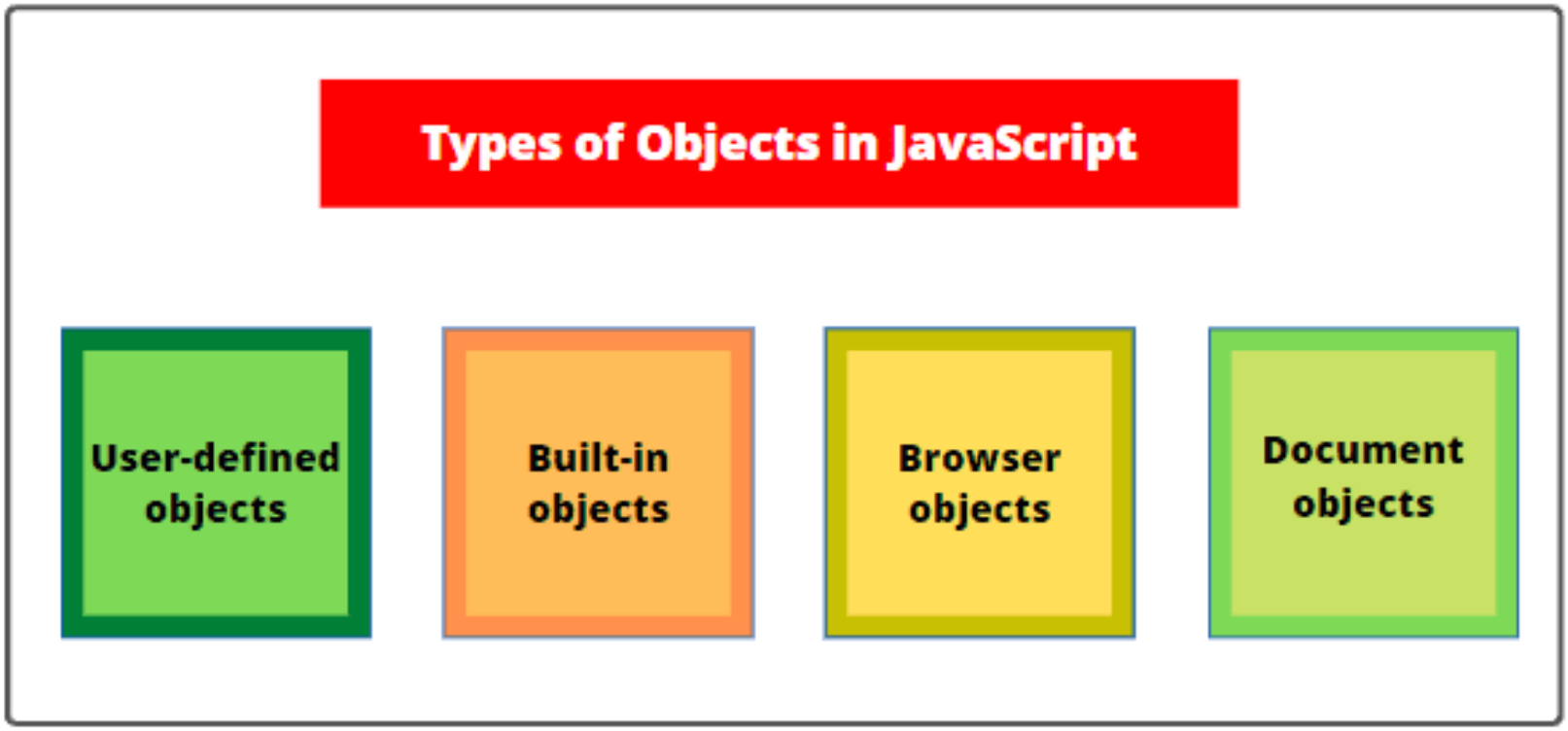
CS385 Mobile Application Development (Lecture 19)

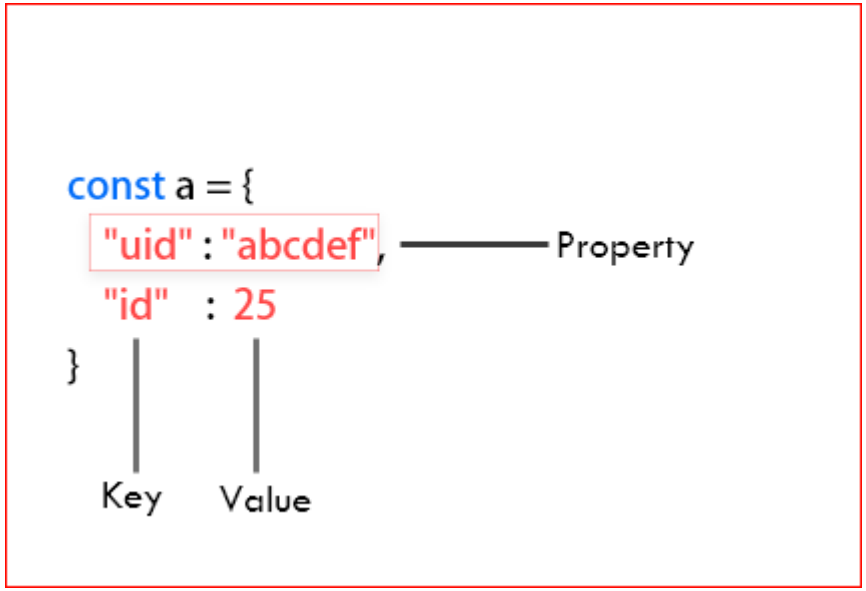
Working with data Objects in Javascript

- Regardless of whatever React functionality and front-end goodness we have been working at – **underneath the hood of any good application is some Javascript code which is working with objects**
- Objects (or just simple primitive types) are what makes your application “do stuff”.
- We have to be sure we know how to work with these objects or types in our applications

A few notes…

- All examples are provided in the Lecture 19 source code zip file.
- **Object** comparison, **Object.getPropertyNames**, and so on will appear in Lab Exam 3
- Working with Objects in this way, as per Lecture 19, is fundamental to understanding how to building robust applications.


