Spread Operator

We are deep diving in to immutability and some tools to help make changes immutably.

Remember, changing something immutably is making a copy of the original and then changing it. Trade-offs – Uses more memory. But this is how React works out which bits of the DOM to re-render when something changes.

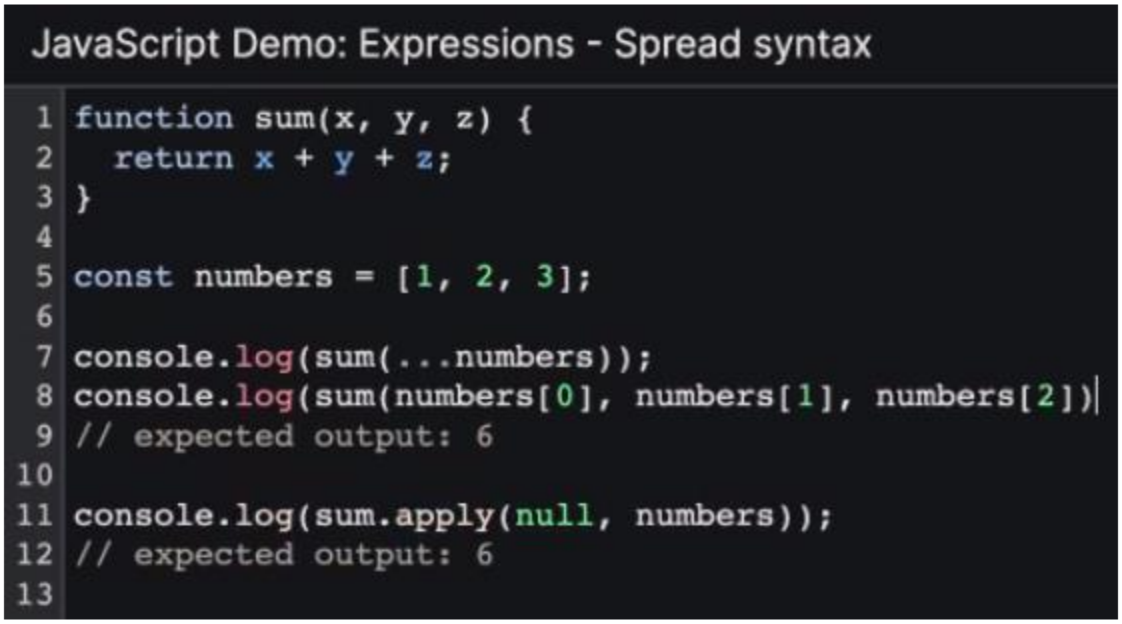
https://open.spotify.com/track/3IGj5Yh7HCiBn3cjAmbWPQ?si=ee5cea2822a74571
"Spread and slice makes things nice, and immutable"

When you're making toast, you don't just slam a block of butter on a slice. You spread it nicely and evenly across the toast. Spread operator is similar to this.

Spread operator is three dots: …

Read the docs! Simple: https://www.w3schools.com/react/react_es6_spread.asp
Complex: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
Useful explanation: https://medium.com/coding-at-dawn/how-to-use-the-spread-operator-in-javascript-b9e4a8b06fab

Allows an iterable such as an array or string to be expanded. E.g. Saves writing out each number in array. Line 8 is the full written out version of line 7:

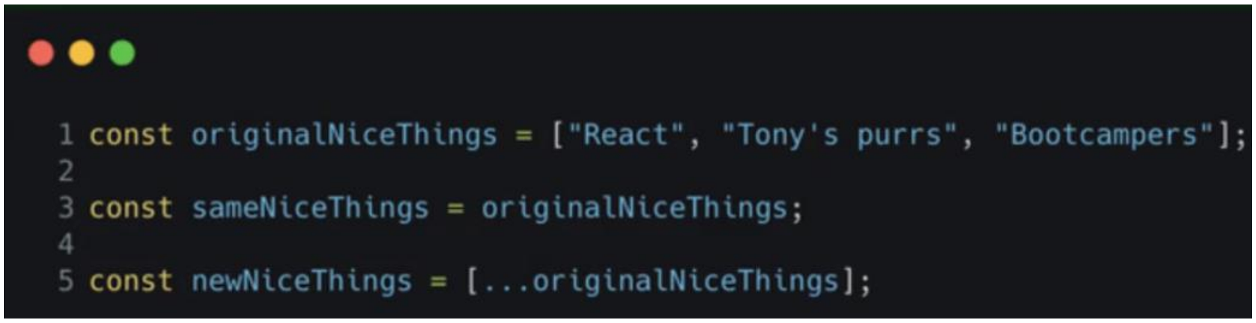


JavaScript Demo: Expressions - Spread syntax

```
 1 function sum(x, y, z) {
 2   return x + y + z;
 3 }
 4
 5 const numbers = [1, 2, 3];
 6
 7 console.log(sum(...numbers));
 8 console.log(sum(numbers[0], numbers[1], numbers[2])
 9 // expected output: 6
10
11 console.log(sum.apply(null, numbers));
12 // expected output: 6
13
```

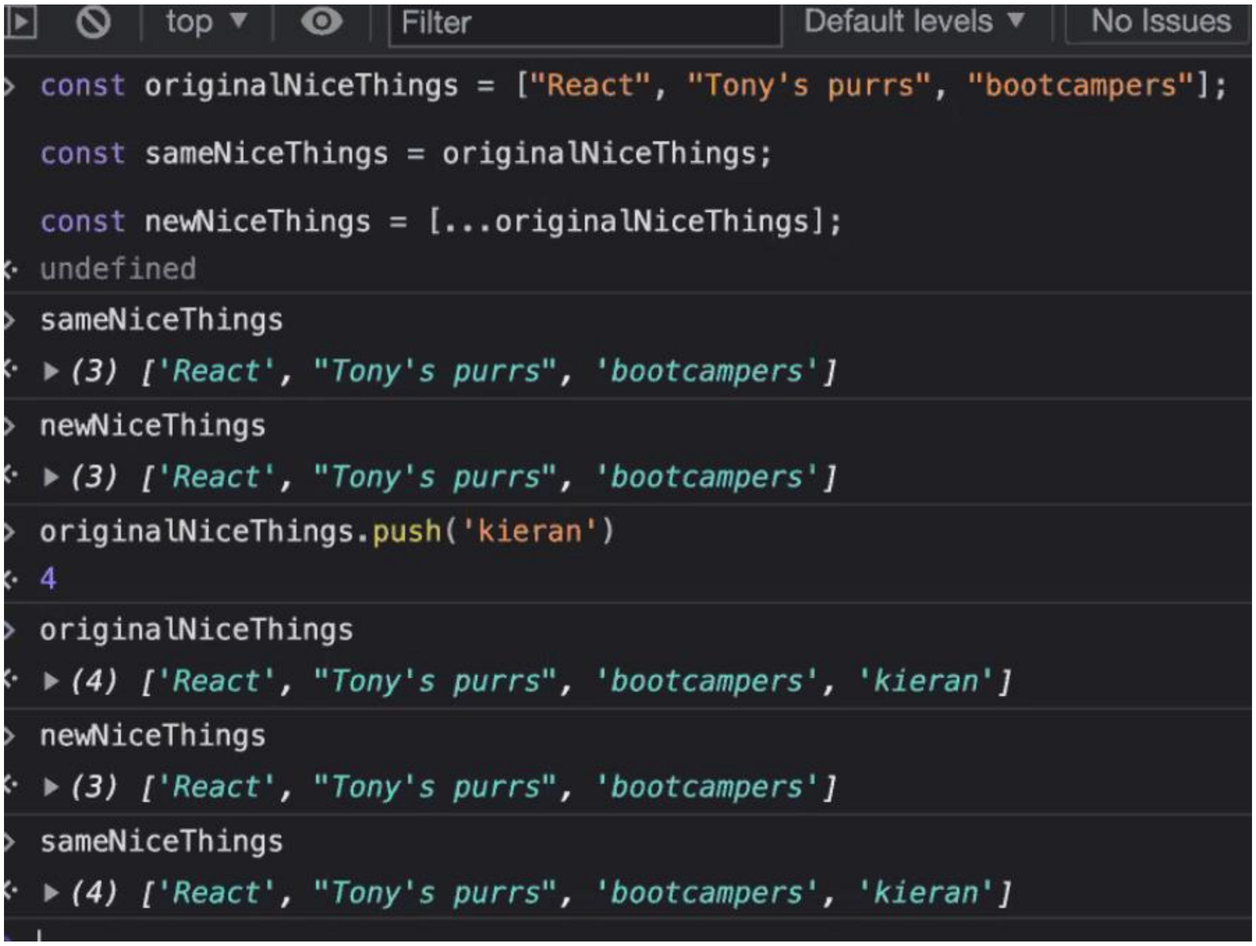
Basic use of spread with arrays:

If you did ,push to originalNiceThings,, it will also update sameNiceThings. Whereas newNiceThings on line 5 wouldn't change.
This is because line 3 is another "telescope" that also points at the same array as line 1, so same would happen if change sameNiceThings – it would change originalNiceThings too! (thinking back to universe analogy from week 1).
So, line 5 is how to change (or copy) an array immutably – it stores it in a new place in memory.



```
1 const originalNiceThings = ["React", "Tony's purrs", "Bootcampers"];
2
3 const sameNiceThings = originalNiceThings;
4
5 const newNiceThings = [...originalNiceThings];
```

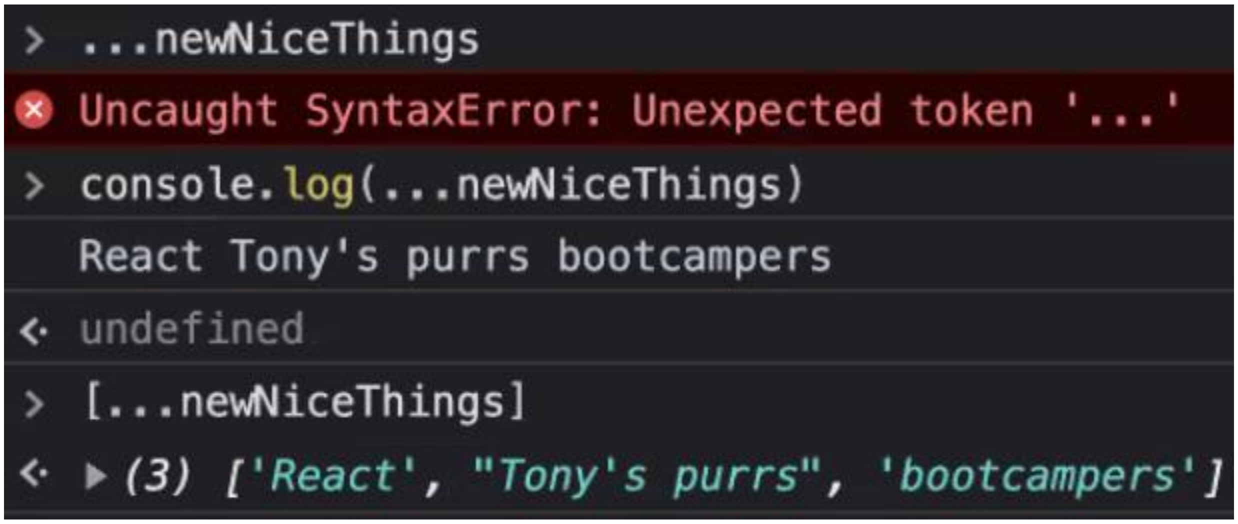
Example of above in action in the console:



```
> const originalNiceThings = ["React", "Tony's purrs", "bootcampers"];

  const sameNiceThings = originalNiceThings;

  const newNiceThings = [...originalNiceThings];
< undefined
> sameNiceThings
< ▶ (3) ['React', "Tony's purrs", 'bootcampers']
> newNiceThings
< ▶ (3) ['React', "Tony's purrs", 'bootcampers']
> originalNiceThings.push('kieran')
< 4
> originalNiceThings
< ▶ (4) ['React', "Tony's purrs", 'bootcampers', 'kieran']
> newNiceThings
< ▶ (3) ['React', "Tony's purrs", 'bootcampers']
> sameNiceThings
< ▶ (4) ['React', "Tony's purrs", 'bootcampers', 'kieran']
```

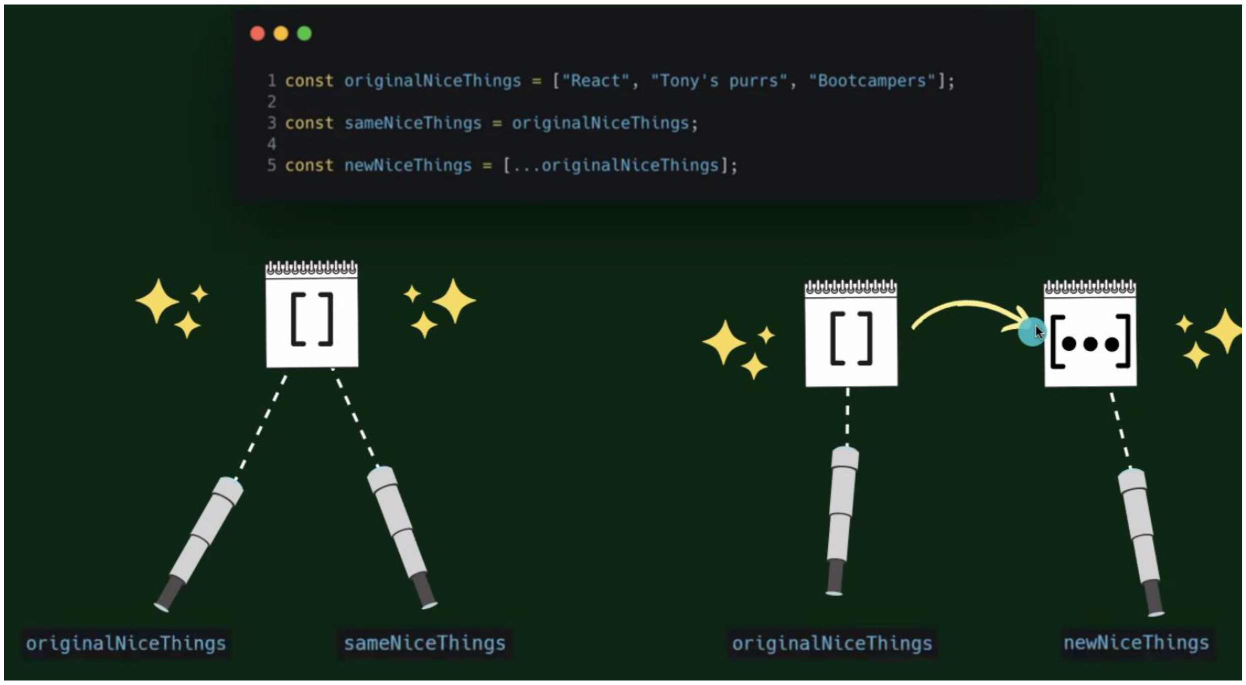
Original arrays remain unchanged by spreading them.

