JANUARY 27, 2022 / #JAVASCRIPT

# Top JavaScript Concepts to Know Before Learning React

Joel Olawanle

If you want to learn React –or any JavaScript framework – you'll first need to understand the fundamental JavaScript methods and concepts.

Otherwise it's like a youngster learning to run before learning to walk.

Many developers choose a "learn as you go" approach when learning React. But this often doesn't result in productivity, and instead worsens the gaps in their JavaScript knowledge. This approach makes assimilating each new feature twice as difficult (you might begin to confuse JavaScript with React).

React is a JavaScript framework for building UI components-based user interfaces. All of its code is written in JavaScript, including the HTML markup, which is written in JSX (this enables developers to easily write HTML and JavaScript together).

In this post, we'll take a practical approach and go over all of the JS ideas and techniques you'll need to grasp before learning React.

In this post. To help you deepen your learning, I will connect distinct links to each method and concept.

*Let's get started...*

# The JavaScript You Need to Know Before Learning React

## Callback Functions in JavaScript

A callback function is a function that is performed after another function has completed its execution. It is typically supplied as an input into another function.

Callbacks are critical to understand since they are used in array methods (such as `map()`, `filter()`, and so on), `setTimeout()`, event listeners (such as click, scroll, and so on), and many other places.

Here's an example of a "click" event listener with a callback function that will be run whenever the button is clicked:

```
//HTML
<button class="btn">Click Me</button>

//JavaScript
const btn = document.querySelector('.btn');

btn.addEventListener('click', () => {
  let name = 'John doe';
```

> **NB:** A callback function can be either an ordinary function or an arrow function.

# Promises in JavaScript

As previously stated, a callback function is executed after the original function is executed. You may now begin to consider stacking so many callback functions on top of each other because you do not want a specific function to run until the parent function has finished running or a specific time has passed.

For example, let's attempt to display 5 names in the console after 2 seconds each – that is, the first name appears after 2 seconds, the second after 4 seconds, and so on...

```
setTimeout(() => {
    console.log("Joel");
    setTimeout(() => {
        console.log("Victoria");
        setTimeout(() => {
            console.log("John");
            setTimeout(() => {
                console.log("Doe");
                setTimeout(() => {
                    console.log("Sarah");
                }, 2000);
            }, 2000);
        }, 2000);
    }, 2000);
}, 2000);
```

This above example will work, but it will be difficult to comprehend, debug, or even add error handling to. This is referred to as**"Callback**

The primary reason for using promises is to prevent callback hell. With Promises, we may write asynchronous code in a synchronous manner.

**Gotcha:** You can learn what synchronous and asynchronous means in JavaScript via this article by TAPAS ADHIKARY.

A promise is an object that returns a value that you anticipate to see in the future but do not now see.

A practical use for promises would be in HTTP requests, where you submit a request and do not receive a response right away because it's an asynchronous activity. You only receive the answer (data or error) when the server responds.

JavaScript promise syntax:

```
const myPromise = new Promise((resolve, reject) => {
    // condition
});
```

Promises have two parameters, one for success (resolve) and one for failure (reject). Each has a condition that must be satisfied in order for the Promise to be resolved – otherwise, it will be rejected:

```
const promise = new Promise((resolve, reject) => {
    let condition;

    if(condition is met) {
        resolve('Promise is resolved successfully.');
    } else {
```

There are 3 states of the Promise object:

- **Pending:** by default, this is the Initial State, before the Promise succeeds or fails.
- **Resolved:** Completed Promise
- **Rejected:** Failed Promise

Finally, let's try to re-implement the callback hell as a promise:

```
function addName (time, name){
  return new Promise ((resolve, reject) => {
    if(name){
      setTimeout(()=>{
        console.log(name)
        resolve();
      },time)
    }else{
      reject('No such name');
    }
  })
}

addName(2000, 'Joel')
  .then(()=>addName(2000, 'Victoria'))
  .then(()=>addName(2000, 'John'))
  .then(()=>addName(2000, 'Doe'))
  .then(()=>addName(2000, 'Sarah'))
  .catch((err)=>console.log(err))
```

You can check through this article by Cem Eygi to better understand promises.

function. The callback function will be run on each array element.

Assume we have an array of users that contains their information.

