## Lifting State in React

Currently, the Search component still has its internal state. While we established a callback handler to pass information up to the App component, we are not using it yet. We need to figure out how to share the Search component's state across multiple components.

The search term is needed in the App to filter the list before passing it to the List component as props. We'll need to **lift state up** from Search to App component to share the state with more components.

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

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

 const handleSearch = event => {

   setSearchTerm(event.target.value);

 };

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

     <Search onSearch={handleSearch} />

     <hr />

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

const Search = props => (
 <div>
   <label htmlFor="search">Search: </label>

   <input id="search" type="text" onChange={props.onSearch} />
 </div>
);
```

src/App.js

We learned about the callback handler previously, because it helps us to keep an open communication channel from Search to App component. The Search component doesn't manage the state anymore, but only passes up the event to the App component after text is entered into the input field. You could also display the `searchTerm` again in the App component or Search component by passing it down as prop.

Always manage the state at a component where every component that's interested in it is one that either manages the state (using information directly from state) or a component below the managing component (using information from props). If a component below needs to update the state, pass a callback handler down to it (see Search component). If a component needs to use the state (e.g. displaying it), pass it down as props.

By managing the search feature state in the App component, we can finally filter the list with the stateful `searchTerm` before passing the `list` to the List component:

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

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

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

 const searchedStories = stories.filter(function(story) {
   return story.title.includes(searchTerm);
 });


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

     <Search onSearch={handleSearch} />

     <hr />

     <List list={searchedStories} />
   </div>
 );
};
```

src/App.js

Here, the JavaScript array's built-in filter function is used to create a new filtered array. The filter function takes a function as an argument, which accesses each item in the array and returns true or false. If the function returns true, meaning the condition is met, the item stays in the newly created array; if the function returns false, it's removed:

```
const words = [
 'spray',
 'limit',
 'elite',
 'exuberant',
 'destruction',
 'present'
];

const filteredWords = words.filter(function(word) {
 return word.length > 6;
});

console.log(filteredWords);
// ["exuberant", "destruction", "present"]
```

src/App.js

The filter function checks whether the `searchTerm` is present in our story item's title, but it's still too opinionated about the letter case. If we search for "react", there is no filtered "React" story in your rendered list. To fix this problem, we have to lower case the story's title and the `searchTerm`.

```
const App = () => {
 ...

 const searchedStories = stories.filter(function(story) {
   return story.title
     .toLowerCase()
     .includes(searchTerm.toLowerCase());
 });

 ...
};
```

src/App.js

Now you should be able to search for "eact", "React", or "react" and see one of two displayed stories. You have just added an interactive feature to your application.

The remaining section shows a couple of refactoring steps. We will be using the final refactored version in the end, so it makes sense to understand these steps and keep them. As learned before, we can make the function more concise using a JavaScript arrow function:

```
const App = () => {
  ...

  const searchedStories = stories.filter(story => {
    return story.title
      .toLowerCase()
      .includes(searchTerm.toLowerCase());
  });

  ...
};
```

src/App.js

In addition, we could turn the return statement into an immediate return, because no other task (business logic) happens before the return:

```
const App = () => {
  ...

  const searchedStories = stories.filter(story =>
    story.title.toLowerCase().includes(searchTerm.toLowerCase())
  );
  ...
};
```

src/App.js

That’s all to the refactoring steps of the inlined function for the filter function. There are many variations to it – and it’s not always simple to keep a good balance between readable and conciseness – however, I feel like keeping it concise whenever possible keeps it most of the time readable as well.

```
  □ □ □□  □   ã□  F   □□  □   □  )□      □   9□  5□  @@  □   °□  n□  □PNG
□
   IHDR   □   □□□   (-□S   äPLTE"""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU
□
   IHDR   □   □□□   ×©ÍÊ   □ePLTE"""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□ìqÎ□+1°□}Â□5
□
   IHDR       □□   D¤□Æ   □APLTE   """""""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0
```

```
□
   IHDR   @   @□□   □·□ì  □:PLTE   """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
¢ßqÇ8Ù□´□mKË±mÆ¶mÛü·yi!è□ÎªYÏuë ÀÏ_Àï?i÷□ý+ò□□ÄA□|□ù{□□´?¿□_En□).□JËD¤<□

©¬¢Z\Ts©R*□(□  ¯©□J□□□□u□X/□4J□9□¡5·DEµ4kÇ4□&i¥V4Ú□¡®Ð□□¯□vsf:àg,□¢èBC»î$¶□ºÍùî□□á□@□
-ê>Û□º«¢XÕ¢î}ß¨ëÛÑ;□ÃöN´□ØvÅý□Î¸ÿ1 □ë×ÄO@&v/Äþ_□ö\ô□Ç\í.□□½+0□□;□□□!□fÊ□¦´Ó%Â JY·O□Â□'
```

Now we can manipulate state in React, using the Search component's callback handler in the App component to update it. The current state is used as a filter for the list. With the callback handler, we used information from the Search component in the App component to update the shared state and indirectly in the List component for the filtered list.

## Exercises:

- Confirm the changes from the last section.