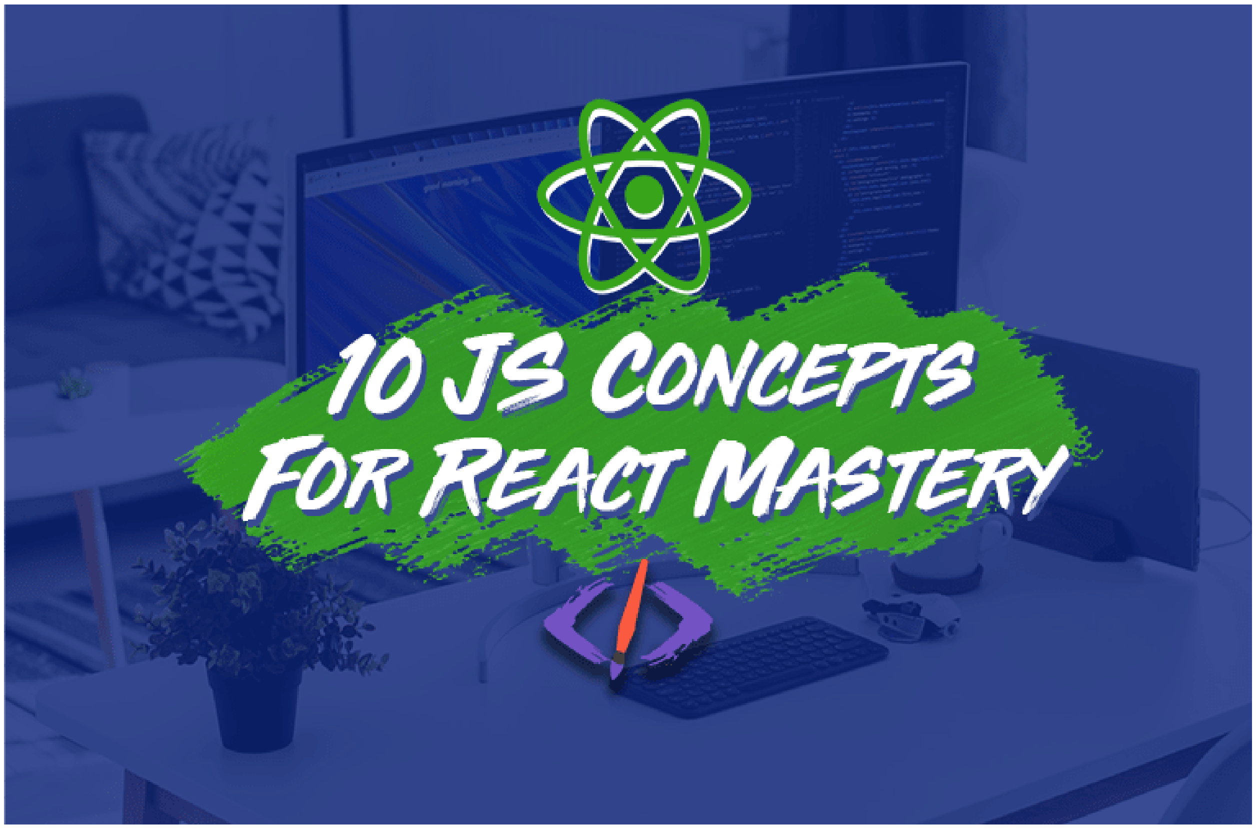
10 JavaScript Concepts for React Cheatsheet



let and const Variables

Why use let and const in JavaScript?

- The let and const keywords give us create more precise, predictable variables than those declared with var.
- let and const fix annoying problems of the var keyword such as hoisting
  - Hoisting is a phenomenon when variables can be used before they are created. It's a pretty strange behavior and this can cause subtle bugs in our code.
- let and const have block scoping

    - Variables declared with let and const within curly braces, that let / const variable only exists within that code block and not outside it.
    - Block scoping results in more predictable, less-error prone code by variables conflicting with each other less often
- let keyword is used for variables that need to be reassigned after being declared; const for variables that are not reassigned
    - const variables are not 'constant', that is, impossible to change. To be precise, const variables cannot be reassigned or given a different value after being created

```
let greeting;
const newUser = true;

if (newUser) {
  // let variables can be reassigned...
  greeting = "Nice to meet you!";
} else {
  greeting = "Welcome back!";
}

// ...while const variables cannot
newUser = false; // Uncaught TypeError: Assignment to constant variable.
```

How are let / const used in React?