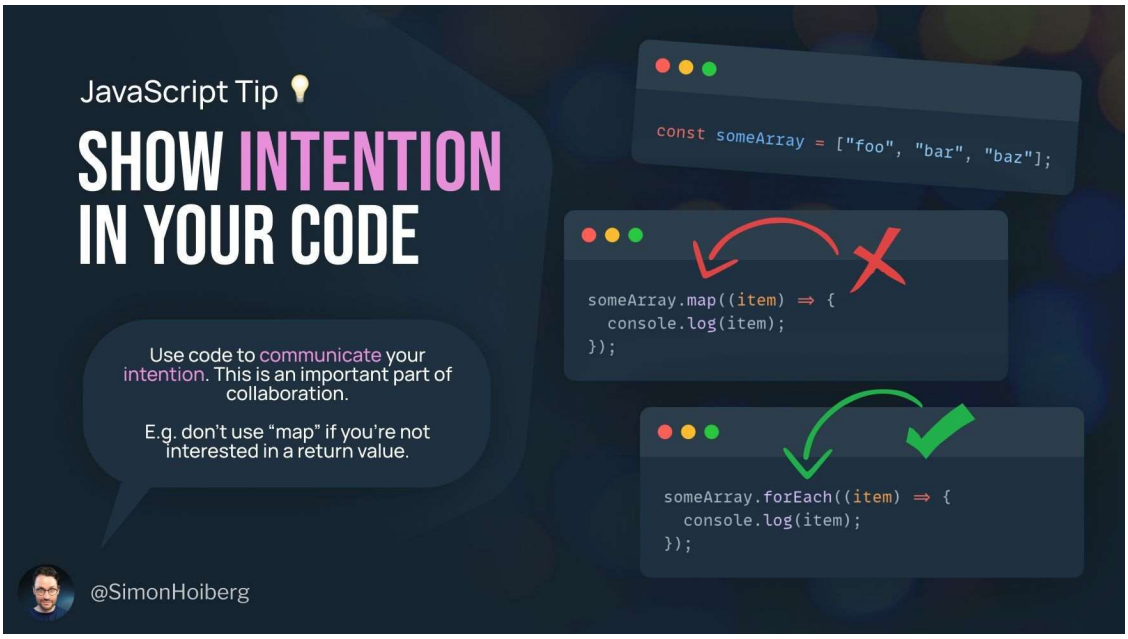
```
let users = [
  { firstName: "Susan", lastName: "Steward", age: 14, hobby: "Singing"
  { firstName: "Daniel", lastName: "Longbottom", age: 16, hobby: "Foot
  { firstName: "Jacob", lastName: "Black", age: 15, hobby: "Singing" }
];
```

We can loop through using map and modify it's output

```
let singleUser = users.map((user)=>{
  //let's add the firstname and lastname together
  let fullName = user.firstName + ' ' + user.lastName;
  return `
    <h3 class='name'>${fullName}</h3>
    <p class="age">${user.age}</p>
  `
});
```

You should note that:

- `map()` always returns a new array, even if it's an empty array.
- It doesn't change the size of the original array compared to the filter method
- It always makes use of the values from your original array when making a new one.




Here is a perfect description by Simon Høiberg

One of the key reasons we use map is so we can encapsulate our data in some HTML, whereas for React this is simply done using JSX.

You can read more about map() here.

# Filter() and Find() in JavaScript

`Filter()` provides a new array depending on certain criteria. Unlike map(), it can alter the size of the new array, whereas `find()` returns just a single instance (this might be an object or item). If several matches exist, it returns the first match – otherwise, it returns undefined.

Suppose you have an array collection of registered users with different ages:

```
  { firstName: "Bruno", age: 56 },
  { firstName: "Jacob", age: 15 },
  { firstName: "Sam", age: 64 },
  { firstName: "Dave", age: 56 },
  { firstName: "Neils", age: 65 }
];
```

You could choose to sort this data by age groups, such as young individuals (ages 1-15), senior people (ages 50-70), and so on...

In this case, the filter function comes in handy as it produces a new array based on the criteria. Let's have a look at how it works.

```
// for young people
const youngPeople = users.filter((person) => {
  return person.age <= 15;
});

//for senior people
const seniorPeople = users.filter((person) => person.age >= 50);

console.log(seniorPeople);
console.log(youngPeople);
```

This generates a new array. It produces an empty array if the condition is not satisfied(no match).

You can read more about this here.

## Find()

The `find()` method, like the `filter()` method, iterates across the array looking for an instance/item that meets the specified condition. Once it finds it, it returns that specific array item and

**For example:**

```
const Bruno = users.find((person) => person.firstName === "Bruno");

console.log(Bruno);
```

You can read more about the find() method here.