Types of Objects in JavaScript
User-defined objects
Built-in objects
Browser objects
Document objects


const a = {
"uid" : "abcdef",
Property
"id" : 25
}
Key
Value

Objects in JavaScript are collections of key-value pairs.

- Each key is a string or a symbol, and the associated value can be any valid JavaScript data type, including other objects.
- **In addition to storing data, objects can also have methods, which are functions that are associated with the object.** Methods can be invoked to perform actions on the object or its data.
- **JavaScript provides various built-in methods for working with objects**, such as **`Object.keys()`**, **`Object.getPropertyNames()`**, **`Object.values()`**, **and** **`Object.entries()`**, which allow you to retrieve information about an object's keys, values, and key-value pairs, respectively.
- These aspects collectively make objects a versatile and powerful feature in JavaScript, enabling developers to model and manipulate complex data structures in their programs.

Example 1: Why are these two objects equal?

App.js

src > App.js > App

```
1  import React from "react";
2
3  function App() {
4    let xObject = { id: 4, name: "Peter" };
5    let yObject = xObject;
6    return (
7      <div className="App">
8        {xObject === yObject && <b>EQUAL</b>}
9        {xObject !== yObject && <b>NOT EQUAL</b>}
10     </div>
11   );
12 }
13
14 export default App;
15
```

Preview

htt

EQUAL

Notice the three equal sign operator

Example 1: Why are these two objects equal? ANS

App.js

src > App.js > App

```
1   import React from "react";
2
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = xObject;
6     return (
7       <div className="App">
8         {xObject === yObject && <b>EQUAL</b>}
9         {xObject !== yObject && <b>NOT EQUAL</b>}
10      </div>
11    );
12  }
13
14  export default App;
15
```

Preview

EQUAL

- Both **xObject** and **yObject** point to the same place in '**object space**'
- Same place in memory

Notice the three equal sign operator

Example 2: Why are these two objects NOT equal?

src > App.js > App

```
1   import React from "react";
2
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = { id: 4, name: "Peter" };
6     return (
7       <>
8         {xObject === yObject && <b>EQUAL</b>}
9         {xObject !== yObject && <b>NOT EQUAL</b>}
10      </>
11    );
12  }
13
14  export default App;
15
```

https://rc

NOT EQUAL

Example 2: Why are these two objects NOT equal? [ANS]

App.js

src > App.js > App

```
1   import React from "react";
2
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = xObject;
6     return (
7       <div className="App">
8         {xObject === yObject && <b>EQUAL</b>}
9         {xObject !== yObject && <b>NOT EQUAL</b>}
10      </div>
11    );
12  }
13
14  export default App;
15
```

Preview

htt

EQUAL

- While xObject and yObject have same properties and values, the === operator sees these objects as occupying to complete different ‘addresses’ or ‘locations in memory

Example 3: Why are these objects not equal?

src > App.js > App

```
1   import React from "react";
2
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = {};
6     // assign the properties individually
7     yObject.id = xObject.id;
8     yObject.name = xObject.name;
9     return (
10      <>
11        <h1>Example 3</h1>
12        {xObject === yObject && <b>EQUAL</b>}
13        {xObject !== yObject && <b>NOT EQUAL</b>}
14      </>
15    );
16  }
17  export default App;
```

https://rc2rj

Example 3

NOT EQUAL

Example 3: Why are these objects not equal?

src > App.js > App

```
1  import React from "react";
2
3  function App() {
4    let xObject = { id: 4, name: "Peter" };
5    let yObject = {};
6    // assign the properties individually
7    yObject.id = xObject.id;
8    yObject.name = xObject.name;
9    return (
10     <>
11       <h1>Example 3</h1>
12       {xObject === yObject && <b>EQUAL</b>}
13       {xObject !== yObject && <b>NOT EQUAL</b>}
14     </>
15   );
16 }
17 export default App;
```

https://rc2rj

Example 3

NOT EQUAL

- For the same reason as before – regardless of property values, they address a different place in memory

Object.is

`Object.is()` determines whether two values are the same value. Two values are the same if one of the following holds:

- both `undefined`
- both `null`
- both `true` or both `false`
- both strings of the same length with the same characters in the same order
- both the same object (means both object have same reference)
- both numbers and
  - both `+0`
  - both `-0`
  - both `NaN`
  - or both non-zero and both not `NaN` and both have the same value

Using the built in `Object.is()` function – still returns the same result (false/not equal)

src > App.js > App

```
1   import React from "react";
2
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = {};
6     // assign the properties individually
7     yObject.id = xObject.id;
8     yObject.name = xObject.name;
9     return (
10      <>
11        <h1>Using Object.is()</h1>
12        {Object.is(xObject, yObject) && <h1>equal</h1>}
13        {!Object.is(xObject, yObject) && <h1>unequal</h1>}
14      </>
15    );
16  }
17  export default App;
```

Edit Ask

https://rc2rj2.csb.app

Using Object.is()

unequal

So what's the difference between == and ===?

