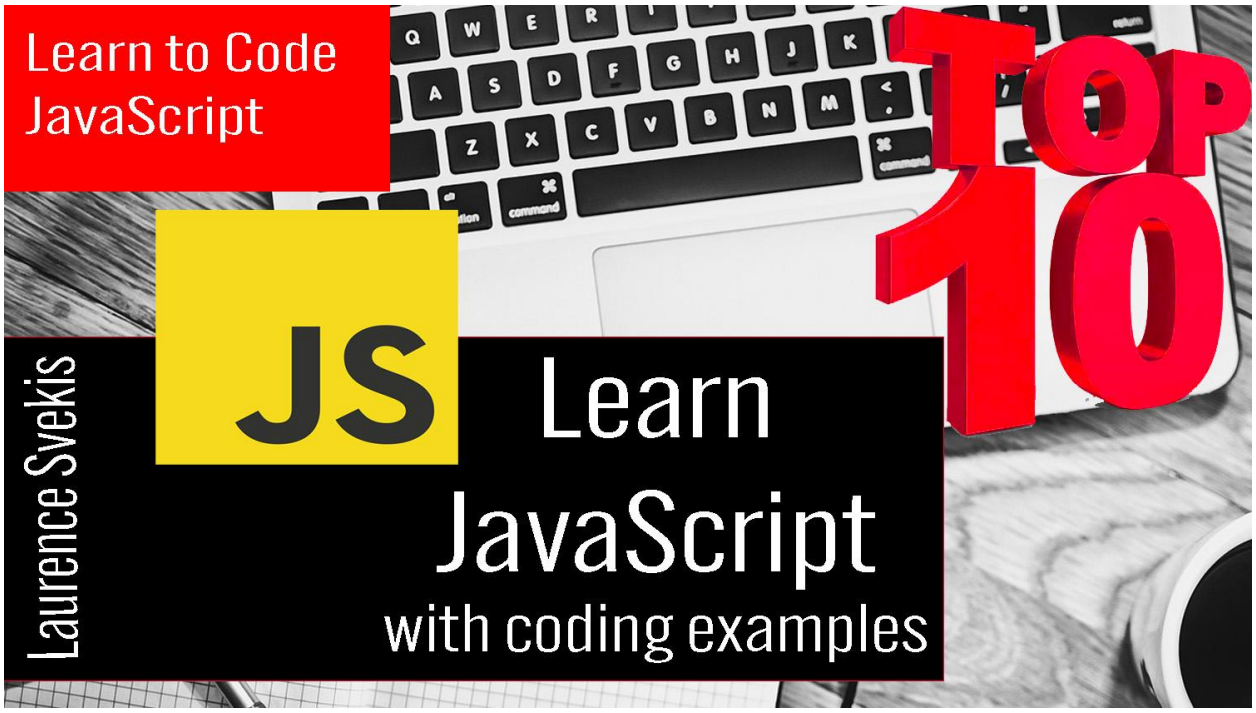
Top 10 Coding Examples and Tips for JavaScript Code



Use spread operator

Use spread operator to combine arrays:

```
const a = [1, 2, 3];
const b = [4, 5, 6];
const c = [...a, ...b]; // [1, 2, 3, 4, 5, 6]
```

Use spread operator to combine arrays: The spread operator (...) can be used to combine arrays into a new array. The operator "spreads" the elements of the original arrays into a new array. For example, in the following code, two arrays a and b are combined into a new array c:

Use of default parameters

Make use of default parameters to handle missing arguments:

```
const greet = (name = 'stranger') =>
console.log(`Hello, ${name}!`);
greet(); // Hello, stranger!
greet('John'); // Hello, John!
```

Make use of default parameters to handle missing arguments: Default parameters allow you to provide a default value for a

function argument in case the argument is not passed when the function is called. This can be useful for handling missing arguments and preventing errors.