You can't just use …newNiceThings on it's own, it needs to go in square brackets to create a new array. If you were to console.log the spread, it would just give you the three strings concatenated:



```
> ...newNiceThings
✖ Uncaught SyntaxError: Unexpected token '...'
> console.log(...newNiceThings)
  React Tony's purrs bootcampers
<· undefined
> [...newNiceThings]
<· ▶(3) ['React', "Tony's purrs", 'bootcampers']
```

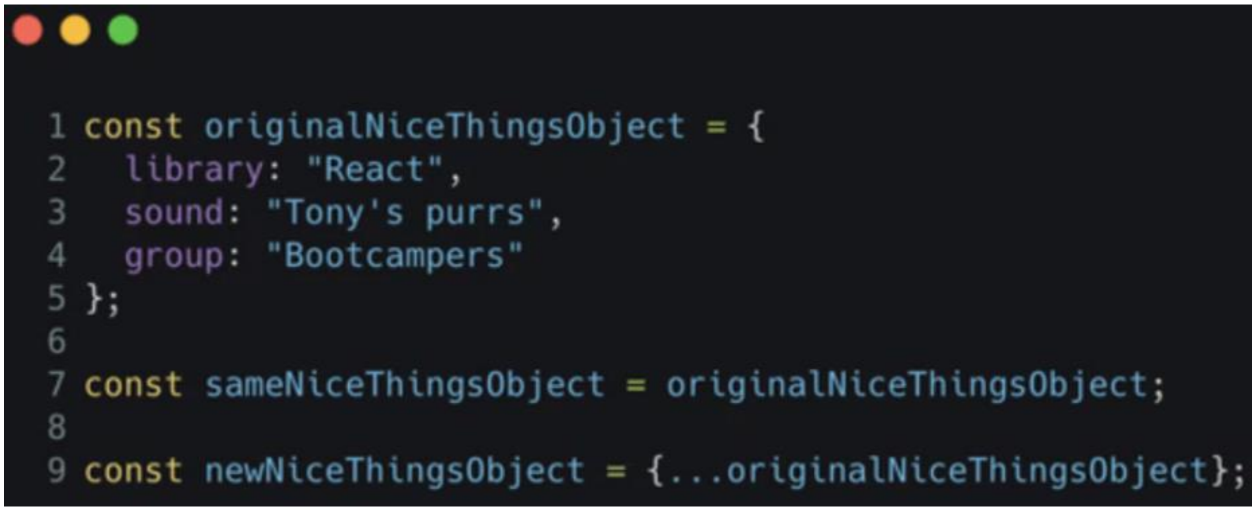
With telescope analogy:



```
const originalNiceThings = ["React", "Tony's purrs", "Bootcampers"];

const sameNiceThings = originalNiceThings;

const newNiceThings = [...originalNiceThings];
```

Basic use of spread with objects:

Very similar to Array example above but with an object

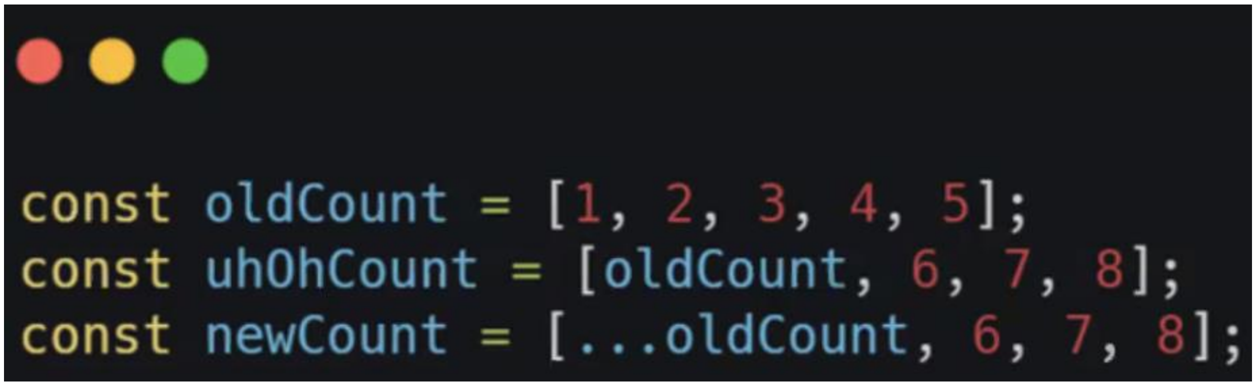


```
const originalNiceThingsObject = {
  library: "React",
  sound: "Tony's purrs",
  group: "Bootcampers"
};

const sameNiceThingsObject = originalNiceThingsObject;

const newNiceThingsObject = {...originalNiceThingsObject};
```

Works exactly the same as Arrays from earlier (although you can't console.log(…object) like you can with array!)

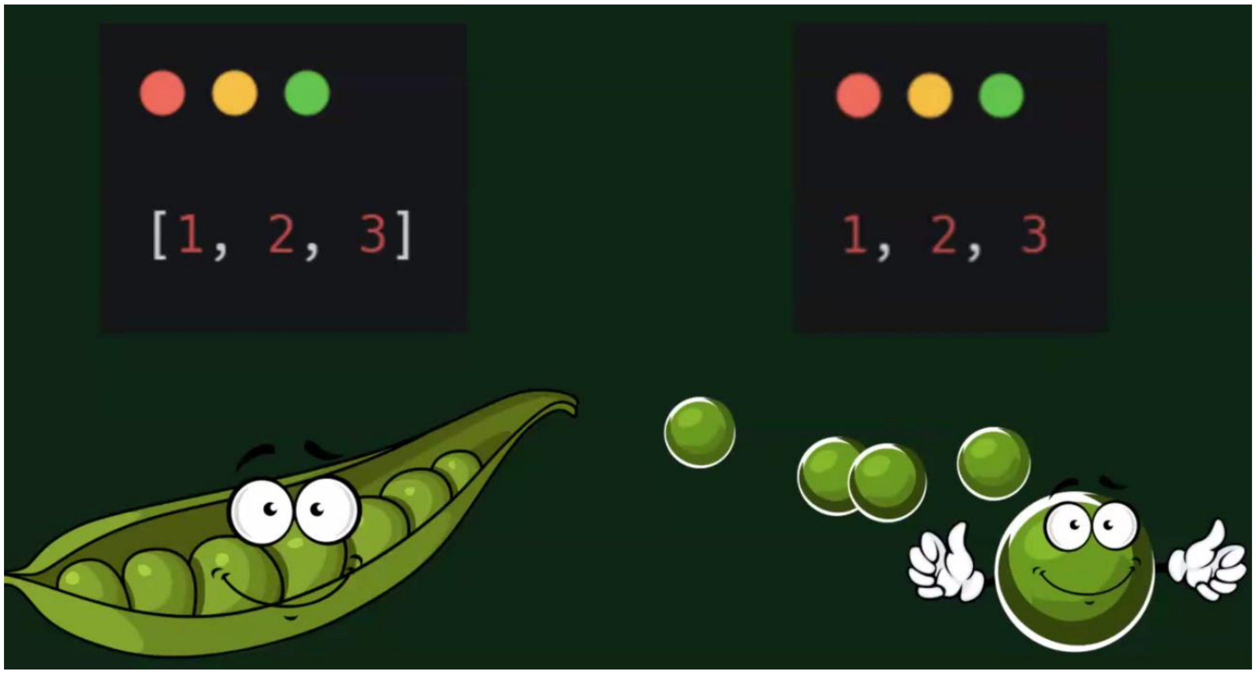
Adding to an array immutably:

uhOhCount doesn't combine the numbers into a brand new array properly (1st index is the oldCount array inside the uhOhCount array).
newCount is the correct way to get [1, 2, 3, 4, 5, 6, 7, 8]



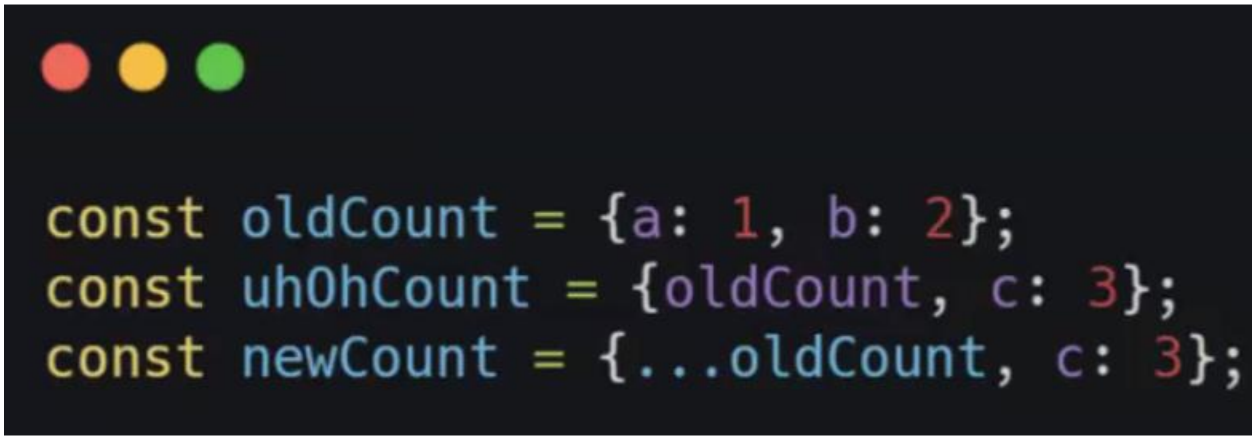
```
const oldCount = [1, 2, 3, 4, 5];
const uhOhCount = [oldCount, 6, 7, 8];
const newCount = [...oldCount, 6, 7, 8];
```

Think of the spread operator like popping peas out of a pod so you can use them individually (although original pod remains unchanged!):



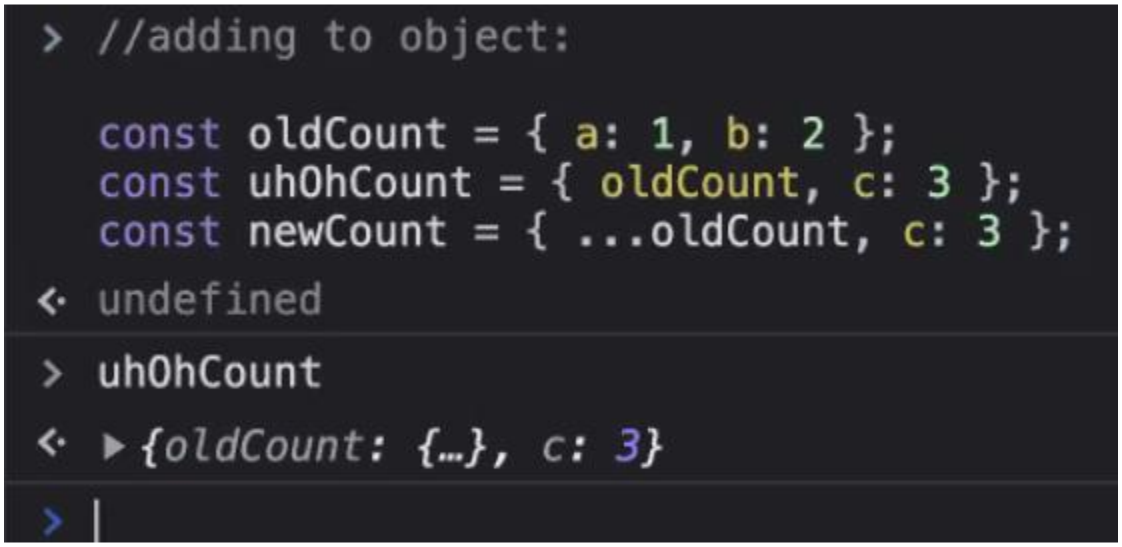
Adding to an object immutably:

Similar to array above, to combine 2 objects in to a single one, you must use the newCount version, not uhOhCount (which gives a nested object with a key of oldCount).



```
const oldCount = {a: 1, b: 2};
const uhOhCount = {oldCount, c: 3};
const newCount = {...oldCount, c: 3};
```

uhOhCount in the console:

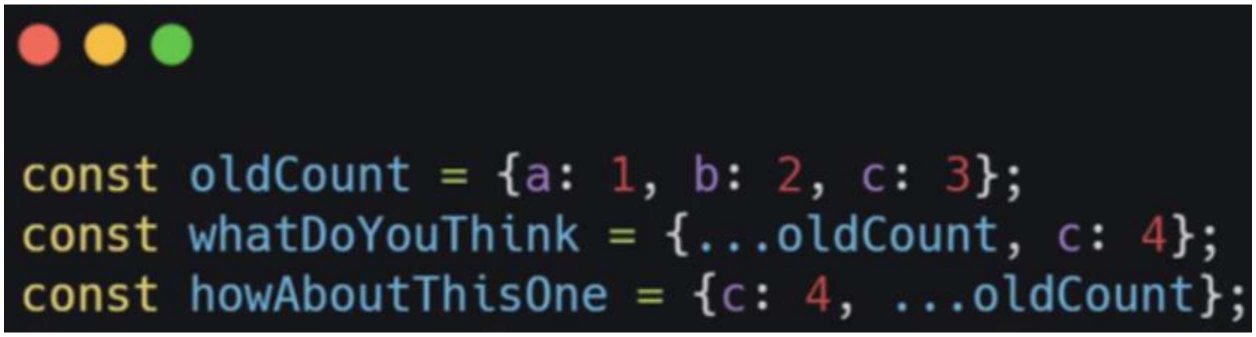


```
> //adding to object:

  const oldCount = { a: 1, b: 2 };
  const uhOhCount = { oldCount, c: 3 };
  const newCount = { ...oldCount, c: 3 };
<· undefined
> uhOhCount
<· ▶{oldCount: {…}, c: 3}
> |
```

What will happen here?

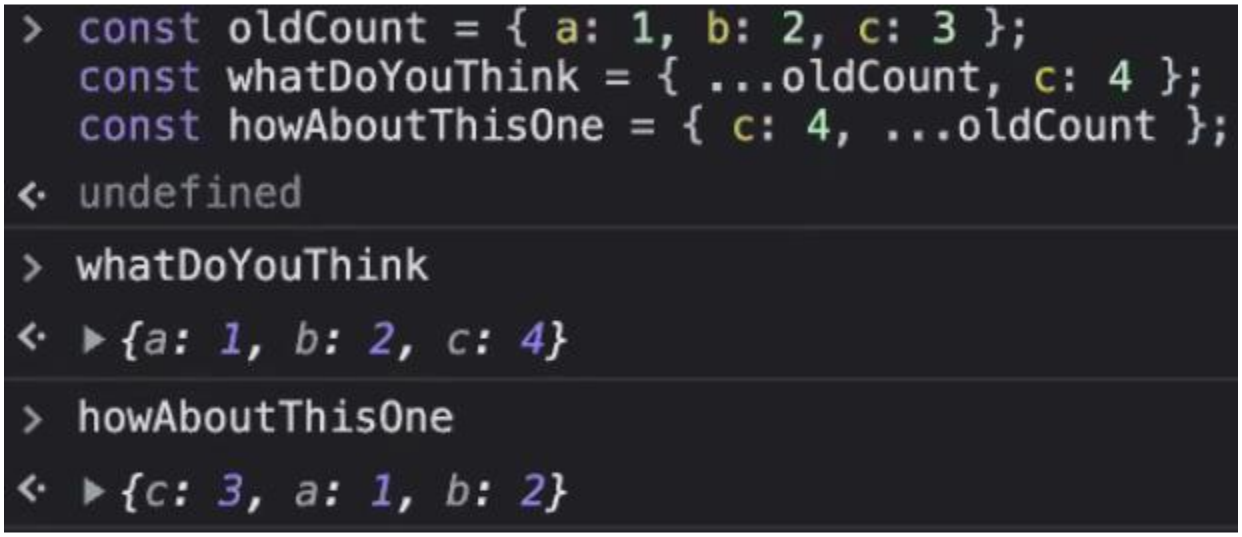
whatDoYouThink would update the value of the key “c” and update it to 4

howAboutThisOne would take the original value of “c” to be 4 but then override it with 3 (and maintain the order displayed! (but order doesn’t matter in an object due to key:value relationship instead of arrays having an index))



```
const oldCount = {a: 1, b: 2, c: 3};
const whatDoYouThink = {...oldCount, c: 4};
const howAboutThisOne = {c: 4, ...oldCount};
```

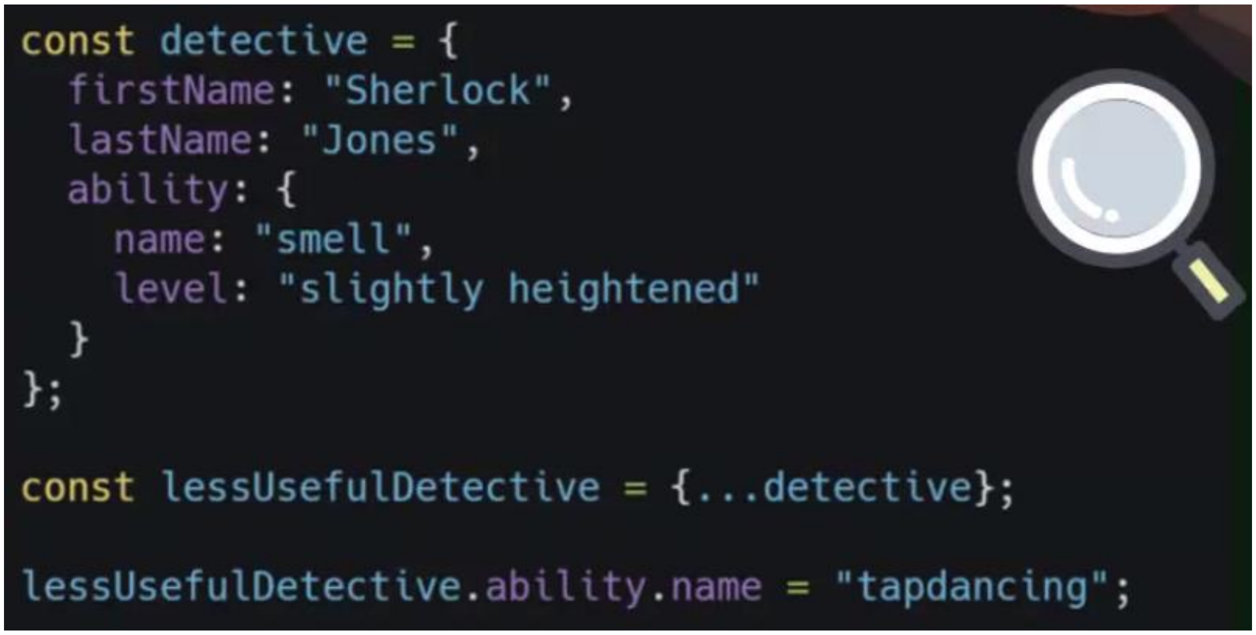
Proven in the console:



```
> const oldCount = { a: 1, b: 2, c: 3 };
  const whatDoYouThink = { ...oldCount, c: 4 };
  const howAboutThisOne = { c: 4, ...oldCount };
<· undefined
> whatDoYouThink
<· ▶{a: 1, b: 2, c: 4}
> howAboutThisOne
<· ▶{c: 3, a: 1, b: 2}
```

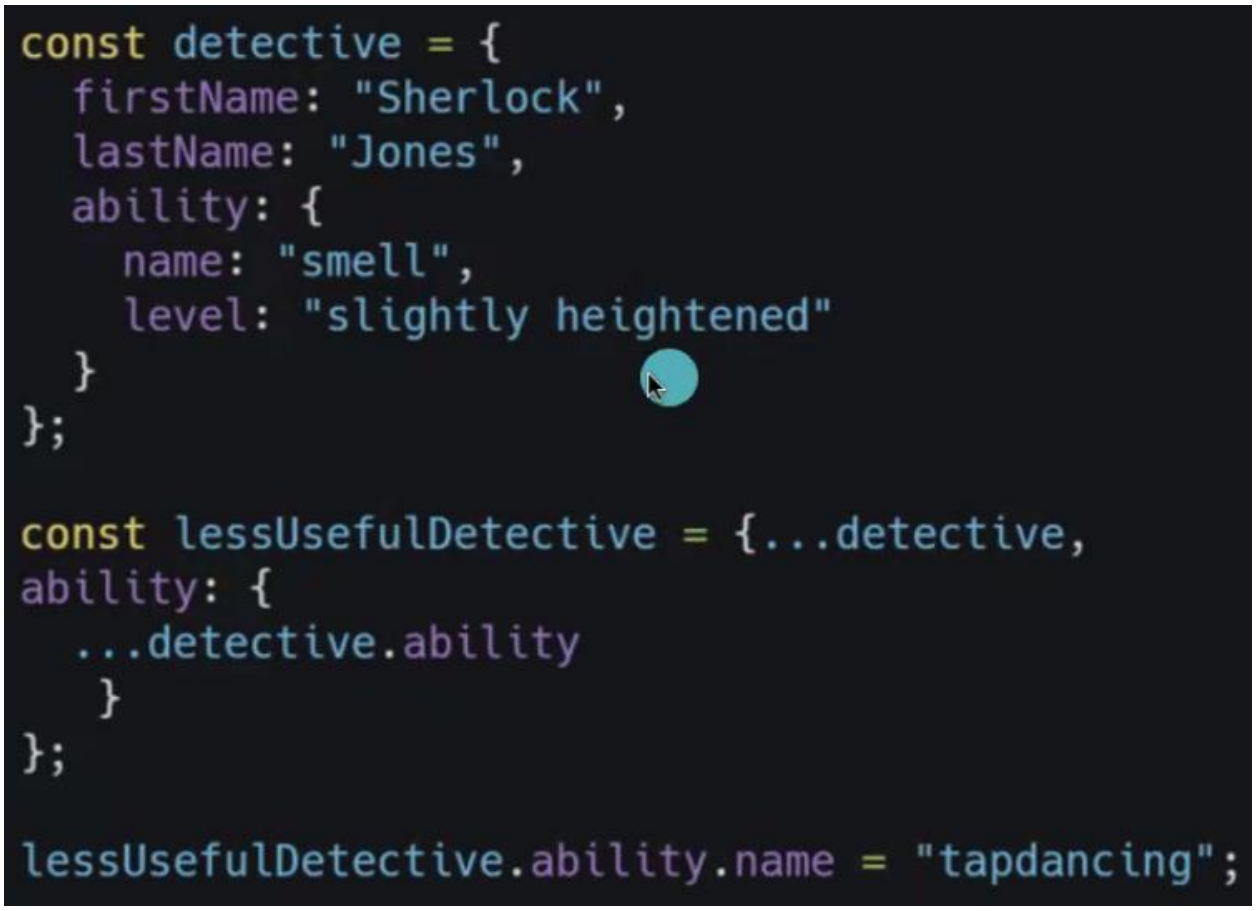
What do you think will happen part 2?

