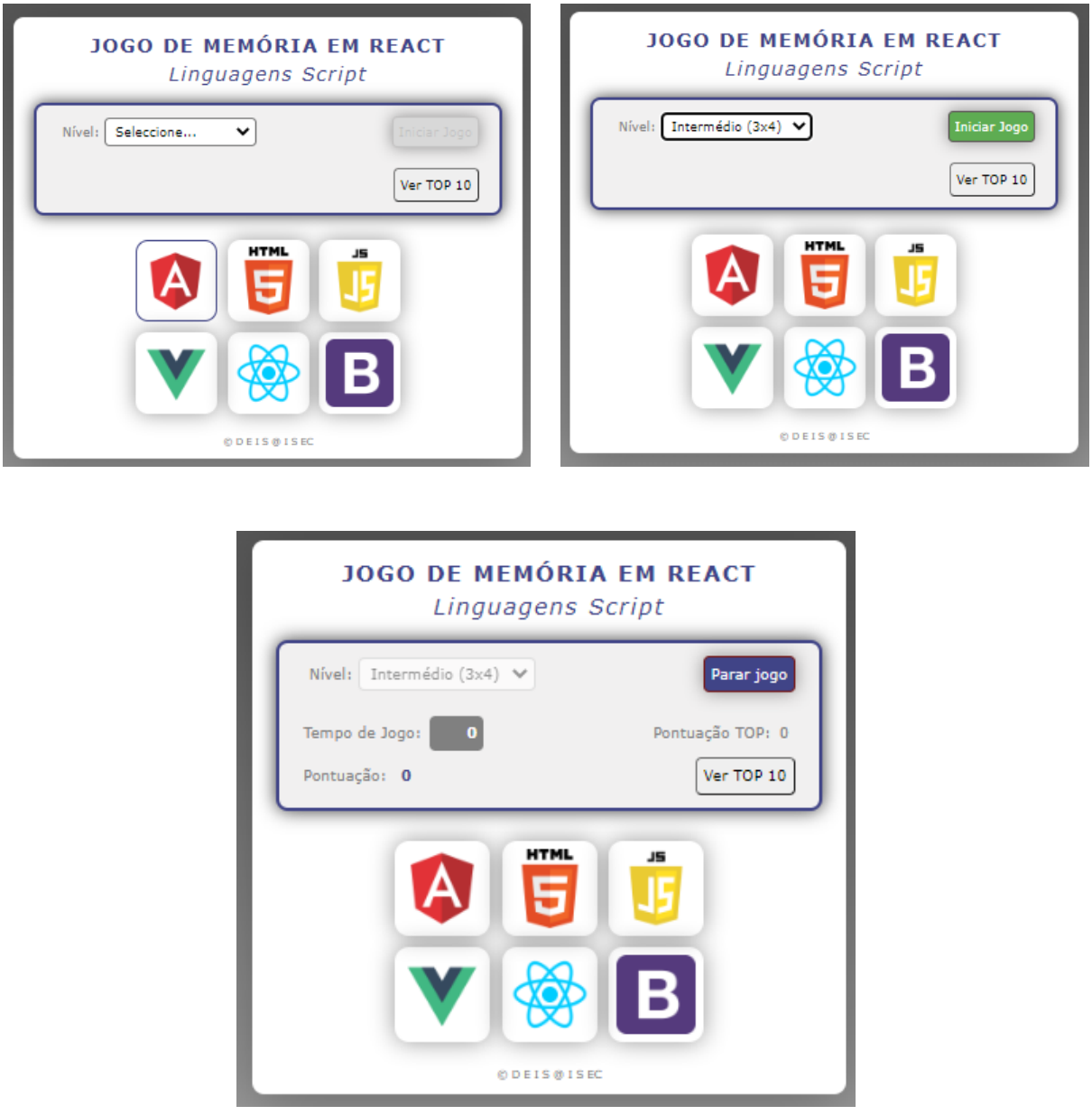

Figura 3 - Jogo inicial

a. No componente App, especifique os seguintes passos:

- Crie as seguintes variáveis de estado:
  - `gameStarted` (estado inicial deve ser false)
  - `selectedLevel` (estado inicial deve ser "0")
- Implemente a função `handleGameStart` que deve **alterar** o **valor da variável de estado** que gere o inicio do jogo. Se esta for ***true***, deve mudar para ***false*** e vice-versa.
- Implemente a função `handleLevelChange` que deve **alterar** o **valor da variável de estado** que gere o nível do jogo. Isto é, esta função, deverá armazenar o nível de jogo selecionado. *Nota:* A identificação do alvo atual para o evento pode ser especificada através da propriedade currentTarget, referenciado assim o elemento associado ao *event handler.*
- Altere a invocação do **ControlPanel**, especificando os atributos seguintes:
  - `gameStarted={gameStarted}`
  - `onGameStart={handleGameStart}`
  - `selectedLevel={selectedLevel}`
  - `onLevelChange={handleLevelChange}`

b. No componente ControlPanel (ficheiro **`control-panel.componente.jsx`**), especifique os seguintes passos:

- Receba as **props** do componente, com recurso à desestruturação de *arrays*, inicializando as seguintes variáveis:
  - `gameStarted`
  - `selectedLevel`
  - `onGameStart`
  - `onLevelChange`
- No componente `select`, especifique os atributos `disabled` e `onChange`, com os valores/referencias obtidas pelas props.
- No componente `button`, especifique os atributos `disabled` e `onClick`, com os valores/referencias obtidos pelas props. Além disso, especifique correctamente o código para que o texto do botão seja alterado para "Parar Jogo" ou "Iniciar Jogo" dependendo do estado do Jogo.

Por fim, adicione a class `gameStarted` aos elementos **dl** que envolvem o tempo de jogo, Pontuação TOP e Pontuação, apenas quando o jogo já tenha iniciado. Deverá efetuar isso, com o menor código possível.