- **The == operator** applies various **coercions** to both sides (if they are not the same Type) before testing for equality.
  - If the operands are of different types, it will attempt to convert them to the same type before making the comparison – for example '5' and 5
- **The === identity operator** returns true if the operands are strictly equal with no type conversion.
  - operands must be of a specific type as well as value

Example 4a – Why do we get EQUAL returned now?

src > App.js > ...

```
1   import React from "react";
2   function App() {
3     let xObject = { id: 4, name: "Peter" };
4     let yObject = {};
5     // assign the properties individually
6     yObject.id = xObject.id;
7     yObject.name = xObject.name;
8     // Make two conditions for checking the
9     // equality of the properties.
10    let cond1 = (xObject.name === yObject.name);
11    let cond2 = (xObject.id === yObject.id);
12
13    return (
14      <>
15        <h1>Example 4a</h1>
16        {(cond1 && cond2) && (<b>EQUAL</b>)}
17        {!(cond1 && cond2) && (<b>NOT EQUAL</b>)}
18        </>
19    );
20  }
```

https://rc2

Example 4a

EQUAL

Example 4a – Why do we get EQUAL returned now? [ANS]

src > App.js > ...

```
import React from "react";
function App() {
  let xObject = { id: 4, name: "Peter" };
  let yObject = {};
  // assign the properties individually
  yObject.id = xObject.id;
  yObject.name = xObject.name;
  // Make two conditions for checking the
  // equality of the properties.
  let cond1 = (xObject.name === yObject.name);
  let cond2 = (xObject.id === yObject.id);

  return (
    <>
      <h1>Example 4a</h1>
      {(cond1 && cond2) && (<b>EQUAL</b>)}
      {!(cond1 && cond2) && (<b>NOT EQUAL</b>)}
      </>
  );
}
```

https://rc2

Example 4a

EQUAL

- Here we are performing OBJECT SPECIFIC comparison – based on the properties

Example 5: Why is this NOT EQUAL?

src > App.js > default

```
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = {};
6     // assign the properties individually
7     yObject.id = xObject.id;
8     yObject.name = "PETER";
9
10    // Make two conditions for checking the
11    // equality of the properties.
12    let cond1 = (xObject.name === yObject.name);
13    let cond2 = (xObject.id === yObject.id);
14
15    return (
16      <div className="App">
17        <h1>Example 5</h1>
18        {(cond1 && cond2) && (<b>EQUAL</b>)}
19        {!(cond1 && cond2) && (<b>NOT EQUAL</b>)}
20        </div>
21    );
22  }
```

https://rc2rj2.

Example 5

NOT EQUAL

Example 5: Why is this NOT EQUAL? [ANS]

src > App.js > default

```
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = {};
6     // assign the properties individually
7     yObject.id = xObject.id;
8     yObject.name = "PETER";
9
10    // Make two conditions for checking the
11    // equality of the properties.
12    let cond1 = (xObject.name === yObject.name);
13    let cond2 = (xObject.id === yObject.id);
14
15    return (
16      <div className="App">
17        <h1>Example 5</h1>
18        {(cond1 && cond2) && (<b>EQUAL</b>)}
19        {!(cond1 && cond2) && (<b>NOT EQUAL</b>)}
20        </div>
21    );
22  }
```

https://rc2rj2.

Example 5

NOT EQUAL

- We have different string representations of `Peter` and `PETER` – these are different objects

Example 5 - Updated

```
3   function App() {
4     let xObject = { id: 4, name: "Peter" };
5     let yObject = {};
6     // assign the properties individually
7     yObject.id = xObject.id;
8     yObject.name = "PETER";
9
10    // Make two conditions for checking the
11    // equality of the properties.
12    let cond1 = (xObject.name.toLowerCase() === yObject.name.toLowerCase());
13    let cond2 = (xObject.id === yObject.id);
14
15    return (        ===
16      <div className="App">
17        <h1>Example 5</h1>
18        {(cond1 && cond2) && (<b>EQUAL</b>)}
19        {!(cond1 && cond2) && (<b>NOT EQUAL</b>)}
20        </div>
21    );
22  }
```

Example

EQUAL

Why worry about object equality checks at all in Javascript?

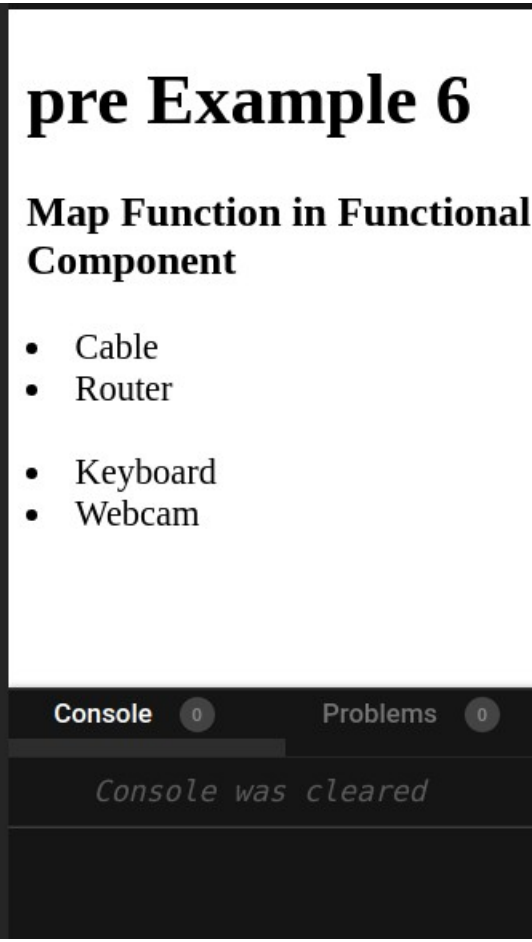
- Because, …. you should worry about these checks in Javascript.
- In CS385 we've written some basic helper functions for **filter, findIndex**, and so on.
- **But in reality, we can make our helper functions as complex as we need.** Very often this will involve checking multiple properties within objects. **Understanding how === works is very important.**

