Callback Handlers in JSX

Next we'll focus on the input field and label, by separating a standalone Search component and creating an instance of it in the App component. Through this process, the Search component becomes a sibling of the List component, and vice versa. We'll also move the handler and the state into the Search component to keep our functionality intact.

```
const App = () => {
  const stories = [ ... ];

  return (
    <div>
      <h1>My Hacker Stories</h1>

      <Search />


      <hr />

      <List list={stories} />
    </div>
  );
};

const Search = () => {
  const [searchTerm, setSearchTerm] = React.useState('');

  const handleChange = event => {
    setSearchTerm(event.target.value);
  };

  return (
    <div>
      <label htmlFor="search">Search: </label>
      <input id="search" type="text" onChange={handleChange} />

      <p>
        Searching for <strong>{searchTerm}</strong>.
      </p>
    </div>
  );
};
```

src/App.js

We have an extracted Search component that handles state and shows state without revealing its content. The component displays the `searchTerm` as text but doesn't share this information with its parent or sibling components yet. Since Search component does nothing except show the search term, it becomes useless for the other components.

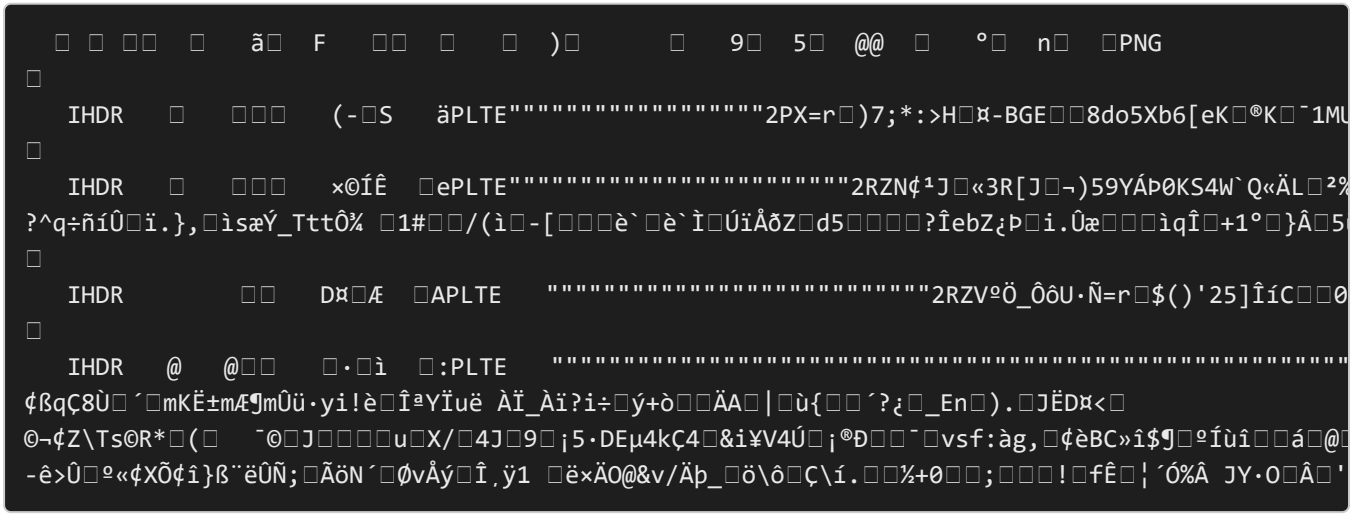
There is no way to pass information as JavaScript data types up the component tree, since props are naturally only passed downwards. However, we can introduce a **callback handler** as a function: A callback function gets introduced (A), is used elsewhere (B), but "calls back" to the place it was introduced ©.

```
const App = () => {
  const stories = [ ... ];

  // A
  const handleSearch = event => {
    // C
    console.log(event.target.value);
  };


  return (
    <div>
      <h1>My Hacker Stories</h1>

      <Search onSearch={handleSearch} />
      <hr />

      <List list={stories} />
    </div>
  );
};

const Search = props => {

  const [searchTerm, setSearchTerm] = React.useState('');

  const handleChange = event => {
    setSearchTerm(event.target.value);

    // B
    props.onSearch(event);

  };

  return ( ... );
};
```

src/App.js

Use comments in your source code to omit A, B, and C, as these are reminders which task each block of code is to perform. Consider the concept of the callback handler: We pass a function from one component (App) to another component (Search); we use it in the second component (Search); but use the actual callback of the function call in the first component (App). This way, we can communicate up the component tree. A handler function used in one component becomes a callback handler, which is passed down to components via React props. React props are always passed down as information to the component tree, and callback handlers passed as functions in props can be used to communicate up the component hierarchy.

Let's visualize this in the code below:



Exercises:

- Confirm the changes from the last section.
- Revisit the concepts of handler and callback handler as many times as you need.