- let / const are the preferred keywords for declaring any kind of variable in React due to the added predictability of how they function
    - Simply put, variables that should be reassigned are declared with let; variables that should not be reassigned and should only reference one thing are declared with const
- const is used to declare components as well when combined with arrow functions (we'll cover arrow functions later)
    - const is used for creating components because they will never to change to something else

```
// function component declared with const (arrow function)
const Header = () => {
  // local variable declared with const
  const username = "Bob";
  return <header>Welcome back, {username}!</header>;
};
```

Template Literals

Why use template literals in JavaScript?

- Template literals are much more powerful, dynamic strings than basic JavaScript strings using single or double quotes
  - Interpolating or inserting values into strings is much easier; uses the `${}` syntax to insert valid JS expressions
  - No need to use the + operator to concatenate or combine strings as before
  - Easier to write multiline strings
    - No need to write new lines with the newline character (`\\n`) or carriage return (`\\r`)
  - Can use nested quotes (with single or double quotes) within template literals strings without errors

```
const username = "Fred";
// connecting strings with the + operator is hard to read and not intuitive
const greeting = "Hi " + username + ", how are you?";

// template literals (``) are a breeze and much easier to write
const username = "Anna";
// dynamic values are inserted with the ${} syntax
const greeting = `Hi ${username}, how are you?`;
```

How are template literals used in React?

- Template literals are used for string interpolation as well as things like dynamically setting the string value of props of components or attributes of elements.
  - Template literals are great for dynamically calculating the styles of elements based on different conditions, such as the example below:

```
function UserCard({ id, name }) {
  // if user id is an odd number...
  const isOddUser = id % 2 !== 0;
  // ...give them a dark background
  return <div className={idOddUser ? "dark-bg" : ""}>{name}</div>;
}

<UserCard id={1} name="Bob" />; // displays UserCard with dark background applied
```

Arrow Functions

Why use arrow functions in JavaScript?

- Arrow functions allow us to write functions in a shorter syntax, resulting in less boilerplate code for our functions
    - Allows us to replace the return keyword and the function body (curly braces) with a fat arrow syntax: ⇒
    - It also makes working with objects and classes easier due to how it handles the this keyword.
- Arrow functions come with 3 shorthands, meaning even shorter functions
    - The parentheses around parameters can be removed if there is just one
    - Curly braces for function body can be removed entirely
    - No need for the return keyword; arrow functions have an implicit return (they return by default without curly braces)

```
// normal function
function capitalize(word) {
  return word.toUpperCase();
}

// arrow function
const capitalize = () => {
  return word.toUpperCase();
};

// arrow function with all 3 shorthands
const capitalize = (word) => word.toUpperCase();
```

How are arrow functions used in React?

- Arrow functions are used anywhere we can create a function in JavaScript

- They are used most often to create function components as well as for higher-order array methods like .map() or .filter()

```
const UserList = ({ users }) => {
  return (
    <ul>
      {users.map((user, index) => (
        <UserCard key={index} {...user} />
      ))}
    </ul>
  );
};
```

Powerful Array Methods (.map(), .filter(), .reduce(), etc.)

Why use powerful array methods in JavaScript?

- As compared to using a for-loop to loop over arrays, array methods like map, filter, and reduce enable us to loop over arrays with a certain goal in mind
  - .map() - allows us to transform each element of an array
  - .filter() - allows us to filter out items from arrays that don't meet a given condition
  - .reduce() - allows us to transform an entire array in whatever way we choose (even into other data types)
- These array methods are shorter and more declarative (more clearly express what they do) than a normal for-loop
  - These are even shorter when arrow functions are used as the callback to each of these methods (see example)

```
// Goal: turn array of users into array of usernames
const users = [
  { name: "Bob", id: 1 },
  { name: "Jane", id: 2 },
  { name: "Fred", id: 3 },
];
const usernames = [];

// for-loop
for (let i = 0; i < users.length; i++) {
  usernames[i] = users[i];
}
```

```
usernames; // ["Bob", "Jane", "Fred"]

// .map() - concise + readable
const usernames = users.map((user) => user.username);

usernames; // ["Bob", "Jane", "Fred"]
```

How are powerful array methods used in React?

- Methods like .map(), .filter(), .reduce() can be used wherever we need to transform or shape our array data
- Most of the time, you'll be using these methods to dynamically display components or elements using JSX
  - These methods can be chained together to perform one transformation after another

```
function UserList() {
  const users = [
    { name: "Bob", id: 1 },
    { name: "Jane", id: 2 },
    { name: "Fred", id: 3 },
  ];

  // filter out user with id of 2, then map over the rest to display their names
  return (
    <ul>
      {users
        .filter((user) => user.id !== 2)
        .map((user) => (
          <li key={id}>{user.name}</li>
        ))}
    </ul>
  );
}
```

Destructuring

Why use destructuring in JavaScript?

- Destructuring enables us to turn object key-value pairs into variables
  - Destructuring is a great convenience because we often don't need to reference an entire object whenever we want to use it.

