Inline Handler in JSX

Learn about inline handlers to help to execute the function right in the JSX.

We'll cover the following

- Exercises:

The list of stories we have so far is only an unstateful variable. We can filter the rendered list with the search feature, but the list itself stays intact if we remove the filter. The filter is just a temporary change through a third party, but we can't manipulate the real list yet.

To gain control over the list, make it stateful by using it as initial state in React's useState Hook. The returned values are the current state (`stories`) and the state updater function (`setStories`). We aren't using the custom `useSemiPersistentState` hook yet, because we don't want to open the browser with the cached list each time. Instead, we always want to start with the initial list.

```
const initialStories = [
  {
    title: 'React',
    ...
  },
  {
    title: 'Redux',
    ...
  },
];

const useSemiPersistentState = (key, initialState) => { ... };

const App = () => {
  const [searchTerm, setSearchTerm] = ...

  const [stories, setStories] = React.useState(initialStories);

  ...
};
```

src/App.js

The application behaves the same because the `stories`, now returned from `useState`, are still filtered into `searchedStories` and displayed in the List. Next `useState`, are still filtered into `searchedStories` and displayed in the List. Next we'll manipulate the list by removing an item from it:

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

  const [stories, setStories] = React.useState(initialStories);

  const handleRemoveStory = item => {
    const newStories = stories.filter(
      story => item.objectID !== story.objectID
    );

    setStories(newStories);
  };

  ...

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

      ...

      <hr />

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

src/App.js

The callback handler in the App component receives an item to be removed as an argument, and filters the current stories based on this information by removing all items that don't meet its condition(s). The returned stories are then set as new state, and the List component passes the function to its child component. It's not using this new information; it's just passing it on:

```
const List = ({ list, onRemoveItem }) =>

  list.map(item => (
    <Item
      key={item.objectID}
      item={item}

      onRemoveItem={onRemoveItem}

    />
  ));
```

src/App.js

Finally, we can use the incoming function in another handler in the Item component to pass the `item` to it. A button element is used to trigger the actual event:

```
const Item = ({ item, onRemoveItem }) => {
  const handleRemoveItem = () => {
    onRemoveItem(item);
  };
  return (
    <div>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
      <span>
        <button type="button" onClick={handleRemoveItem}>
          Dismiss
        </button>
      </span>
    </div>
  );
};
```

src/App.js

We could have passed only the item's `objectID`, since that's all we need in the App component's callback handler, but we aren't sure what information the handler might need later. It may need more than an identifier to remove an item. If we call the handler `onRemoveItem`, it should be the item being passed, not just its identifier.

We have made the list of stories stateful with React's useState Hook; passed the still searched stories down as props to the List component; and implemented a callback handler (`handleRemoveStory`) and handler (`handleRemoveItem`) to be used in their respective components. Since a handler is just a function, and in this case it doesn't return anything, we could remove the block body for it for the sake of completeness.

```
const Item = ({ item, onRemoveItem }) => {

  const handleRemoveItem = () =>
    onRemoveItem(item);
  ...
};
```

src/App.js

This change makes our source code less readable as we accumulate handlers in the function component. Sometimes I refactor handlers in a function component from an arrow function back to a normal function statement, just to make the component more explorable:

```
console.log('Hello World');
const Item = ({ item, onRemoveItem }) => {
  function handleRemoveItem() {
    onRemoveItem(item);
  }
  ...
};
```

src/App.js

In this lesson, we applied props, handlers, callback handlers, and state. Those are all lessons learned from before. Now we'll tackle **inline handlers**, which allow us to execute the function right in the JSX. There are two solutions using the incoming function in the Item component as an inline handler. First, using JavaScript's bind method:

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
    <span>
      <button type="button" onClick={onRemoveItem.bind(null, item)}>
        Dismiss
      </button>
    </span>
  </div>
);
```

src/App.js

Using JavaScript's bind method on a function allows us to bind arguments directly to that function that should be used when executing it. The bind method returns a new function with the bound argument attached.

The second and more popular solution is to use a wrapping arrow function, which allows us to sneak in arguments like `item`:

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
    <span>


      <button type="button" onClick={() => onRemoveItem(item)}>
        Dismiss
      </button>
    </span>
  </div>
);
```

src/App.js

This is a quick solution because sometimes we don't want to refactor a function component's concise function body back to a block body to define an appropriate handler between function signature and return statement. While this way is more concise than the others, it can also be more difficult to debug because JavaScript logic may be hidden in JSX. It becomes even more verbose if the wrapping arrow function encapsulates more than one line of implementation logic, by using a block body instead of a concise body. This should be avoided:

```
const Item = ({ item, onRemoveItem }) => (
  <div>
    ...
    <span>
      <button
        type="button"
        onClick={() => {
          // do something else

          // note: avoid using complex logic in JSX

          onRemoveItem(item);
        }}
      >
        Dismiss
      </button>
    </span>
  </div>
);
```

src/App.js

All three handler versions, two of which are inline and the normal handler, are acceptable. The non-inlined handler moves the implementation details into the function component's block body; the inline handler move the implementation details into the JSX.

```
  □ □ □□  □    ã□  F    □□  □    □  )□       □    9□   5□  @@  □    °□   n□   □PNG
□
   IHDR    □    □□□   (-□S   äPLTE"""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU9gs3S
□
```

```
   IHDR   □   □□□   ×©ÍÊ   □ePLTE"""""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%+-0JR
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□ìqÎ□+1°□}Â□5ù  ïçd
□

   IHDR       □□   D¤□Æ  □APLTE   """""""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0LS<o}X
□
   IHDR   @   @□□   □·□ì  □:PLTE   """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
¢ßqÇ8Ù□´□mKË±mÆ¶mÛü·yi!è□ÎªYÏuë ÀÏ_Àï?i÷□ý+ò□□ÄA□|□ù{□□´?¿□_En□).□JËD¤<□
©¬¢Z\Ts©R*□(□  ¯©□J□□□□u□X/□4J□9□¡5·DEµ4kÇ4□&i¥V4Ú□¡®Ð□□¯□vsf:àg,□¢èBC»î$¶□ºÍùî□□á□@□ô□I_
-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□'/Å]_□
```

Exercises:

- Confirm the changes from the last section.
- Review handlers, callback handlers, and inline handlers.