# Destructuring Arrays and Objects in JavaScript

Destructuring is a JavaScript feature introduced in ES6 that allows for faster and simpler access to and unpacking of variables from arrays and objects.

Before destrucuring was introduced, if we had an array of fruits and wanted to get the first, second, and third fruits separately, we would end up with something like this:

```
let fruits= ["Mango", "Pineapple" , "Orange", "Lemon", "Apple"];

let fruit1 = fruits[0];
let fruit2 = fruits[1];
let fruit3 = fruits[2];

console.log(fruit1, fruit2, fruit3); //"Mango" "Pineapple" "Orange"
```

This is like repeating the same thing over and over which could become cumbersome. Let's see how this could be distructured to

```
let [fruit1, fruit2, fruit3] = fruits;

console.log(fruit1, fruit2, fruit3); //"Mango" "Pineapple" "Orange"
```

You might be wondering how you could skip data if you just want to print the first and final fruits, or the second and fourth fruits. You would use commas as follows:

```
const [fruit1 ,,,, fruit5] = fruits;
const [,fruit2 ,, fruit4,] = fruits;
```

## Object destructuring

Let's now see how we could destructure an object – because in React you will be doing a lot of object descructuring.

Suppose we have an object of user which contains their firstname, lastname, and lots more,

```
const Susan = {
  firstName: "Susan",
  lastName: "Steward",
  age: 14,
  hobbies: {
    hobby1: "singing",
    hobby2: "dancing"
  }
};
```

In the old way, getting these data could be stressful and full of

```
const firstName = Susan.firstName;
const age = Susan.age;
const hobby1 = Susan.hobbies.hobby1;

console.log(firstName, age, hobby1); //"Susan" 14 "singing"
```

but with destructuring its a lot easier:

```
const {firstName, age, hobbies:{hobby1}} = Susan;

console.log(firstName, age, hobby1); //"Susan" 14 "singing"
```

We can also do this within a function:

```
function individualData({firstName, age, hobbies:{hobby1}}){
  console.log(firstName, age, hobby1); //"Susan" 14 "singing"
}
individualData(Susan);
```

You can read more about destructuring Arrays and Objects here.

# Rest and Spread Operators in JavaScript

JavaScript spread and rest operators use three dots . . . The rest operator gathers or collects items – it puts the "rest" of some specific user-supplied values into a JavaScript array/object.

Suppose you have an array of fruits:

We could destructure to get the first and second fruits and then place the“rest” of the fruits in an array by making use of the rest operator.

```
const [firstFruit, secondFruit, ...rest] = fruits

console.log(firstFruit, secondFruit, rest); //"Mango" "Pineapple" ["Or
```

Looking at the result, you'll see the first two items and then the third item is an array consisting of the remaining fruits that we didn't destructure. We can now conduct any type of processing on the newly generated array, such as:

```
const chosenFruit = rest.find((fruit) => fruit === "Apple");

console.log(`This is an ${chosenFruit}`); //"This is an Apple"
```

It's important to bear in mind that this has to come last always (placement is very important).

We've just worked with arrays – now let's deal with objects, which are absolutely the same.

Assume we had a user object that has their firstname, lastname, and a lot more. We could destructure it and then extract the remainder of the data.

```
  firstName: "Susan",
  lastName: "Steward",
  age: 14,
  hobbies: {
    hobby1: "singing",
    hobby2: "dancing"
  }
};

const {age, ...rest} = Susan;
console.log(age, rest);
```

This will log the following result:

```
14
{
firstName: "Susan" ,
lastName: "Steward" ,
hobbies: {...}
}
```

Let's now understand how the spread operator works, and finally summarize by differentiating between both operators.

## Spread operator

The spread operator, as the name implies, is used to spread out array items. It gives us the ability to get a list of parameters from an array. The spread operator has a similar syntax to the rest operator, except it operates in the opposite direction.

**Note:** A spread operator is effective only when used within array literals, function calls, or initialized properties objects.

```
let pets= ["cat", "dog" , "rabbits"];

let carnivorous = ["lion", "wolf", "leopard", "tiger"];
```

You might want to combine these two arrays into just one animal array. Let's try it out:

```
let animals = [pets, carnivorous];

console.log(animals); //[["cat", "dog" , "rabbits"], ["lion", "wolf",
```

This is not what we want – we want all the items in just one single array. And we can achieve this using the spread operator:

```
let animals = [...pets, ...carnivorous];

console.log(animals); //["cat", "dog" , "rabbits", "lion", "wolf", "le
```

This also works with objects. It is important to note that the spread operator cannot expand the values of object literals, since a properties object is not an iterable. But we can use it to clone properties from one object into another.