    - With destructuring, we can just create what looks to some like a reversed version of the object and pluck off what ever values we need, making them independent variables
    - Allows us to reduce repetition by not referencing a single object every time we need a value from it.
- Note that destructuring can be also done with arrays as well as normal objects

```
const user = {
  name: "Reed",
  username: "ReedBarger",
  email: "reed@gmail.com",
  details: {
    title: "Programmer",
  },
};

// object property access without destructuring
console.log(`${user.name}, ${user.email}`); // logs: Reed, reed@gmail.com

// object destructuring for less repetition
const { name, email } = user;

console.log(`${name}, ${email}`); // logs: Reed, reed@gmail.com

// object destructuring with nested object "details"
const {
  username,
  details: { title },
} = user;

console.log(`${username}, ${title}`); // logs: ReedBarger, Programmer
```

How is destructuring used in React?

- Destructuring is used most often for getting individual values from the props object
    - Most often, we don't need the entire props object, especially if we pass down only one prop to a given component. Instead of referencing 'props', we can just destructure it to get the props data as individual variables in our component.

```
function App() {
  return (
    <div>
```

```
      <h1>All Users</h1>
      <UserList users={['Bob', 'Jane', 'Fred']} />
    </div>
    );
  }
}

function UserList({ users }) {
  return (
    <ul>
      {users.map((user, index) => (
        <li key={index}>{user}</li>
      ))}
      </ul>
  );
}
```

Default parameters

Why use default parameters in JavaScript?

- To handle the event that a function that doesn't have values passed to it that it needs as arguments
- Default parameters helps us prevent errors and have more predictable code by giving arguments default values (with the equals sign) if none are provided

```
// without default parameters
function sayHi(name) {
  return "Hi" + name;
}
sayHi(); // "Hi undefined"

// with default parameters
function sayHi(name = 'Bob') {
  return "Hi" + name;
}
sayHi(); // "Hi Bob"

// with default parameters using an arrow function
const sayHi = (name = 'Jane') => "Hi" + name;
sayHi(): // "Hi Jane"
```

How are default parameters used in React?

- Default parameters are often used in the case of props

- In this example, we are using object destructuring to grab a prop called 'username' from the props object. But no prop value has been passed, so we set a default parameter value of 'guest' and our component can still works.

```
const Header = ({ username = "guest" }) => {
  return <header>Welcome, {username}!</header>;
};

<Header />; // displays: Welcome, guest!
```

Spread Operator

Why use the spread operator in JavaScript?

- The spread operator allows us to "spread" objects (their key-value pairs) into new ones
    - Spread operator only works when creating a new object or array
- Spread operator is great for creating new objects by merging their properties together
    - Whenever an object or array is spread into a new object or array, a shallow copy of it is made, which helps prevent errors
- The spread operator can be used with both objects and arrays

```
// Merge default empty data with user data from a sign up form with spread operator
const user = {
  name: "",
  email: "",
  phoneNumber: "",
};

const newUser = {
  name: "ReedBarger",
  email: "reed@gmail.com",
};

/*
  the object that is spread in last overwrites the previous object's values
  if the properties have the same name
*/
const mergedUser = { ...user, ...newUser };
mergedUser; // { name: "ReedBarger", email: "reed@gmail.com", phoneNumber: "" };
```

How is the spread operator used in React?

- The spread operator is used a great deal for dynamically creating new objects and arrays in an immutable fashion
  - Often used in common React libraries like Redux to make sure data is changed in a predictable manner
- Specific to React, however, the spread operator is used to easily pass down all of an object's data as individual props (without having to pass down each prop one-by-one)
  - How does this work? We can spread an object into a component because the props object is what we're spreading into.

```
function App() {
  const name = {
    first: "Reed",
    last: "Barger"
  };

  return (
    <div>
    {/* spread in name object, as compared to:
      <UserGreeting
        first={name.first}
        last={name.last}
      />
    */}
      <UserGreeting {...name} />
    </div>
    );
  }
}

function User({ first, last }) {
  return (
    <p>Hi, {first} {last}</p>
  );
}
```

Short Conditionals

Why use short conditionals in JavaScript?

- There is a shorter way of writing an if-else conditional in JavaScript called the ternary.

- As compared to an if-else statement, ternaries are expressions. This enables us a lot more flexibility and allows us to use ternaries wherever an expression can be evaluated (such as in the `${}` of a template literal)
- Ternaries shouldn't always be preferred over if-else statements, especially if there is more than one condition to evaluated. In that case, ternaries are hard to read.

```
let age = 26;
let greeting;

// if-else conditions are sometimes unnecessary, especially for situations like
// this where we are just giving a variable one or another value based on
// a condition
if (age > 18) {
  greeting = "Hello, fellow adult";
} else {
  greeting = "Hey kiddo";
}

// ternaries do the same thing, but greatly shorten our code
const greeting = age > 18 ? "Hello, fellow adult" : "Hey kiddo";
greeting; // 'Hello, fellow adult';
```

How are short conditionals used in React?