Working with the Map function as an OBJECT in functional Javascript

Reducing the number of lines of code you are writing

Map function calls – as a const variable for rendering

```
3   function App() {
4     let xObject = [
5       { id: 4, price: 17.8, product: "Cable" },
6       { id: 54, price: 27.8, product: "Router" }
7     ];
8     let yObject = [
9       { fid: 4, fprice: 27.5, fproduct: "Keyboard" },
10      { fid: 14, fprice: 37.5, fproduct: "Webcam" }
11    ];
12
13    const xObjectNames = xObject.map((x) => <li key={x.id}>{x.product}</li>);
14    const yObjectNames = yObject.map((y) => <li key={y.fid}>{y.fproduct}</li>)
15
16    return (
17      <div className="App">
18        <h1>pre Example 6</h1>
19        <h3>Map Function in Functional Component</h3>
20        {xObjectNames}
21        <br />
22        {yObjectNames}
23      </div>
24    );
25  }
```



Here – line 13 and 14 we invoke a map function call with JSX. We've seen this many times in our class-based components. But notice that we assign the OUTPUT (rendered evaluated output) of the map function call to a constant variable (as it won't change value)

Object.getOwnPropertyNames()

On this Page

The `Object.getOwnPropertyNames()` method returns an array of all properties (including non-enumerable properties except for those which use Symbol) found directly in a given object.

JavaScript Demo: Object.getOwnPropertyNames()

```
const object1 = {
  a: 1,
  b: 2,
  c: 3
};

console.log(Object.getOwnPropertyNames(object1));
// expected output: Array ["a", "b", "c"]
```

Example:

`Object.getOwnPropertyNames()`

- Returns an array with the property names from an object (in the order the object is originally defined)

```
3   function App() {
4     let xObject = { pkey: 24, name: "Mobile" };
5     let yObject = { pkey: 14, name: "MOBILE",weight: 10 };
6     let zObject = { fkey: 14, Name: "Tablet",fweight: 10 };
7
8     // Object.getOwnProperty names returns an array
9     // with just the property names.
10    // Ordering of the array is exactly as the object is
11    // defined.
12    let xN = Object.getOwnPropertyNames(xObject);
13    let yN = Object.getOwnPropertyNames(yObject);
14    let zN = Object.getOwnPropertyNames(zObject);
15
16    console.log(xN);
17    console.log(yN);
18    console.log(zN);
19
20
21
22
```

Object.getOwnPropertyNames

```
Console 0    Problems 0    React DevTools 0

Console was cleared

▾ (2) ["pkey", "name"]
    0: "pkey"
    1: "name"

▾ (3) ["pkey", "name", "weight"]
    0: "pkey"
    1: "name"
    2: "weight"

▾ (3) ["fkey", "Name", "fweight"]
    0: "fkey"
    1: "Name"
    2: "fweight"
```

Example 6 – considering PROPERTY names in objects

- **GetOwnPropertyNames** returns an array

src > App.js > default

```
3   function App() {
4     let xObject = { id: 54, price: 27.8, product: "Router" };
5     let yObject = {
6       fid: 4,
7       fprice: 27.5,
8       fproduct: "Keyboard",
9       manufacturer: "Dell"
10    };
11
12    const xNames = Object.getOwnPropertyNames(xObject);
13    const yNames = Object.getOwnPropertyNames(yObject);
14
15    return (
16      <>
17        <h1>Example 6</h1>
18        x={xNames.length},y={yNames.length}
19      </>
20    );
21  }
```

https://rc2

Example 6

x=3,y=4

We can access this array like any other Javascript array

```
function App() {
  let xObject = { id: 54, price: 27.8, product: "Router" };
  let yObject = {fid: 4, fprice: 27.5, fproduct: "Keyboard",
  manufacturer: "Dell"};
  // create an array of the property names of xObject
  const xObjectNames = Object.getOwnPropertyNames(xObject);
  // create a map function rendering of the xObjectNames array
  const xMap = xObjectNames.map((x, i) => <li key={i}>{x}</li>);

  const yObjectNames = Object.getOwnPropertyNames(yObject);
  const yMap = yObjectNames.map((y, i) => <li key={i}>{y}</li>);

  return (
    <>
      <h1>Example 6</h1>
      <b>xObject getOwnPropertyNames</b>{xMap}
      <b>yObject getOwnPropertyNames</b>{yMap}
    </>
  );
```

Notice our short-hand way of invoking the map function call?

Example 6

xObject getOwnPropertyNames

- id
- price
- product

yObject getOwnPropertyNames

- fid
- fprice
- fproduct
- manufacturer

‘Shallow’ object comparison in Javascript code

- Depending on the logic in your application it may be necessary to compare to objects using “shallow” object comparision.
- Suppose we have two objects x and y. Then a shallow comparision of x and y inspects all of the properties of both objects and their corresponding value.
- The properties are compared as basic data types.