Nested objects and arrays have their own identity, so will get mutated in the below code:



```
const detective = {
  firstName: "Sherlock",
  lastName: "Jones",
  ability: {
    name: "smell",
    level: "slightly heightened"
  }
};

const lessUsefulDetective = {...detective};

lessUsefulDetective.ability.name = "tapdancing";
```

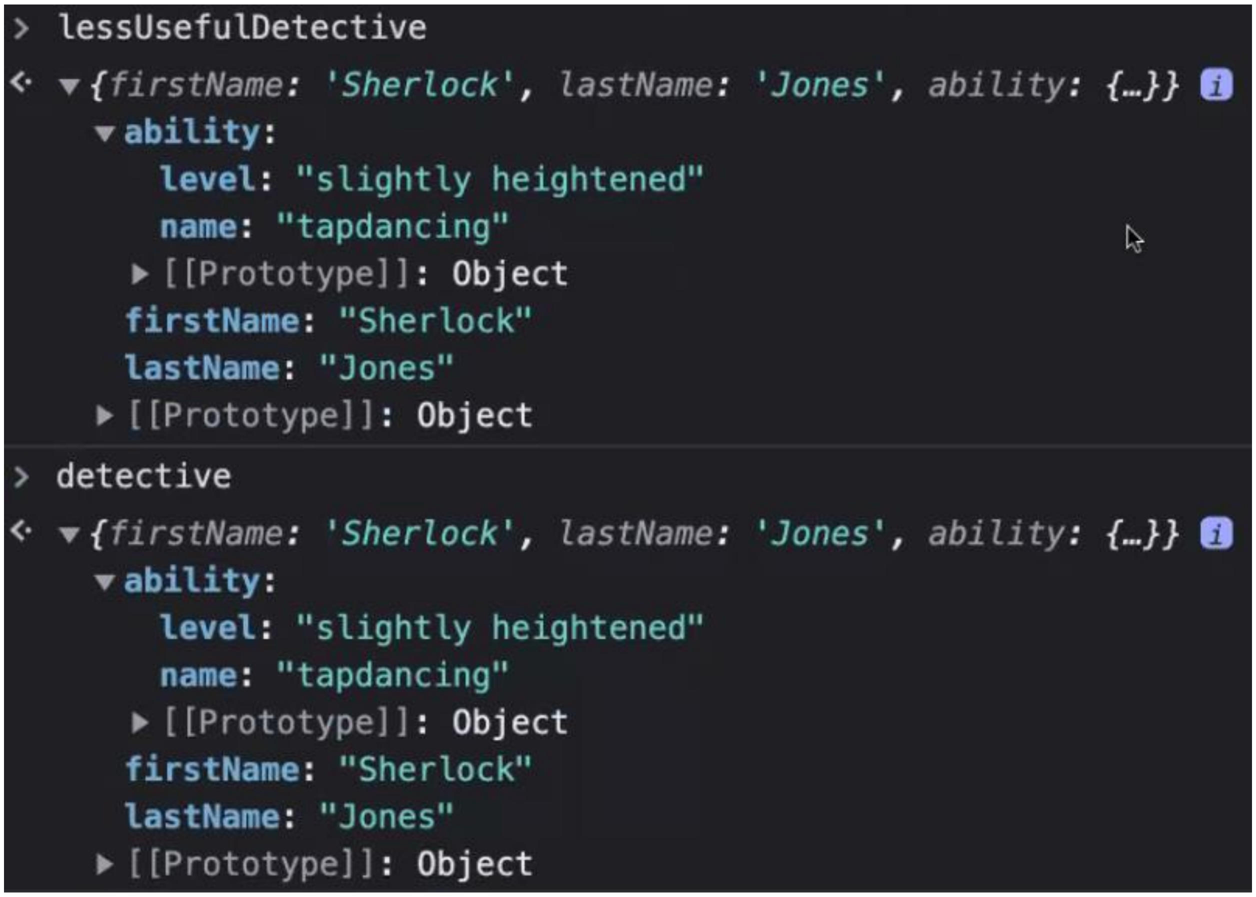
So, you need to spread the nested things as well:



```
const detective = {
  firstName: "Sherlock",
  lastName: "Jones",
  ability: {
    name: "smell",
    level: "slightly heightened"
  }
};

const lessUsefulDetective = {...detective,
ability: {
  ...detective.ability
   }
};

lessUsefulDetective.ability.name = "tapdancing";
```

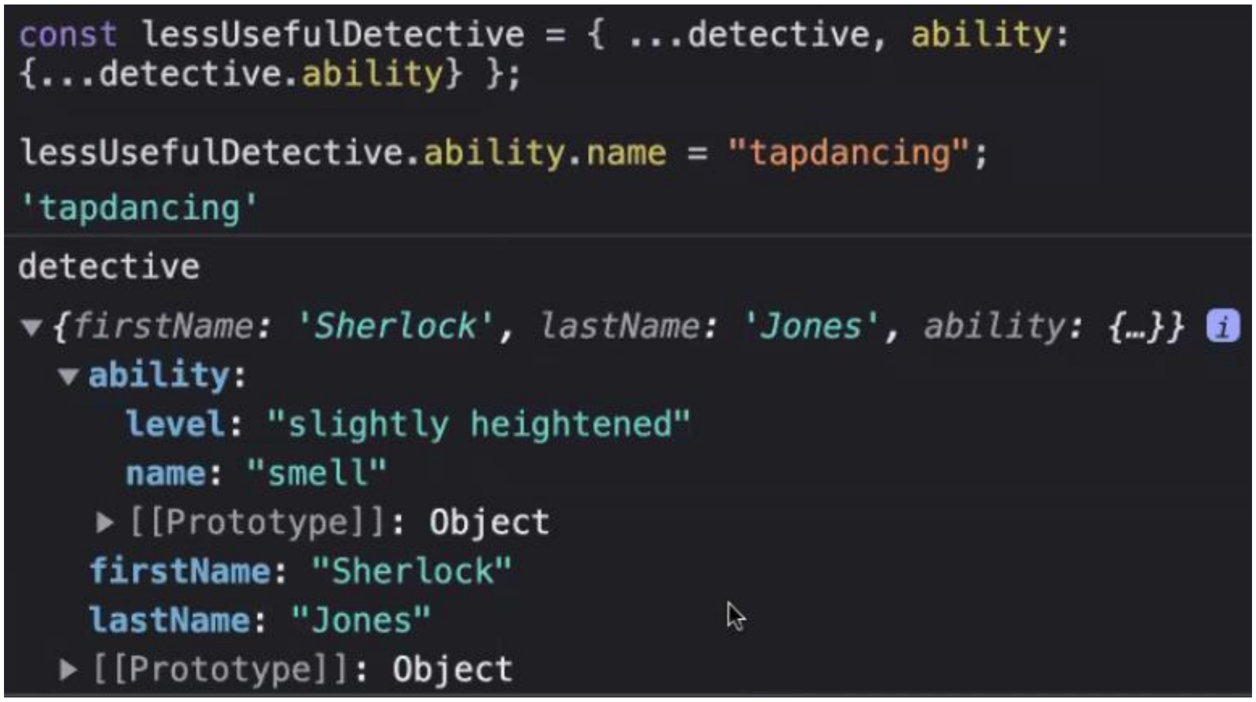
This is because "ability" is its own telescope inside the detective object. So spreading things out in this way is overriding the ability key with a spread of the .ability from the original object. So is being stored in a new place in memory (new box).

Proven in console:



```
> lessUsefulDetective
<· ▼{firstName: 'Sherlock', lastName: 'Jones', ability: {…}} i
    ▼ability:
       level: "slightly heightened"
       name: "tapdancing"
     ▶[[Prototype]]: Object
     firstName: "Sherlock"
     lastName: "Jones"
   ▶[[Prototype]]: Object
> detective
<· ▼{firstName: 'Sherlock', lastName: 'Jones', ability: {…}} i
    ▼ability:
       level: "slightly heightened"
       name: "tapdancing"
     ▶[[Prototype]]: Object
     firstName: "Sherlock"
     lastName: "Jones"
   ▶[[Prototype]]: Object
```

Spreading out correctly:



```
const lessUsefulDetective = { ...detective, ability:
{...detective.ability} };

lessUsefulDetective.ability.name = "tapdancing";
'tapdancing'
detective
▼{firstName: 'Sherlock', lastName: 'Jones', ability: {…}} i
  ▼ability:
     level: "slightly heightened"
     name: "smell"
   ▶[[Prototype]]: Object
   firstName: "Sherlock"
   lastName: "Jones"
 ▶[[Prototype]]: Object
```

Remember, spreading only makes a "shallow" copy (so top level key:values are copied, but nested things are being copied in their original form). Hence needing to spread each level!

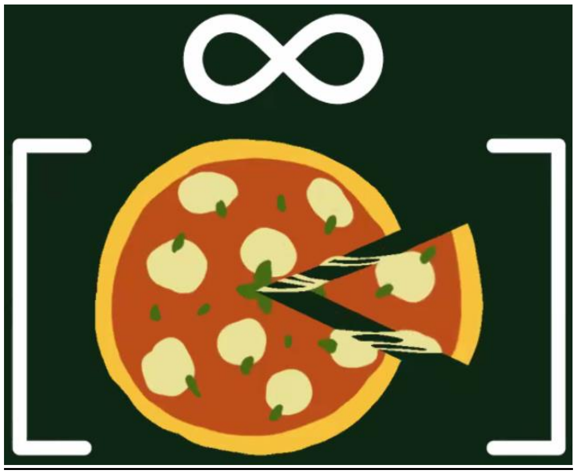
Slice

This is different to **splice**! Splice is mutable.

From MDN docs: The slice() method returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) where "start" and "end" represent the index of items in that array. The original array will not be modified.

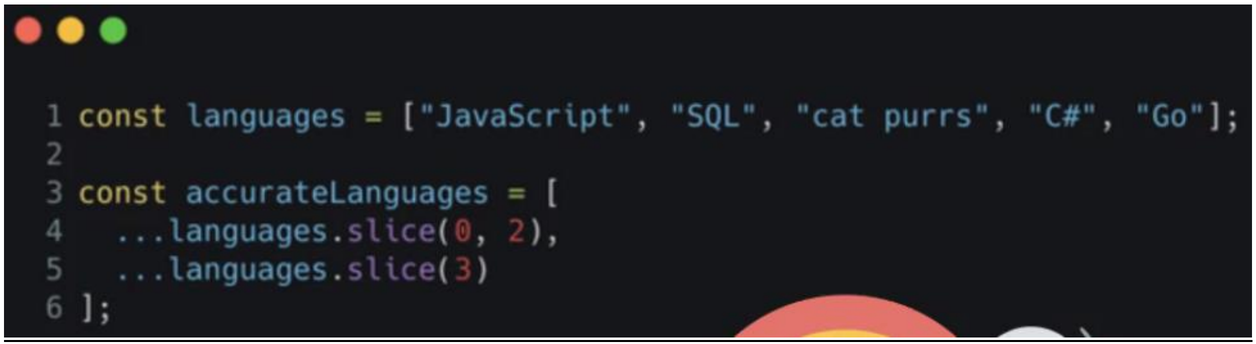
Great link to docs explaining: https://medium.com/geekculture/slice-in-javascript-2a42b32e23e7

Think of it like infinite pizza! Slice leaves the pizza there as well as giving you a slice!



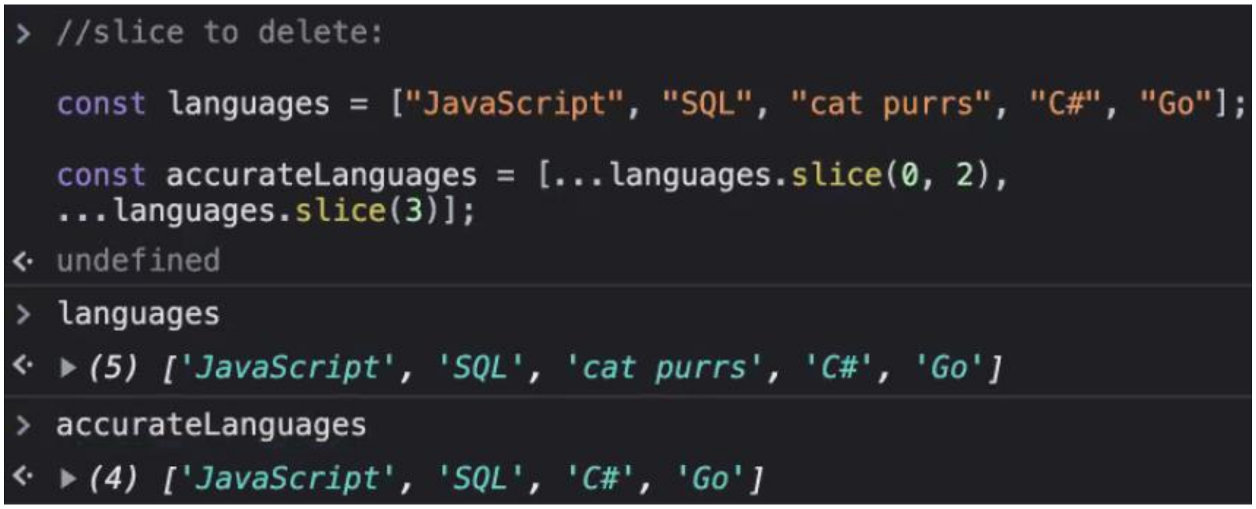
Deleting an item from an array immutably:

The below code is removing "cat purrs". Remember end index isn't included so even though index 2 is referenced, it's not included. If you don't provide a 2nd arg with the "end" index, it will go to end automatically. **N.B.** We still need to spread the part of the array we are slicing each time!



```
const languages = ["JavaScript", "SQL", "cat purrs", "C#", "Go"];

const accurateLanguages = [
  ...languages.slice(0, 2),
  ...languages.slice(3)
];
```