- The benefit of ternaries is that they allow us to more succinct write if-else conditionals in JSX, where if a condition is true, we show one thing, if false, a different thing
- An alternative to the ternary, where if we just want to show one thing if a condition is true, otherwise nothing is the && operator. If the condition is true, return that thing, otherwise, nothing will be shown

```
const Navbar = () => {
  const isAuth = true;

  return (
      <div>
      {/* if user is authenticated, show auth links, otherwise a login link */}
      {isAuth ? <AuthLinks /> : <Login />}
      {/* if user is authenticated, show their profile. If not, nothing. */}
      {isAuth && <UserProfile/>}
    </div>
  );
}
```

ES Modules

Why use ES Modules in JavaScript?

- ES Modules allow us to conveniently share code across multiple files in our application
- We export things that we want to pass to other files in our app, primarily variables and functions and import whatever things (that have been exported) within files that need these things
- We can export / import multiple things within curly braces (and the keywords export / import) or just one thing with no curly braces (with keywords export default and import)
- We usually use modules to make our code more modular, to only write code where we need it, not to put everything in a single file. See below how we have a getLocalTime function that lives in its own file (of the same name), but we bring into app.js to use it there.

```
// utils/getLocalTime.js
const getLocalTime = () => new Date().toLocaleTimeString();

export default getLocalTime;

// app.js
import getLocalTime from "./utils/getLocalTime.js";

const App = () => {
  return (
    <div>
      <header>The time is {getLocalTime()}</header>
      ...
    </div>
  );
};
```

How are ES Modules used in React?

- ES Modules are used extensively in React to build our applications effectively.
- ES Modules are used to import both relative (local files) and absolute imports (packages like React)

    - Local files that have components in them are capitalized (like the component name itself)
- We can export and import virtually anything, not just JavaScript (variables and functions), but also CSS and image files, etc
- Additionally, in React we often don't have to add the extension at the end whenever we are importing JavaScript. We only have to add the extension when importing other types of files into our JavaScript, for example:

```
// App.js
const App = () => <div>hello world!</div>

// styles.css
html, body {
  margin: 0;
  padding: 0;
}

h1 {
  color: cornflowerblue;
}

// index.js
import React from 'react';
import './styles.css'

import ReactDOM from "react-dom";

import App from "./App";

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Promises + Async / Await

Why use promises + async / await in JavaScript?

- Certain things in JavaScript don't happen right away, they take an unknown amount of time (for example, a setTimeout(), an event listener or a network request with the fetch API)
- Promises are a means of making asynchronous code in JavaScript predictable. They help us resolve our async code and give us a means of handling the case that it was successful with a .then() callback and a .catch() callback if there was an error in performing that async operation

- Async / await is an improved syntax for working with promises that makes our async code looks synchronous and is the most readable way of working with async code

```
// async code; 'done' is logged after position data, even though 'done' is supposed
// to be executed later in our code
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log(position);
  },
  (error) => {
    console.error(error);
  }
);
console.log("done");

// async code handled with a promise; we get the result we want - position data
// is logged, then 'done' is logged
const promise = new Promise((resolve, reject) => {
  navigator.geolocation.getCurrentPosition(resolve, reject);
});

promise
  .then((position) => console.log(position))
  .catch((error) => console.error(error))
  .finally(() => console.log("done"));

// async code with async/await looks like synchronous code; the most readable way
// of  working with promises
async function getPosition() {
  // async/await works in functions only (for now)
  const result = await new Promise((resolve, reject) => {
    navigator.geolocation.getCurrentPosition(resolve, reject);
  });
  const position = await result;
  console.log(position);
  console.log("done");
}

getPosition();
```

How are promises + async / await used in React?

- Promises + async / await are used by far the most for making network requests, such as a request to a REST API or a GraphQL API
- Many libraries such as the fetch API or axios use promises to resolve these requests that take an unknown period of time to complete, and promises and

async / await are used extensively with these libraries that are used to make network requests

- See the app in the cheatsheet for a practical example of how data is fetched from an API and resolved with async / await

```
// fetching data from an API with basic promise syntax (notice the use of arrow functions)
window
  .fetch("<http://jsonplaceholder.typicode.com/posts>")
  .then((response) => response.json())
  .then((data) => console.log(data));

// fetching same data from API with async/await
async function getPostData() {
  const response = await window.fetch(
    "<http://jsonplaceholder.typicode.com/posts>"
  );
  // we need to resolve two promises using await to get the final data
  const data = await response.json();
  console.log(data);
}
getPostData();
```