Example: ‘Shallow’ object comparison

```
3   // write our own function to perform a
4   // shallow comparison of two objects
5   // a and b are unknown objects.
6   function isEquivalent(a, b) {
7     let aPropNames = Object.getOwnPropertyNames(a);
8     let bPropNames = Object.getOwnPropertyNames(b);
9
10    // if the two objects do not have the same number of pro
11    if (aPropNames.length !== bPropNames.length) {
12      return false;
13    }
14    // Now check each property in Object A. Check if this
15    // property exists in Object B.
16    for (let i = 0; i < aPropNames.length; i++) {
17      let propName = aPropNames[i];
18      if (a[propName] !== b[propName]) {
19        return false;
20      }
21    } // end for
22    // if we get out of the for loop then our objects are th
23    return true;
24  }
```

- The **isEquivalent** function will ensure that we have **the same properties and values** (no conversions are made)
- You can of course make specific types of comparisons depending on what you want equality of objects to mean in your code.

Example 7 – using shallow comparison `isEquivalent(a,b)`

```
function App() {
  let xObject1 = { id: 54, price: 27.8, product: "Router" };
  let xObject2 = { id: 54, price: 27.800001, product: "Router" };
  let xObject3 = { id: 54, price: 27.8, product: "Router" };
  let xObject4 = { id: 54, price: 27.8, product: "Router" };
  let xObject5 = { id: 54, price: 27.8, product: "Router", name: "Electron" };

  return (
    <>
      <h1>Example 7</h1>
      {isEquivalent(xObject1, xObject2) && <b>xObject1,xObject2 EQUAL</b>}
      {!isEquivalent(xObject1, xObject2) && <b>xObject1,xObject2 NOT EQUAL</b>}
      <br />
      {isEquivalent(xObject3, xObject4) && <b>xObject3,xObject4 EQUAL</b>}
      {!isEquivalent(xObject3, xObject4) && <b>xObject3,xObject4 NOT EQUAL</b>}
      <br />
      {isEquivalent(xObject4, xObject5) && <b>xObject4,xObject5 EQUAL</b>}
      {!isEquivalent(xObject4, xObject5) && <b>xObject4,xObject5 NOT EQUAL</b>}
    </>
  );
}
```

Example 7

xObject1,xObject2 NOT EQUAL
xObject3,xObject4 EQUAL
xObject4,xObject5 NOT EQUAL

Object comparison - summary

- We have seen here that comparing objects is more detailed than we first imagined.
- “Shallow” comparison can be OK in most cases – but it will reject objects which the same properties but different values – such as one name in capitals and the othe lower case.
- It may also not be able to consider certain numerical comparisons the same – for example 88.6780 is the same as 88.6779 (depending on our accuracy)
- **Depending on your application, you might need to write application-specific object comparision code.**

Deep object comparison

- Depending on your objects, and the types of comparisons you need to make, you might need to write your own version of **isEquivalent** with specific extra conditions included.
- This is not unusual. For example, you might write regular expressions to check two properties for string equality. This would allow, for example, for spelling errors in property values to be considered the same value.
- It is the one way where you, the developer, has complete control over the comparison of two JSON objects **a** and **b**.

Our current STRICT approach to finding objects in an array

```
import React from "react";
function App() {
  let original = [
    { x: 4, y: 5 },{ x: -99, y: 125 }, { x: 99, b: 125 }, { x: 99, y: 125 },
    { x: 28, y: 1000 }];

  let peter = { x: 99, y: 125 };
  let mary = { x: 100, y: 126 };

  function findObject(needle) {
    return function (haystack) {
      return haystack.x === needle.x && haystack.y === needle.y;
    };
  }

  let n = original.findIndex(findObject(peter));
  let n1 = original.findIndex(findObject(mary));

  return (
    <>
      {n >= 0 && <h1>Yes, the object is in the array</h1>}
      {n < 0 && <h1>No, the object is not in the array</h1>}
      {n1 >= 0 && <h1>Yes, the object is in the array</h1>}
      {n1 < 0 && <h1>No, the object is not in the array</h1>}
    </>
  );
}
```

https://rc2rj2.csb.app

Yes, the object is in the array

No, the object is not in the array

Application specific ‘fuzzy’ matching based on values

```
2   function App() {
3     let original = [
4       { x: 4, y: 5 },
5       { x: -99, y: 125 },
6       { x: 99, b: 125 },
7       { x: 99, y: 125 },
8       { x: 28, y: 1000 },
9     ];
10
11    let peter = { x: 99, y: 125 };
12    let mary = { x: 100, y: 126 };
13    // allow for some 'difference' between the x and y values
14    function fuzzyFindObject(needle) {
15      return function (haystack) {
16        return (
17          Math.abs(haystack.x - needle.x) < 5 &&
18          Math.abs(haystack.y - needle.y) < 5
19        );
20      };
21    }
```

Important

```
23    let n = original.findIndex(fuzzyFindObject(peter));
24    let n1 = original.findIndex(fuzzyFindObject(mary));
25
26    return (
27      <>
28        {n >= 0 && <h1>Yes, the object is in the array</h1>}
29        {n < 0 && <h1>No, the object is not in the array</h1>}
30        {n1 >= 0 && <h1>Yes, the object is in the array</h1>}
31        {n1 < 0 && <h1>No, the object is not in the array</h1>}
32      </>
33    );
```