For example:

```
let name = {firstName:"John", lastName:"Doe"};
```

You can read more on JavaScript Spread and Rest operators here.

# Unique Value - Set() in JavaScript

Recently, I was try to create a categories tab for an application where I needed to fetch the categories value from an array.

```
let animals = [
  {
    name:'Lion',
    category: 'carnivore'
  },
  {
    name:'dog',
    category:'pet'
  },
  {
    name:'cat',
    category:'pet'
  },
  {
    name:'wolf',
    category:'carnivore'
  }
]
```

The first thing was to loop through the array, but I got repeated values:

This meant that I needed to set up a condition to avoid repetition. It was a little bit tricky until I came across the `set()` constructor/object provided by ES6 :).

A set is a collection of items which are unique, that is no element can be repeated. Let's see how we can implement this easily.

```
//wrap your iteration in the set method like this
let category = [...new Set(animals.map((animal)=>animal.category))];

console.log(category); ////["carnivore" , "pet"]
```

**NB:** I decided to spread the values into an array. You can read more on unique values here.

# Dynamic Object keys in JavaScript

This enables us to add object keys using square bracket notation. This may not make sense to you right now, but as you continue learning React or begin working with teams, you may come across it.

In JavaScript, we know that objects are often made up of properties/keys and values, and we may use the dot notation to add, edit, or access some value(s). As an example:

```
let lion = {
```

```
lion.baby = 'cub';
console.log(lion.category); // carnivore
console.log(lion); // { category: "carnivore" , baby: "cub" }
```

We also have the option of using square bracket notation, which is utilized when we need **dynamic object keys.**

**What do we mean by dynamic object keys?** These are keys that might not follow the standard naming convention of properties/keys in a object. The standard naming convention only permits camelCase and snake_case, but by using square bracket notation we can solve this problem.

For example, suppose we name our key with a dash in between words, for example ( `lion-baby` ):

```
let lion = {
  'lion-baby' : "cub"
};

// dot notation
console.log(lion.lion-baby); // error: ReferenceError: baby is not def
// bracket notation
console.log(lion['lion-baby']); // "cub"
```

You can see the difference between the dot notation and the bracket notation. Let's see other examples:

```
let category = 'carnivore';
let lion = {
  'lion-baby' : "cub",
```

You can also perform more complex operations by using conditions within the square bracket, like this:

```
const number = 5;
const gavebirth = true;

let animal = {
  name: 'lion',
  age: 6,
  [gavebirth && 'babies']: number
};

console.log(animal); // { name: "lion" , age: 6 , babies: 5 }
```

You can read more about this here.

# reduce() in JavaScript

This is arguably the most powerful array function. It can replace the `filter()` and `find()` methods and is also quite handy when doing `map()` and `filter()` methods on large amounts of data.

When you chain map and filter method together, you wind up doing the work twice – first filtering every single value and then mapping the remaining values. On the other hand, `reduce()` allows you to filter and map in a single pass. This method is powerful, but it's also a little more sophisticated and tricky.

We iterate over our array and then obtain a callback function, which is similar to `map()`, `filter()`, `find()`, and the others. The main

Another thing to keep in mind about the reduce() method is that we are passing in two arguments, which has not been the case since you began reading this tutorial.

The first argument is the sum/total of all computations, and the second is the current iteration value (which you will understand shortly).

For example, suppose we have a list of salaries for our staff:

```
let staffs = [
  { name: "Susan", age: 14, salary: 100 },
  { name: "Daniel", age: 16, salary: 120 },
  { name: "Bruno", age: 56, salary: 400 },
  { name: "Jacob", age: 15, salary: 110 },
  { name: "Sam", age: 64, salary: 500 },
  { name: "Dave", age: 56, salary: 380 },
  { name: "Neils", age: 65, salary: 540 }
];
```

And we want to calculate a 10% tithe for all staff. We could easily do this with the reduce method, but before doing that let's do something easier: let's calculate total salary first.

```
const totalSalary = staffs.reduce((total, staff) => {
  total += staff.salary;
  return total;
},0)
console.log(totalSalary); // 2150
```

NB: We passed a second argument which is the total, it could be anything – for example a number or object.

before adding them up.

```
const salaryInfo = staffs.reduce(
  (total, staff) => {
    let staffTithe = staff.salary * 0.1;
    total.totalTithe += staffTithe;
    total['totalSalary'] += staff.salary;
    return total;
  },
  { totalSalary: 0, totalTithe: 0 }
);

console.log(salaryInfo); // { totalSalary: 2150 , totalTithe: 215 }
```

**Gotcha:** We used an object as the second argument and we also used dynamic object keys. You can read more about the reduce method here.