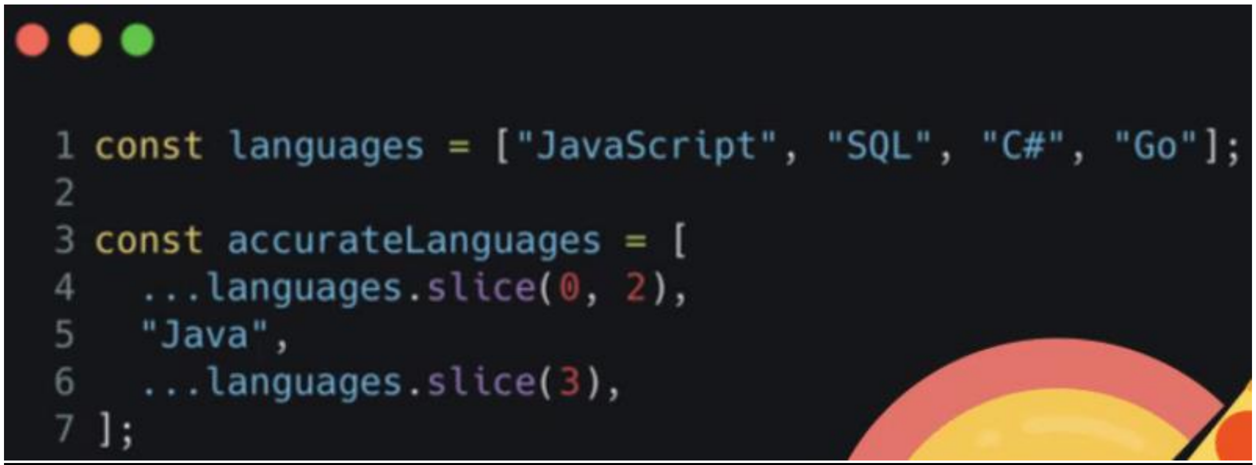
Proven in console:



```
> //slice to delete:

  const languages = ["JavaScript", "SQL", "cat purrs", "C#", "Go"];

  const accurateLanguages = [...languages.slice(0, 2),
  ...languages.slice(3)];
<· undefined
> languages
<· ▶ (5) ['JavaScript', 'SQL', 'cat purrs', 'C#', 'Go']
> accurateLanguages
<· ▶ (4) ['JavaScript', 'SQL', 'C#', 'Go']
```

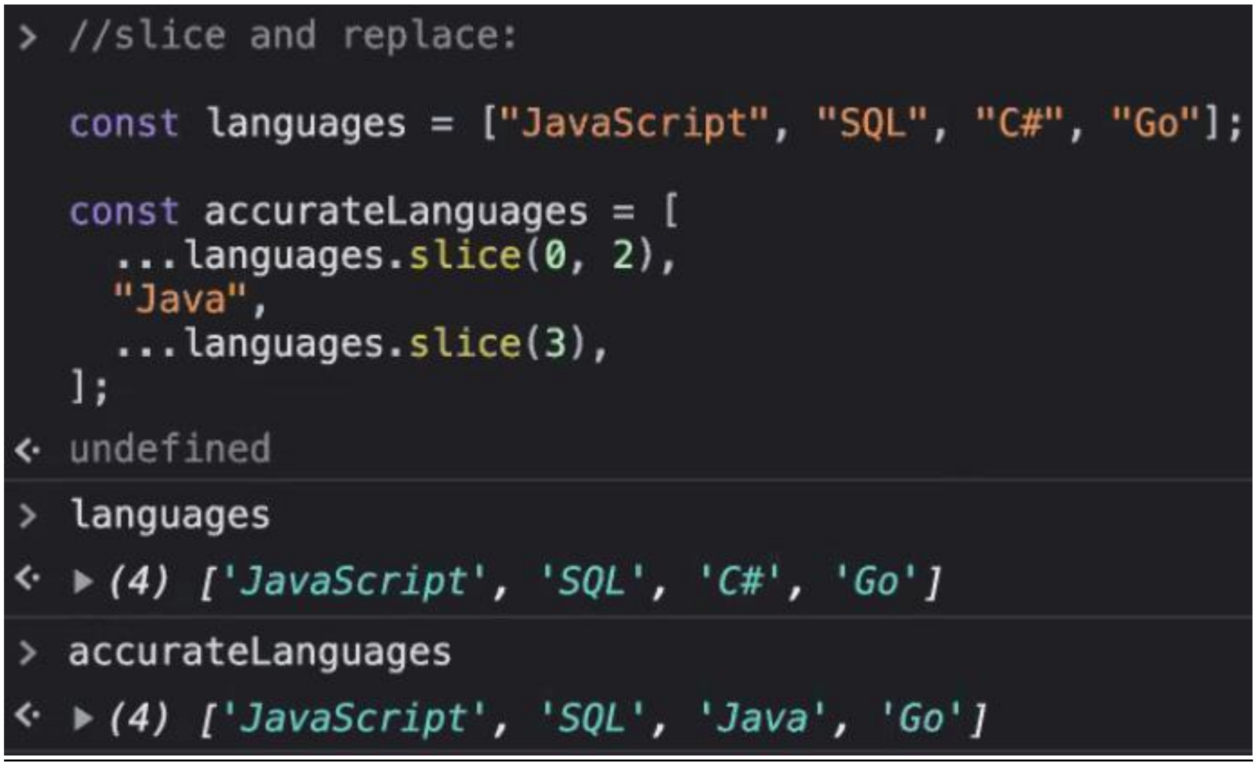
Changing an item from an array immutably:

Same code as above but removing "C#" and adds "Java" in its place on line 5:



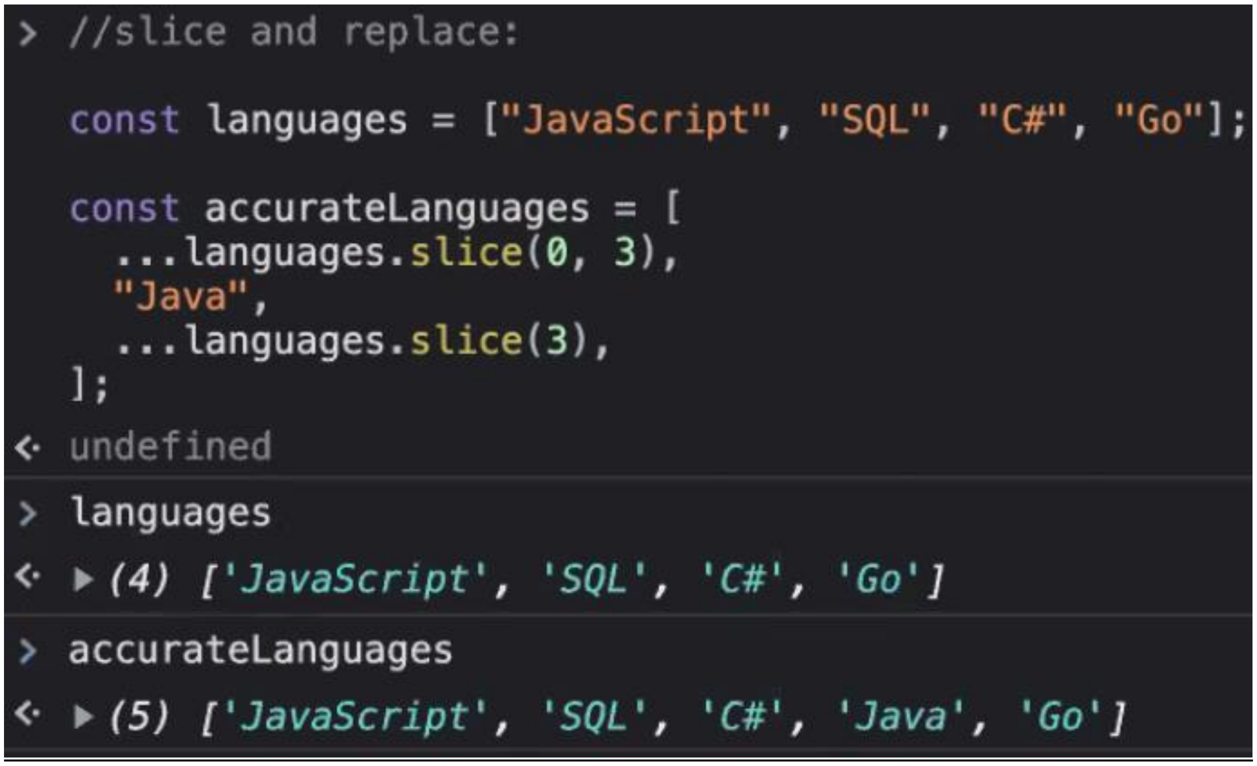
```
1 const languages = ["JavaScript", "SQL", "C#", "Go"];
2
3 const accurateLanguages = [
4   ...languages.slice(0, 2),
5   "Java",
6   ...languages.slice(3),
7 ];
```

In practice in the console:



```
> //slice and replace:

  const languages = ["JavaScript", "SQL", "C#", "Go"];

  const accurateLanguages = [
    ...languages.slice(0, 2),
    "Java",
    ...languages.slice(3),
  ];
<· undefined
> languages
<· ▶ (4) ['JavaScript', 'SQL', 'C#', 'Go']
> accurateLanguages
<· ▶ (4) ['JavaScript', 'SQL', 'Java', 'Go']
```

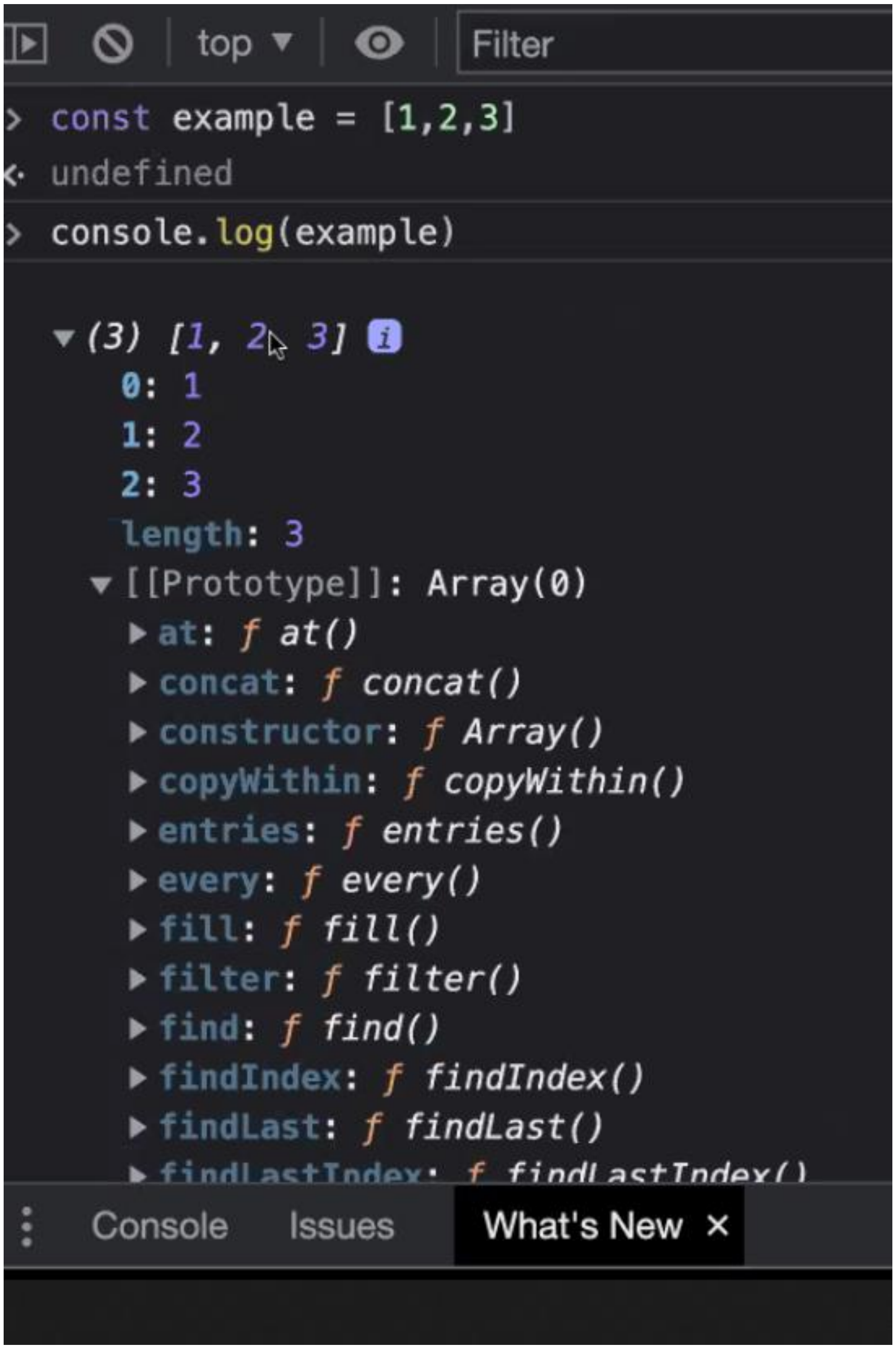
If you just want to add "Java" without removing anything, this code inserts it between "C#" and "Go":



```
> //slice and replace:

  const languages = ["JavaScript", "SQL", "C#", "Go"];

  const accurateLanguages = [
    ...languages.slice(0, 3),
    "Java",
    ...languages.slice(3),
  ];
<· undefined
> languages
<· ▶ (4) ['JavaScript', 'SQL', 'C#', 'Go']
> accurateLanguages
<· ▶ (5) ['JavaScript', 'SQL', 'C#', 'Java', 'Go']
```

<u>More tools for manipulating React components immutably</u>

Under the hood, arrays are a specific type of JavaScript object, so they have methods. (try creating an array in console of simple numbers then expand the "Prototype" bit!):