Yes, the object is in the array

Yes, the object is in the array

Regular expression fuzzy matching

App.js

src > App.js > App > search2 > x

```
import React from "react";
function App() {
  let original = [
    { x: 4, y: "rHj78h" },
    { x: 99, y: "CS385" },
    { x: 99, y: "Mobile385" },
    { x: 99, y: "App385" },
    { x: 28, y: "Peter356" },
  ];
  const strPattern = /^.*\d{3,5}$/;
  let search1 = { x: 99, y: "CS9999" };
  let search2 = { x: 28, y: "ANDROID1234" };
  // fuzzy comparision of x and y
  // x is numerical (+/- 0 .. 9)
  // y is a regex
  function fuzzyFindObject(needle) {
    return function (haystack) {
      return haystack.x === needle.x && strPattern.test(needle.y);
    };
  }

  let n = original.findIndex(fuzzyFindObject(search1));
  let n1 = original.findIndex(fuzzyFindObject(search2));

  return (
    <>
      {n >= 0 && <h1>Yes, the object is in the array</h1>}
      {n < 0 && <h1>No, the object is not in the array</h1>}
      {n1 >= 0 && <h1>Yes, the object is in the array</h1>}
      {n1 < 0 && <h1>No, the object is not in the array</h1>}
    </>
```

Preview

https://rc2rj2.csb.app

Yes, the object is in the array

Yes, the object is in the array

Rendering unknown or previously unseen objects in React

Rendering unseen or unknown objects in React

- In every example we have used in CS385 we have knowledge about the objects we are working with. This means that we know their properties. Then we can use map, filter, sort, findIndex, and so on.
- But **what happens if we do not have knowledge about the objects? Is it possible to render these objects despite the fact that we do not know their properties?**
- Yes, we can render unseen objects – using `Object.getOwnPropertyNames` and a **double or nested map function** call

Classical, standard, map function call to render KNOWN objects. This means we know the properties of all objects. Here we have an array of objects

```
function App() {
  // classical standard map function code
  // to render an array of objects.
  // We know the properties of all objects in advance.
  let arrObjects = [
    { id: 54, price: 27.8, product: "Router" },
    { id: 123, price: 89.99, product: "Monitor" },
    { id: 223, price: 189.99, product: "SSD Hard Drive" },
  ];
  return (
    <>
      <h1>Example 8</h1>
      {arrObjects.map((x) => (
        <li key={x.id}>
          {x.product},€{x.price}
        </li>
      ))}
    </>
  );
}
```

Example 8

Router,€27.8
Monitor,€89.99
SSD Hard Drive,€189.99

Example 8

Example 9 – rendering objects with a nested map function call

- Conceptually think of a nested for loop. Line 14 is the **map** function for the array. We will process each object one by one. Line 16 is where we have a second **map** function. This map function will process each item in the array created by **Object.getOwnPropertyNames**

App.js > App > arrObjects

```
let arrObjects = [
  { id: 54, price: 27.8, product: "Router" },
  { id: 123, price: 89.99, product: "Monitor" },
  { id: 223, price: 189.99, product: "SSD Hard Drive" },
  { id: 223, price: 189.99, product: "SSD Hard Drive" },
];
return (
  <>
    <h1>Example 9 - Rendering Unseen Objects</h1>
    <p>
      Using a nested map function to render an object array. Here we do not
      need to specifically write the names of properties in order to render
      the values{" "}
    </p>
    {arrObjects.map((x, i) => (
      <p key={i}>
        {Object.getOwnPropertyNames(x).map((y, i) => (
          <li key={i}>
            <b>Property:</b> {y}  <b>Value:</b> {x[y]}
          </li>
        ))}
      </p>
```

Open command palette CTRL SHIFT P

https://rc2rj2.csb.app

Example 9 - Rendering Unseen Objects

Using a nested map function to render an object array. Here we do not need to specifically write the names of propertie in order to render the values

Property: id **Value:** 54
Property: price **Value:** 27.8
Property: product **Value:** Router

Property: id **Value:** 123
Property: price **Value:** 89.99
Property: product **Value:** Monitor

Property: id **Value:** 223

The value **y is** the PROPERTY. Then we have **x[y]** to obtain the value of the current property for the current object **x**.

Example 9a – we can have different objects in our array

```
function App() {
  let arrObjects = [
    { id: 54, price: 27.8, product: "Router" },
    { Fid: 123, Fprice: 89.99, Fproduct: "Monitor" },
    { product: "SSD Hard Drive", price: 189.99, id: 223 },
    { alpha: 20, beta: "Text", gamma: "More text" },
  ];

  return (
    <>
      <h1>Example 9a - Rendering Unseen Objects</h1>
      {arrObjects.map((x, i) => (
        <p key={i}>
          {Object.getOwnPropertyNames(x).map((y, i) => (
            <li key={i}>
              <b>Property:</b> {y}  <b>Value:</b> {x[y]}
            </li>
          ))}
        </p>
      ))}
    </>
```

Example 9a - Rendering Unseen Objects

Property: id **Value:** 54
Property: price **Value:** 27.8
Property: product **Value:** Router

Property: Fid **Value:** 123
Property: Fprice **Value:** 89.99
Property: Fproduct **Value:** Monitor

Property: product **Value:** SSD Hard Drive
Property: price **Value:** 189.99
Property: id **Value:** 223

Property: alpha **Value:** 20
Property: beta **Value:** Text
Property: gamma **Value:** More text

Example 9, 9a – PROBLEMS

- **This nested map function code is very flexible** (any object, ordering of properties does not matter). **However, it has a number of disadvantages**:
  - **"Look-and-feel"** – you would need to add lots of additional code (conditional rendering) to render specific properties in a particular way (for example bold text, color, and so on)
  - **Messy Array** – where are you getting these object arrays from? The data is very messy and unstructured and difficult to work with.
  - **Filtering** – very difficult to write useful filter functions if you do not know the properties of the objects in advance