# Optional chaining in JavaScript

Optional chaining is a safe way to access nested object properties in JavaScript rather than having to do multiple null checks when accessing a long chain of object properties. It is a new feature introduced in ES2020.

For example:

```
let users = [
{
    name: "Sam",
    age: 64,
    hobby: "cooking",
    hobbies: {
```

```
  { name: "Bruno", age: 56 },
  { name: "Dave", age: 56, hobby: "Football" },
  {
    name: "Jacob",
    age: 65,
    hobbies: {
      hobb1: "driving",
      hobby2: "sleeping"
    }
  }
];
```

Suppose you are trying to get the hobbies from the array above. Let's try it out:

```
users.forEach((user) => {
  console.log(user.hobbies.hobby2);
});
```

When you look in your console, you'll notice that the first iteration was completed, but the second iteration had no hobby. So it had to throw an error and break out of the iteration – which meant it couldn't acquire data from other Objects in the array.

**Output:**

```
"sleeping"
error: Uncaught TypeError: user.hobbies is undefined
```

This error can be fixed with optional chaining, though there are

## Conditional rendering method:

```
users.forEach((user) => {
  console.log(user.hobbies && user.hobbies.hobby2);
});
```

## Optional chaining:

```
users.forEach((user) => {
  console.log(user ?.hobbies ?.hobby2);
});
```

Output:

```
"sleeping"
undefined
undefined
"sleeping"
```

This might not really make sense to you now, but by the time you are working on something bigger in the future, it'll fall into place! You can read more here.

# Fetch API & Errors in JavaScript

The fetch API, as the name implies, is used to get data from APIs. It is a browser API that allows you to use JavaScript to make basic AJAX (Asynchronous JavaScript and XML) requests.

configuration is fairly simple to grasp. The fetch API delivers a promise by default (I covered promises earlier in this article).

Let's see how to fetch data via the fetch API. We'll use a free API which contains thousands of random quotes:

```
fetch("https://type.fit/api/quotes")
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((err) => console.log(err));
```

What we did here was:

- **Line 1:** we got the data from the API, which returned a promise
- **Line 2:** We then got the `.json()` format of the data which is also a promise
- **Line 3:** We got our data which now returns JSON
- **Line 4:** We got the errors in case there are any

We will see how this can be done with async/await in the next section. You can read more about the fetch API here.

## How to Handle Errors in the Fetch API

Let's now take a look at how we can handle errors from fetch API without needing to depend on the catch keyword. The `fetch()` function will automatically throw an error for network errors but not for HTTP errors such as 400 to 5xx responses.

This is very simple to implement:

```
fetch("https://type.fit/api/quotes")
  .then((response) => {
    if (!response.ok) {
      throw Error(response.statusText);
    }
    return response.json();
  })
  .then((data) => console.log(data))
  .catch((err) => console.log(err));
```

You can read more about Fetch API errors here.

# Async/Await in JavaScript

Async/Await allows us to write asynchronous code in a synchronous fashion. This means that you don't need to continue nesting callbacks.

An async function **always** returns a promise.

You could be racking your brain wondering what the difference between synchronous and asynchronous means. Simply put, synchronous means that jobs are completed one after the other. Asynchronous means that tasks are completed independently.

Note that we always have async in front of the function and we can only use await when we have async. You will understand soon!

Let’s now implement the Fetch API code we worked on earlier using

```
const fetchData = async () =>{
  const quotes = await fetch("https://type.fit/api/quotes");
  const response = await quotes.json();
  console.log(response);
}

fetchData();
```

This is way more easier to read, right?

You might be wondering how we can handle errors with async/await. Yup! You use the try and catch keywords:

```
const fetchData = async () => {
  try {
    const quotes = await fetch("https://type.fit/api/quotes");
    const response = await quotes.json();
    console.log(response);
  } catch (error) {
    console.log(error);
  }
};

fetchData();
```

You can read more about async/await here.

# Conclusion

In this article, we have learned over 10 JavaScript methods and concepts that everyone should understand thoroughly before learning React.

learn React.

Suppose you are just getting started with JavaScript – I have curated an awesome list of resources that will help you learn JavaScript concepts and topics here. Don't forget to star and share! :).

---

**Joel Olawanle**

Frontend Developer & Technical Writer

---

If you read this far, tweet to the author to show them you care.

Tweet a thanks