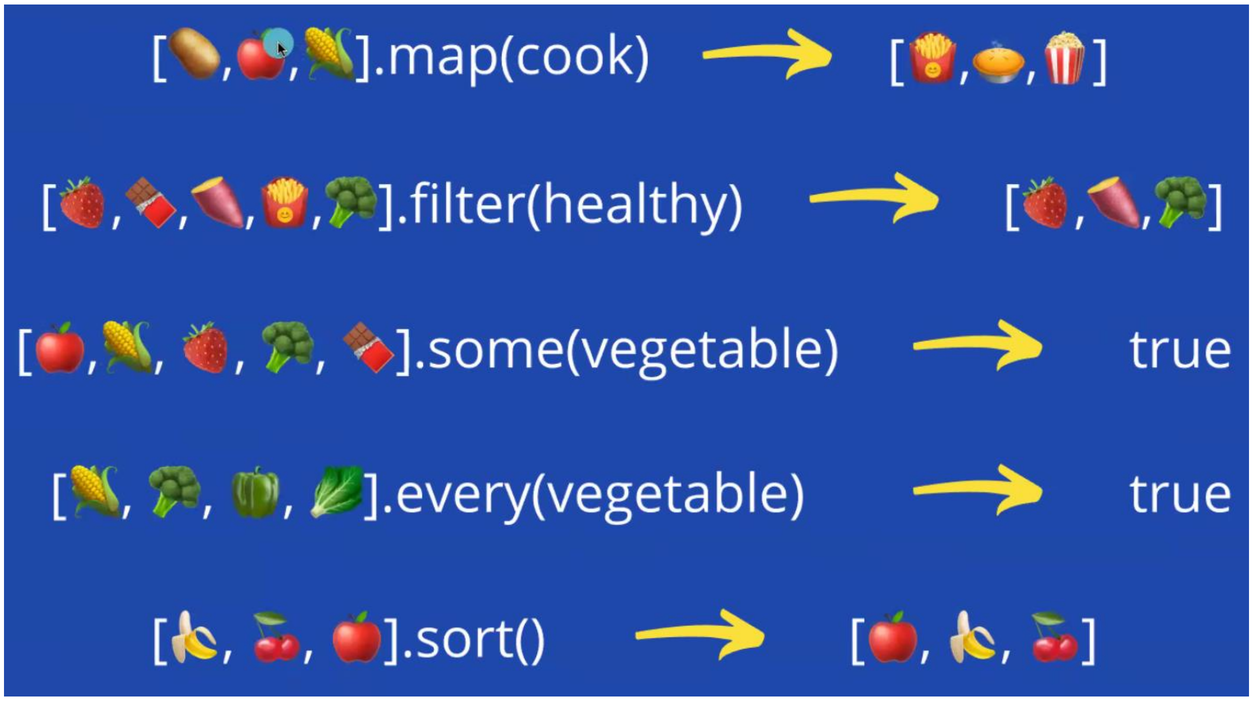


```
> const example = [1,2,3]
<· undefined
> console.log(example)

  ▼(3) [1, 2, 3] i
     0: 1
     1: 2
     2: 3
     length: 3
   ▼[[Prototype]]: Array(0)
     ▶at: ƒ at()
     ▶concat: ƒ concat()
     ▶constructor: ƒ Array()
     ▶copyWithin: ƒ copyWithin()
     ▶entries: ƒ entries()
     ▶every: ƒ every()
     ▶fill: ƒ fill()
     ▶filter: ƒ filter()
     ▶find: ƒ find()
     ▶findIndex: ƒ findIndex()
     ▶findLast: ƒ findLast()
     ▶findLastIndex: ƒ findLastIndex()
```

We're going to explore a few more and you can look up the rest!

Push and Pop, Shift and Unshift are all mutable. So we want to make a new copy of the array and do our changes there – using .map.

Pseudo code for some methods:


[🥔,🍎,🌽].map(cook) → [🍟,🥧,🍿]
[🍓,🍫,🍠,🍟,🥦].filter(healthy) → [🍓,🍠,🥦]
[🍎,🌽, 🍓, 🥦, 🍫].some(vegetable) → true
[🌽, 🥦, 🫑, 🥬].every(vegetable) → true
[🍌, 🍒, 🍎].sort() → [🍎, 🍌, 🍒]

.map takes in a call back function in () and visits each item in the array and calls the function on it. .map returns a new array, so this is immutable!

.filter is similar in that it takes in a callback function called "healthy" that does a test and returns a Boolean. .filter returns a new array too.

These are the most useful in React, but do check out the others on the slide and there are more than listed here too! But on a basic level:

.some – returns true if any of the items in the list are vegetables

.every() – returns true if every item in the list is a vegetable

.sort() – sorts the array based on the params in the ()

<u>.map method</u>

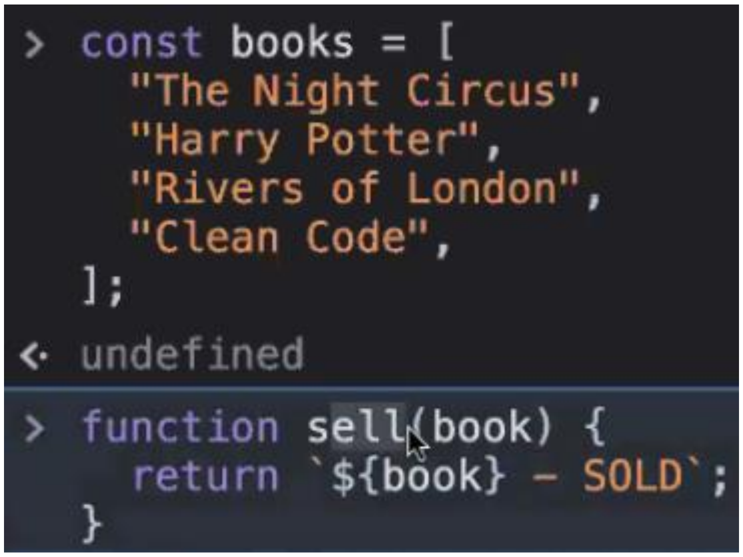
Here we have an array of books and a function that returns a string with the name of the book and marking it as SOLD:

We want to call function of sell on each item in the books array **immutably**.

We do this with the .map. The function you are passing in MUST take in an argument, because this argument references each item in the array (like in a for loop):

const soldBooks = books.map(function (book){
        return `${book} – SOLD`;

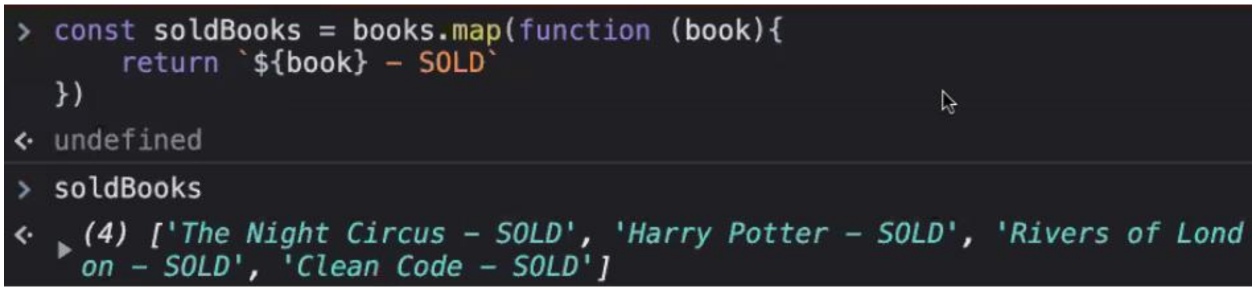
}



```
> const books = [
    "The Night Circus",
    "Harry Potter",
    "Rivers of London",
    "Clean Code",
  ];
<· undefined
> function sell(book) {
    return `${book} - SOLD`;
  }
```

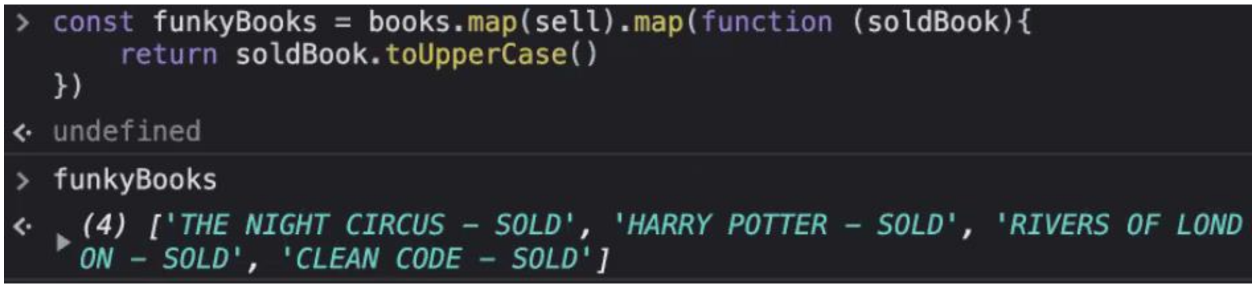
The above lines are just writing out the "sell" function that Liz declared, could also use:

const soldBooks = books.map(sell)



```
> const soldBooks = books.map(function (book){
      return `${book} - SOLD`
  })
<· undefined
> soldBooks
<· ▶ (4) ['The Night Circus - SOLD', 'Harry Potter - SOLD', 'Rivers of Lond
    on - SOLD', 'Clean Code - SOLD']
```

Can chain multiple methods on the end of .map (can be any other method, not just .map) as it returns an array. This code runs the sell function first (so adds – SOLD) then converts each item in the new array to uppercase:



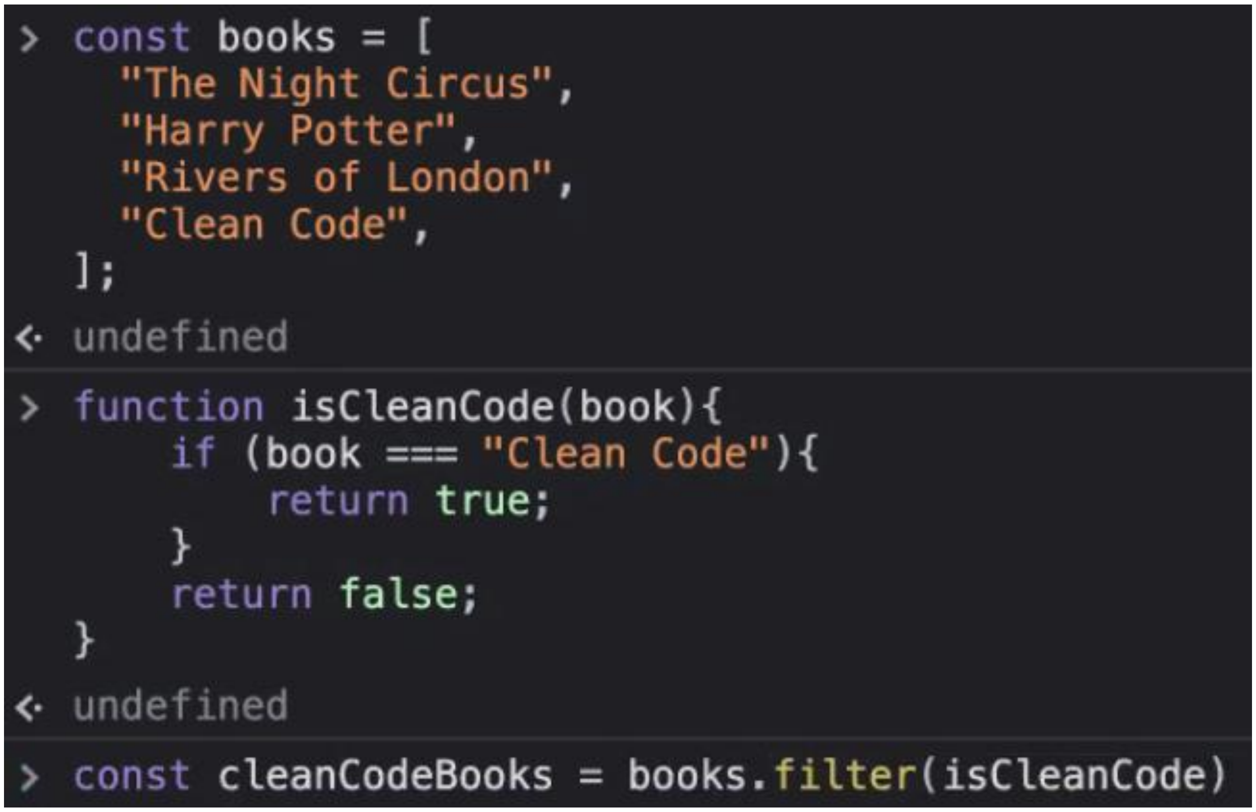
```
> const funkyBooks = books.map(sell).map(function (soldBook){
      return soldBook.toUpperCase()
  })
<· undefined
> funkyBooks
<· ▶ (4) ['THE NIGHT CIRCUS - SOLD', 'HARRY POTTER - SOLD', 'RIVERS OF LOND
    ON - SOLD', 'CLEAN CODE - SOLD']
```

Trade-offs!! You can achieve same functionality with a for loop. .map is an example of abstraction and is declarative, whereas a for loop is imperative code.