Example 10: Dealing with `undefined` properties in objects

```
7   const [vegetables, setVegetables] = useState([
8     { name: "Cabbage", price: 1.5 },
9     { name: "Carrots", price: 0.8 }
10  ]);
11
12  const [fruit, setFruit] = useState([
13    { name: "Orange", price: 0.55, origin: "ES" },
14    { name: "Apple", price: 0.45, origin: "IE" }
15  ]);
16
17  // Combine the two arrays using the spread operator
18  // this is just .... before the array
19  let comboArray = [...vegetables, ...fruit];
```

- The problem here with the spread operator result on line 19 is that the property **origin** is only available for the fruit objects. When we are rendering the **comboArray** we will get an error if we try to render the **origin** property for the vegetable objects.
- Luckily, **we can test if a property is `undefined` in Javascript**

Example 10: We test if the origin property is `undefined` for the current object in the map function

```
function App() {
  const [vegetables, setVegetables] = useState([
    { name: "Cabbage", price: 1.5 },
    { name: "Carrots", price: 0.8 },
  ]);

  const [fruit, setFruit] = useState([
    { name: "Orange", price: 0.55, origin: "ES" },
    { name: "Apple", price: 0.45, origin: "IE" },
  ]);

  let comboArray = [...vegetables, ...fruit];

  return (
    <div className="App">
      <h1>Example 10</h1>
      <h3>Using 'undefined' for object properties</h3>
      {comboArray.map((c, i) => (
        <li key={i}>
          {c.name}, Price: €{c.price} 
          {c.origin === undefined && <b>N/A</b>}
          {c.origin !== undefined && <b>{c.origin}</b>}
        </li>
```

Edit Ask

Example 10

Using 'undefined' for object properties

Name: Cabbage, Price: €1.5 **N/A**

Name: Carrots, Price: €0.8 **N/A**

Name: Orange, Price: €0.55 **ES**

Name: Apple, Price: €0.45 **IE**

Objects: summary

- Objects are the very life and soul of Javascript programming. It is difficult to avoid using them when writing Javascript.
- The previous examples (all source code available on Moodle) are an exploration into ways in which you can work with objects using good programming practice.
- **We also seen how to handle situations which could introduce errors into application (such as missing properties, object equality, and so on)**

Removing or changing properties in arrays of objects

Original Array

- {"id":1,"name":"Alice","module":"CS385"}
- {"id":21,"name":"Simon","module":"CS440"}
- {"id":11,"name":"Sarah","module":"CS285"}
- {"id":211,"name":"Stephen","module":"CS210"}
- {"id":20,"name":"Anne","module":"CS171"}
- {"id":12,"name":"Sarah","module":"CS130"}
- {"id":23,"name":"Toby","module":"CS385"}

New Array

- {"id":1,"name":"Alice"}
- {"id":21,"name":"Simon"}
- {"id":11,"name":"Sarah"}
- {"id":211,"name":"Stephen"}
- {"id":20,"name":"Anne"}
- {"id":12,"name":"Sarah"}
- {"id":23,"name":"Toby"}

Removing or changing properties in arrays of objects

```
4   const originalArray = [
5     { id: 1, name: "Alice", module: "CS385" },
6     { id: 21, name: "Simon", module: "CS440" },
7     { id: 11, name: "Sarah", module: "CS285" },
8     { id: 211, name: "Stephen", module: "CS210" },
9     { id: 20, name: "Anne", module: "CS171" },
10    { id: 12, name: "Sarah", module: "CS130" },
11    { id: 23, name: "Toby", module: "CS385" },
12  ];
13  //use a map function
14  //specify the names of the properties you want to ke
15  // don't specify the ones for removal.
16  const newArray = originalArray.map((item) => {
17    return {
18      id: item.id,
19      name: item.name,
20    };
21  });
```

- The map function allows to specify a return statement

```
<h2>New Array</h2>
<ul>
  {newArray.map((obj, index) => (
    <li key={index}>{JSON.stringify(obj)}</li>
  ))}
</ul>
</>
```

Original Array

- {"id":1,"name":"Alice","module":"CS385"}
- {"id":21,"name":"Simon","module":"CS440"}
- {"id":11,"name":"Sarah","module":"CS285"}
- {"id":211,"name":"Stephen","module":"CS210"}
- {"id":20,"name":"Anne","module":"CS171"}
- {"id":12,"name":"Sarah","module":"CS130"}
- {"id":23,"name":"Toby","module":"CS385"}

New Array

- {"id":1,"name":"Alice"}
- {"id":21,"name":"Simon"}
- {"id":11,"name":"Sarah"}
- {"id":211,"name":"Stephen"}
- {"id":20,"name":"Anne"}
- {"id":12,"name":"Sarah"}
- {"id":23,"name":"Toby"}

Using deconstruction in arrays

```
const { name } = user;

const user = {
  'name': 'Alex',
  'address': '15th Park Avenue',
  'age': 43
}
```

@tapasadhikary

- **The destructuring assignment** syntax is a JavaScript expression that makes it possible to **unpack values** from arrays, or properties from objects, into distinct variables.

Rest property

You can end a destructuring pattern with a rest property `...rest`. This pattern will store all remaining properties of the object or array into a new object or array.