<u>Filter method</u>

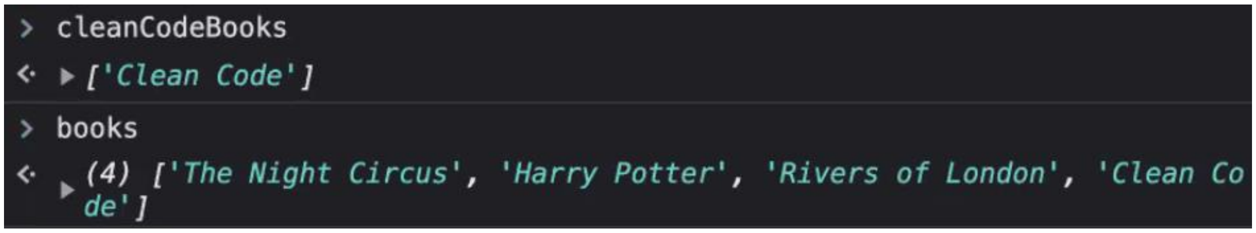
Same example of books. Remember to pass in the argument “book” as it refers to each item being iterated through.

N.B. Filter returns an item based on a Boolean! So the callback function “isCleanCode” is telling .filter whether to return or not



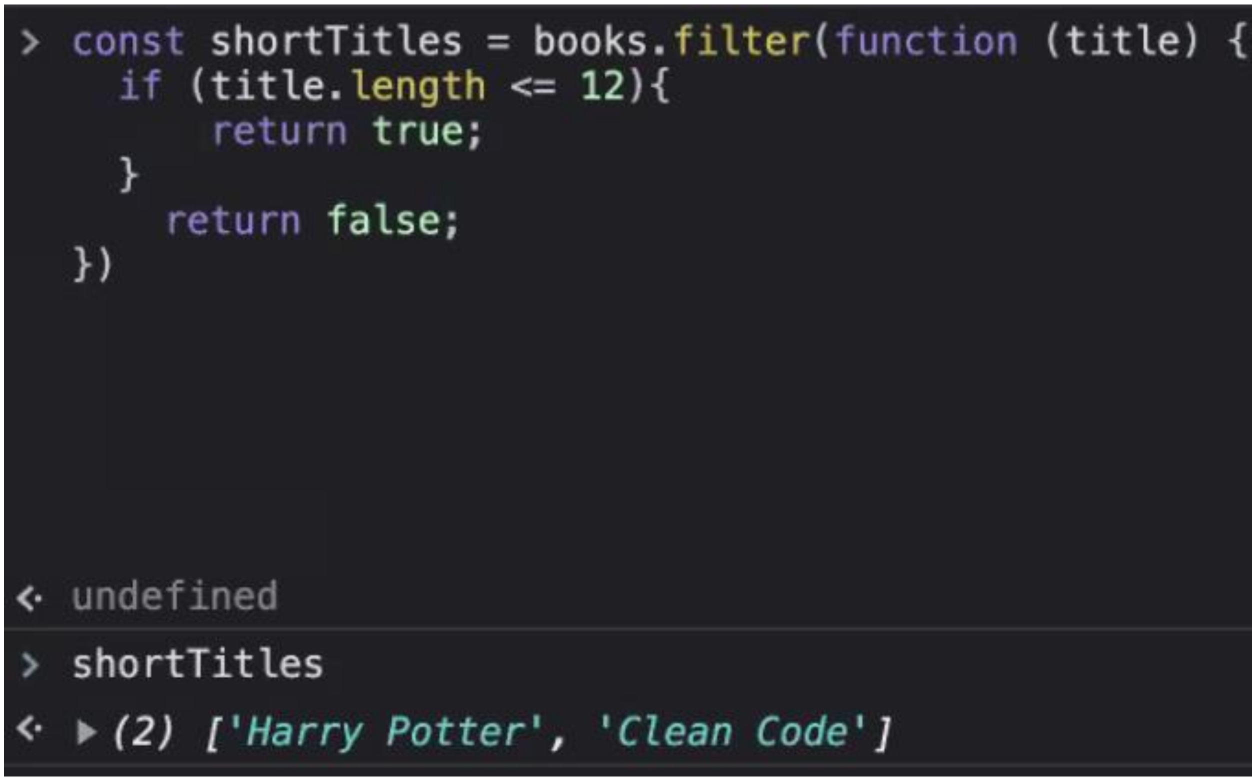
```
> const books = [
    "The Night Circus",
    "Harry Potter",
    "Rivers of London",
    "Clean Code",
  ];
<· undefined
> function isCleanCode(book){
      if (book === "Clean Code"){
          return true;
      }
      return false;
  }
<· undefined
> const cleanCodeBooks = books.filter(isCleanCode)
```

Console showing the new array containing Clean Code and original array unchanged – immutable!



```
> cleanCodeBooks
<· ▶ ['Clean Code']
> books
<· ▶ (4) ['The Night Circus', 'Harry Potter', 'Rivers of London', 'Clean Co
      de']
```

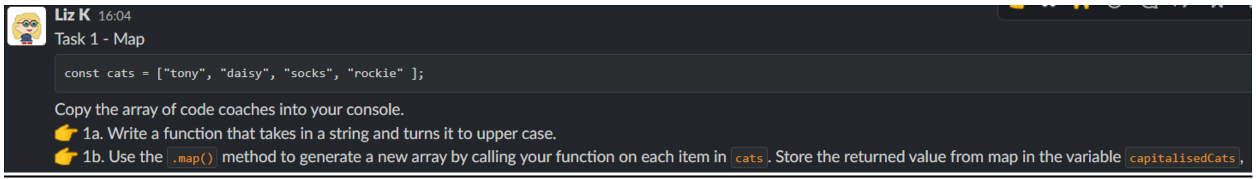
Can give it a callback function that returns any Boolean. Declared in-line this time and returns all books whose string length is less than or equal to 12:



```
> const shortTitles = books.filter(function (title) {
    if (title.length <= 12){
        return true;
    }
      return false;
  })
<· undefined
> shortTitles
<· ▶ (2) ['Harry Potter', 'Clean Code']
```

__Quick tasks to check understanding of the above__

(Cats, not code coaches)



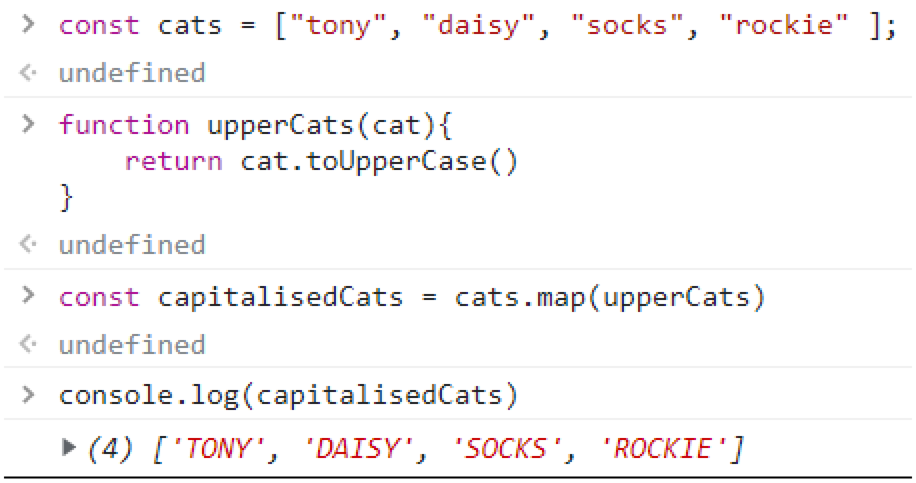
Liz K 16:04
Task 1 - Map

```
const cats = ["tony", "daisy", "socks", "rockie" ];
```

Copy the array of code coaches into your console.
👉 1a. Write a function that takes in a string and turns it to upper case.
👉 1b. Use the `.map()` method to generate a new array by calling your function on each item in `cats`. Store the returned value from map in the variable `capitalisedCats`,

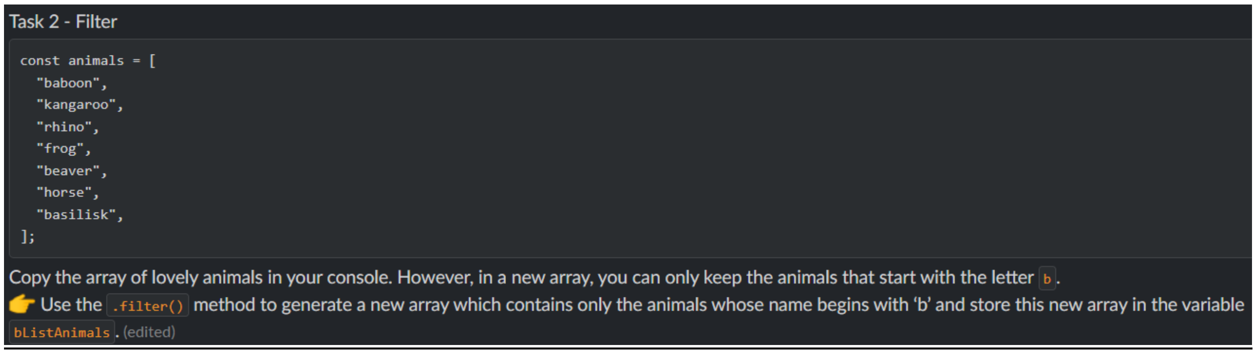
.map method

My code:



```
> const cats = ["tony", "daisy", "socks", "rockie" ];
<· undefined
> function upperCats(cat){
      return cat.toUpperCase()
  }
<· undefined
> const capitalisedCats = cats.map(upperCats)
<· undefined
> console.log(capitalisedCats)
  ▶ (4) ['TONY', 'DAISY', 'SOCKS', 'ROCKIE']
```

.filter method

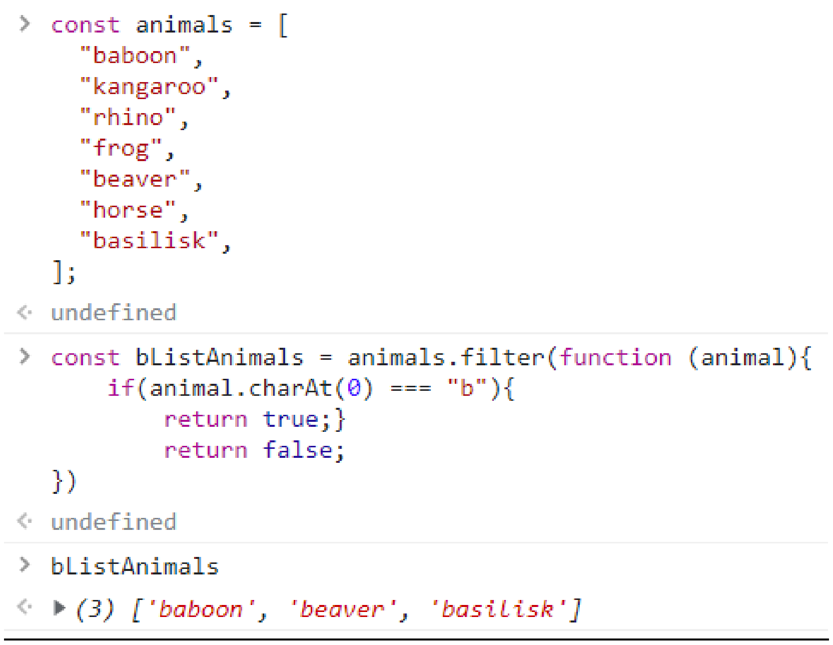


Task 2 - Filter

```
const animals = [
  "baboon",
  "kangaroo",
  "rhino",
  "frog",
  "beaver",
  "horse",
  "basilisk",
];
```

Copy the array of lovely animals in your console. However, in a new array, you can only keep the animals that start with the letter `b`.
👉 Use the `.filter()` method to generate a new array which contains only the animals whose name begins with ‘b’ and store this new array in the variable `bListAnimals`. (edited)

My code to solve:



```
> const animals = [
    "baboon",
    "kangaroo",
    "rhino",
    "frog",
    "beaver",
    "horse",
    "basilisk",
  ];
<· undefined
> const bListAnimals = animals.filter(function (animal){
      if(animal.charAt(0) === "b"){
          return true;}
          return false;
  })
<· undefined
> bListAnimals
<· ▶ (3) ['baboon', 'beaver', 'basilisk']
```

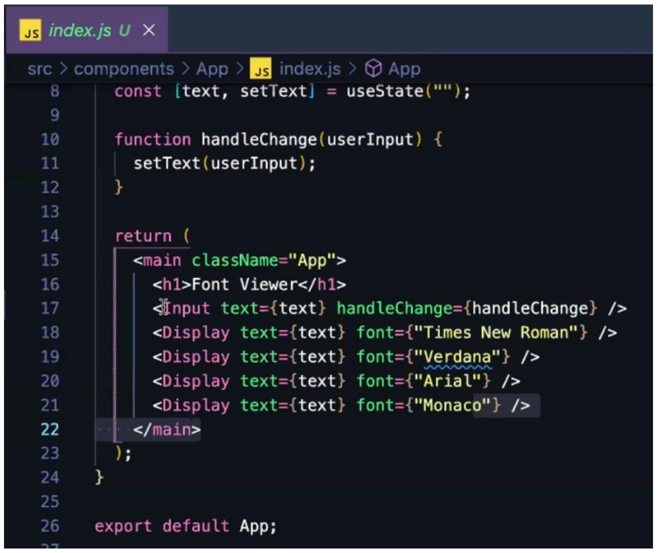
Linking back to React

If you google "Something", your search results come back as components – each result is a component and just passing in the correct data to each one!

When we make a call to APIs and we get data back as arrays we can use the above methods to update different components dynamically.

Liz showed us her version of completed code from yesterday's workshop:

App component:

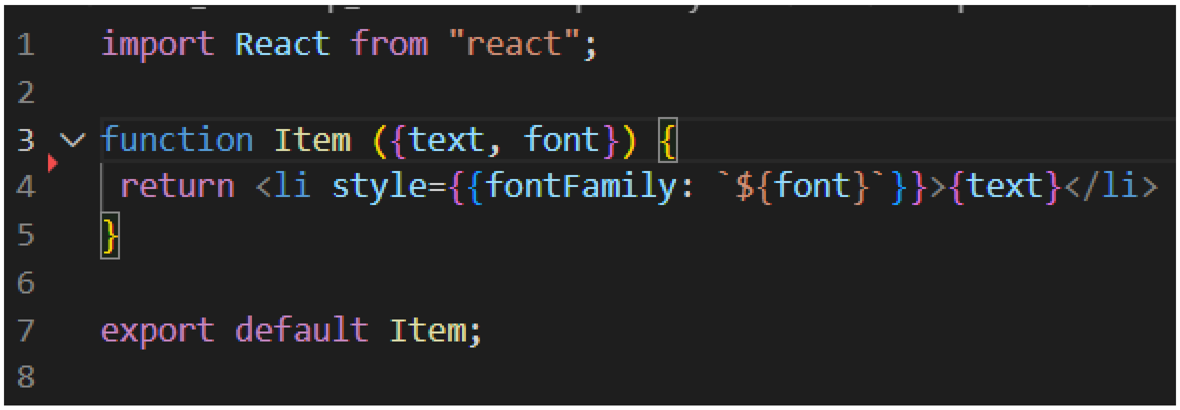


index.js U ×

src > components > App > index.js > App

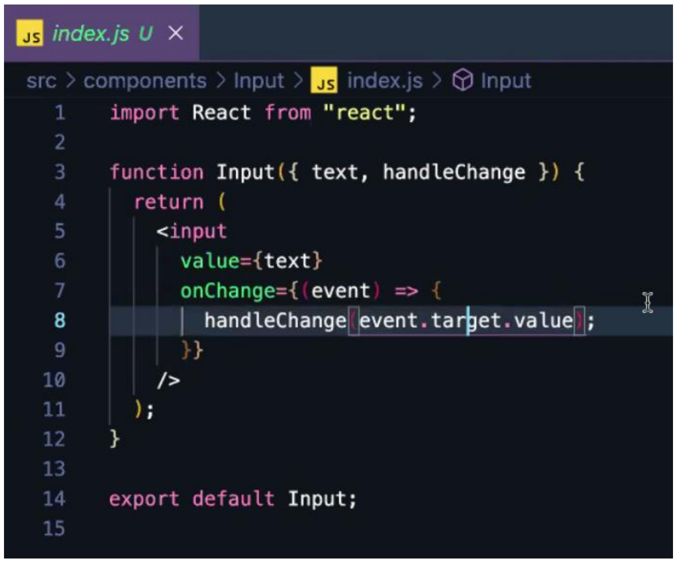
```
  const [text, setText] = useState("");

  function handleChange(userInput) {
    setText(userInput);
  }

  return (
    <main className="App">
      <h1>Font Viewer</h1>
      <Input text={text} handleChange={handleChange} />
      <Display text={text} font={"Times New Roman"} />
      <Display text={text} font={"Verdana"} />
      <Display text={text} font={"Arial"} />
      <Display text={text} font={"Monaco"} />
    </main>
  );
}

export default App;
```

Item component (my code as didn't get a screenshot of Liz's):



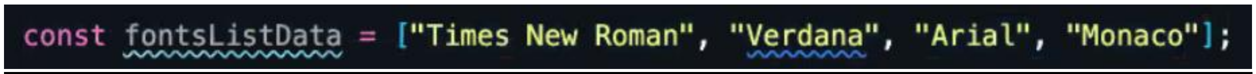
```
import React from "react";

function Item ({text, font}) {
  return <li style={{fontFamily: `${font}`}}>{text}</li>
}

export default Item;

```

Input component:



```
import React from "react";

function Input({ text, handleChange }) {
  return (
    <input
      value={text}
      onChange={(event) => {
        handleChange(event.target.value);
      }}
    />
  );
}

export default Input;

```

Pulling out fonts and having them as an array instead of hardcoding for each Display component:

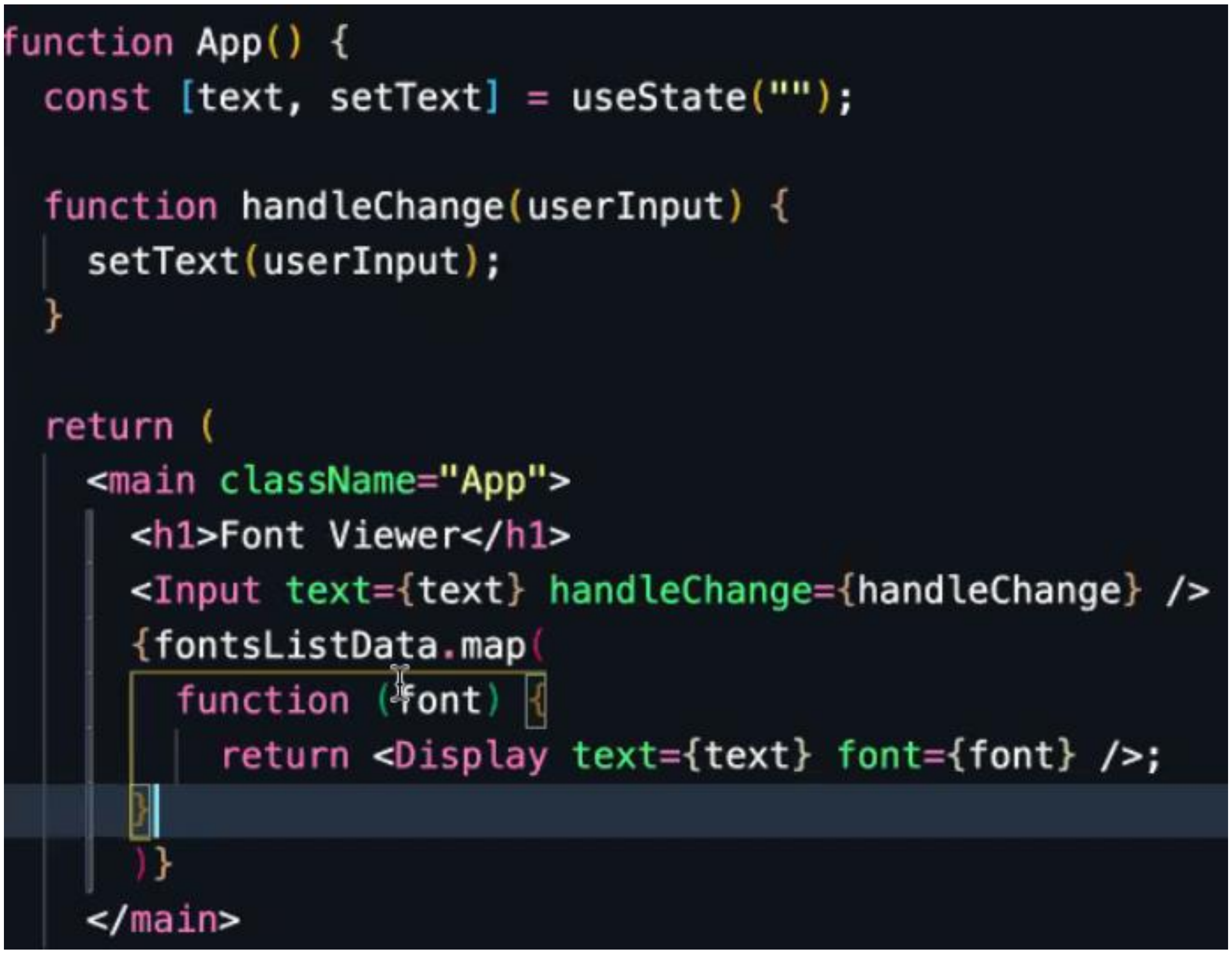


```
const fontsListData = ["Times New Roman", "Verdana", "Arial", "Monaco"];
```

<u>Plan:</u>

- Map over fonts list
- In map callback function, take in each string (font) and return a Display component with that font as a prop

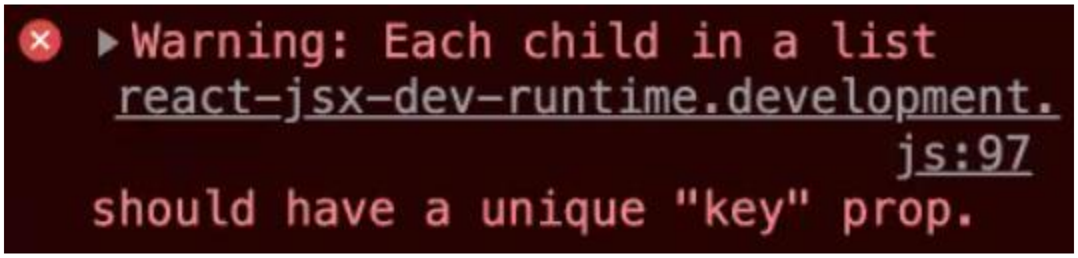
Use .map to loop through array and create a Display component:



```
function App() {
  const [text, setText] = useState("");

  function handleChange(userInput) {
    setText(userInput);
  }

  return (
    <main className="App">
      <h1>Font Viewer</h1>
      <Input text={text} handleChange={handleChange} />
      {fontsListData.map(
        function (font) {
          return <Display text={text} font={font} />;
        }
      )}
    </main>
```

Unique keys

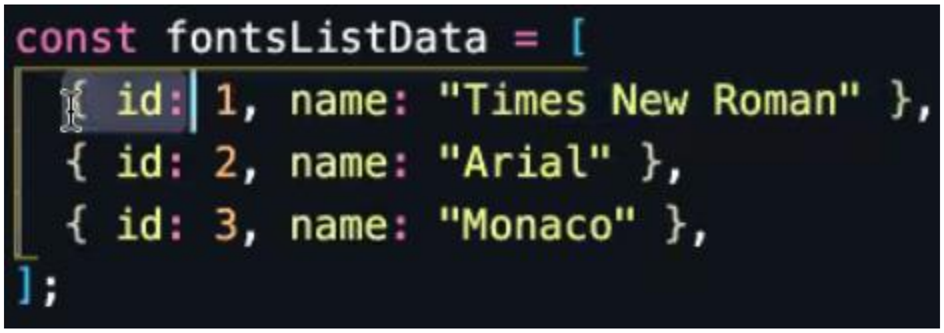
So above code is working but shows an error:



```
▶Warning: Each child in a list
 react-jsx-dev-runtime.development.
                              js:97
 should have a unique "key" prop.
```

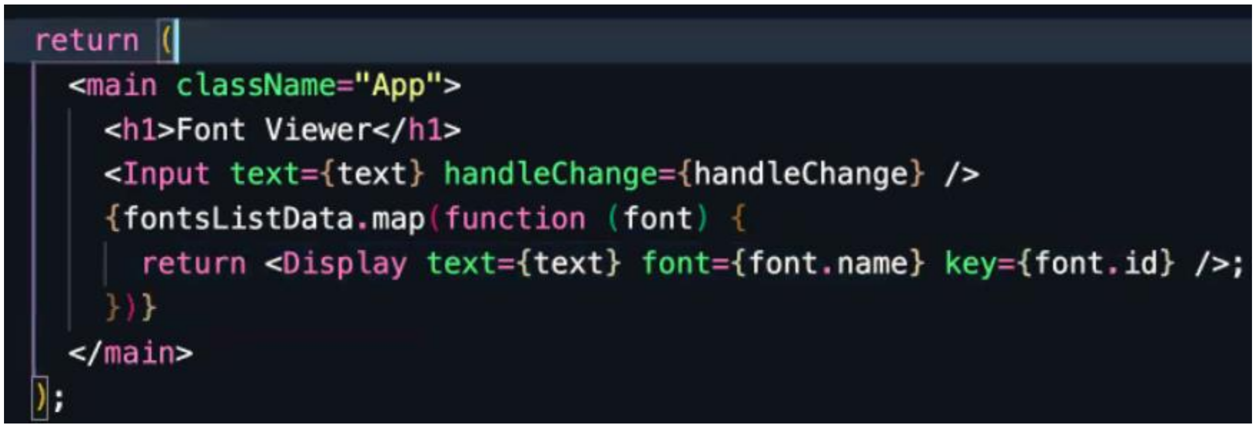
When React is rendering an array, it wants to be able to keep track of every unique item. Similar to generating an index as Primary Key in SQL.

So refactor fontsListData so each item has individual id:



```
const fontsListData = [
  { id: 1, name: "Times New Roman" },
  { id: 2, name: "Arial" },
  { id: 3, name: "Monaco" },
];
```

Then need to refactor the .map Callback function to get the right data out of the above object, ensuring it gives each item a unique key:



```
return (
  <main className="App">
    <h1>Font Viewer</h1>
    <Input text={text} handleChange={handleChange} />
    {fontsListData.map(function (font) {
      return <Display text={text} font={font.name} key={font.id} />;
    })}
  </main>
);
```