JS

```js
const { a, ...others } = { a: 1, b: 2, c: 3 };
console.log(others); // { b: 2, c: 3 }

const [first, ...others2] = [1, 2, 3];
console.log(others2); // [2, 3]
```

```
3   function App() {
4     // Example object
5     const person = {
6       name: 'John Fahy',
7       age: 30,
8       city: 'New York',
9     };
10
11    // Object destructuring
12    const { name, age, city } = person;
13
14    return (
15      <div>
16        <h2>Person Information</h2>
17        <p>Name: {name}</p>
18        <p>Age: {age}</p>
19        <p>City: {city}</p>
20      </div>
21    );
22  }
```

Person Information

Name: John Fahy

Age: 30

City: New York

Example – object deconstruction in arrays – removing a property

```
function App() {
  const originalArray = [
    { id: 1, name: "John", age: 25 },
    { id: 2, name: "Jane", age: 30 },
    { id: 3, name: "Colm", age: 28 },
  ];
  // deconstruct the array objects - remember, map iterates
  // over all of the objects in the array
  // each time ...rest is the other properties.
  // Remove the "age" property from each object in the array
  const newArrayWithoutAge = originalArray.map(({ age, ...rest }) => {
    return rest;
  });
```

Original Array

- {"id":1,"name":"John","age":25}
- {"id":2,"name":"Jane","age":30}
- {"id":3,"name":"Colm","age":28}

New Array

- {"id":1,"name":"John"}
- {"id":2,"name":"Jane"}
- {"id":3,"name":"Colm"}

Creating a new property for all objects in the array

```
function App() {
  const originalArray = [
    { id: 1, name: "John", exam1: 25, exam2: 30 },
    { id: 2, name: "Jane", exam1: 30, exam2: 10 },
    { id: 3, name: "Colm", exam1: 28, exam2: 15 },
  ];
  // deconstruct the array objects - remember, map iterates
  // over all of the objects in the array
  // each time ...element is all of the properties.
  // Create a new property total score

  const newArray = originalArray.map((element) => ({
    ...element,
    totalScore: element.exam1 + element.exam2,
  }));
```

Original Array

- {"id":1,"name":"John","exam1":25,"exam2":30}
- {"id":2,"name":"Jane","exam1":30,"exam2":10}
- {"id":3,"name":"Colm","exam1":28,"exam2":15}

New Array

- {"id":1,"name":"John","exam1":25,"exam2":30,"totalScore":55}
- {"id":2,"name":"Jane","exam1":30,"exam2":10,"totalScore":40}
- {"id":3,"name":"Colm","exam1":28,"exam2":15,"totalScore":43}

Using decontruction to hide data in an object array

```
function App() {
  const originalArray = [
    { id: 1, name: "John", exam1: 25, exam2: 30 },
    { id: 2, name: "Jane", exam1: 30, exam2: 10 },
    { id: 3, name: "Colm", exam1: 28, exam2: 15 },
  ];
  // deconstruct the array objects - remember, map iterates
  // over all of the objects in the array
  // each time ...element is all of the properties.
  // This time we drop the exam1 and exam2 properties
  // Create a new property total score

  const newArray = originalArray.map((element) => ({
    id: element.id,
    name: element.name,
    totalScore: element.exam1 + element.exam2
  }));
```

Original Array

- {"id":1,"name":"John","exam1":25,"exam2":30}
- {"id":2,"name":"Jane","exam1":30,"exam2":10}
- {"id":3,"name":"Colm","exam1":28,"exam2":15}

New Array

- {"id":1,"name":"John","totalScore":55}
- {"id":2,"name":"Jane","totalScore":40}
- {"id":3,"name":"Colm","totalScore":43}

Working with duplicate objects

- How can we remove or identify duplicate objects with an array of objects?

```
const employees = [
  { id: 1, name: "Alice" },
  { id: 21, name: "Simon" },
  { id: 1, name: "Sarah" },
  { id: 21, name: "Simon" },
  { id: 2, name: "Anne" },
  { id: 1, name: "Sarah" },
  { id: 23, name: "Toby" },
];
```

Removing duplicate objects is reasonably difficult, depending on how we define ‘duplicate’

- If we are just checking ONE property, such as an ‘id’ property, then it is relatively straighforward to check for duplicates.
- The algorithmic approach becomes more complicated when we try to look for REAL duplicate objects – that is, objects that have the same keys and values (for every key-value pair)

Example – removing duplicates by considering ONE property

```
const employees = [
  { id: 1, name: "Alice" },
  { id: 21, name: "Simon" },
  { id: 1, name: "Sarah" },
  { id: 21, name: "Simon" },
  { id: 2, name: "Anne" },
  { id: 1, name: "Sarah" },
  { id: 23, name: "Toby" },
];
```

```
let uniqueIds = [];

// element is the object in the employees array
// it will be checked for unique-ness

function checkObject(element) {
  let isDuplicate = uniqueIds.includes(element.id);

  if (!isDuplicate) {
    // place the id into the array uniqueIds
    uniqueIds.push(element.id);
    return true;
  } else return false;
}

let uniqueEmployees = employees.filter(checkObject);
```

1 Alice

21 Simon

2 Anne

23 Toby

```
return (
  <>
    {uniqueEmployees.map((v, index) => (
      <p key={index}>
        {v.id} {v.name}
      </p>
    ))}
  </>
```

Lab Exam 3 – initial hints from this lecture

Lab Exam 3 will contain

- Questions on **nested map functions**
- Questions which use the `Object.getOwnPropertyNames()` function
- Questions on **object equality** (these will involve **conditional rendering**)
- The usual questions on **updating state**
- **Fuzzy matching** of objects.
- Undefined objects
- Using **Object Deconstruction**
- **In class demo lab 3 next week.**