Use rest operator

Use rest operator to pass multiple arguments as an array:

```
const add = (...numbers) => numbers.reduce((a, b) => a
+ b, 0);
console.log(add(1, 2, 3, 4, 5)); // 15
```

Use rest operator to pass multiple arguments as an array: The rest operator (...) can be used to gather all remaining arguments into an array. This is useful for passing a variable number of arguments to a function.

Use object literals

Use object literals to create objects:

```
const name = 'John Doe';
const age = 30;
const person = { name, age };
```

Use object literals to create objects: Object literals are a concise and convenient way to create objects in JavaScript. The syntax is similar to an array literal, but with curly braces ({}) instead of square brackets ([]). You can also use property value shorthand to create properties using the same name as the variable.

Use destructuring with rest operator

Use destructuring with rest operator to extract remaining values:

```
const colors = ['red', 'green', 'blue', 'yellow',
'orange'];
const [first, second, ...rest] = colors;
console.log(first); // red
console.log(second); // green
console.log(rest); // [blue, yellow, orange]
```

Use destructuring with rest operator to extract remaining values: Destructuring is a powerful feature in JavaScript that allows you to extract values from arrays and objects and assign them to variables. You can use the rest operator (...) in combination with destructuring to extract remaining values from an array.

Use of async/await

Make use of async/await for asynchronous programming:

```
const fetchData = async () => {
  try {
    const response = await
fetch('https://jsonplaceholder.typicode.com/posts');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
};
fetchData();
```

Make use of async/await for asynchronous programming: The async/await syntax provides a convenient way to write asynchronous code that is easier to read and debug than traditional callback-based code. An async function returns a Promise and can be awaited to pause execution until the Promise is resolved.

Use destructuring with default values

Use destructuring with default values to handle missing properties:

```
const person = {
  name: 'John Doe'
};
const { name, age = 30 } = person;
console.log(name); // John Doe
console.log(age); // 30
```

Use destructuring with default values: Destructuring can also be used to provide default values for variables in case the values are undefined. For example:

```
const getUser = ({ name = 'stranger', age = 'unknown' }
= {}) =>
  console.log(`Name: ${name}, Age: ${age}`);
getUser({ name: 'John Doe', age: 30 }); // Name: John
Doe, Age: 30
getUser(); // Name: stranger, Age: unknown
```

Use named exports

Use named exports to export multiple values from a module:

```
// utils.js
export const PI = 3.14;
export const add = (a, b) => a + b;

// main.js
import { PI, add } from './utils.js';
console.log(PI); // 3.14
console.log(add(1, 2)); // 3
```

Use object spread operator

Use object spread operator to merge objects:

```
const a = { name: 'John Doe', age: 30 };
const b = { job: 'Software Engineer' };
const c = { ...a, ...b };
console.log(c); // { name: 'John Doe', age: 30, job:
'Software Engineer' }
```

Use try/catch

Make use of try/catch blocks to handle errors: Try/catch blocks provide a convenient way to handle errors in JavaScript. A try block is used to enclose the code that might throw an error, and a catch block is used to catch the error and handle it. For example:

```
const divide = (a, b) => {
```

```
  try {
    if (b === 0) {
      throw new Error('Cannot divide by zero');
    }
    return a / b;
  } catch (error) {
    console.error(error.message);
  }
};
console.log(divide(10, 5)); // 2
console.log(divide(10, 0)); // Cannot divide by zero
```

Use ternary operator

Use ternary operator for simple conditional statements: The ternary operator (?) provides a shorthand way to write simple if/else statements. The operator takes three operands: the condition to be tested, the expression to be returned if the condition is true, and the expression to be returned if the condition is false.

```
const isPositive = (number) => (number >= 0 ?
'positive' : 'negative');
console.log(isPositive(10)); // positive
console.log(isPositive(-10)); // negative
```

Use named export/import

Use named export/import to manage modular code: Named exports and imports allow you to organize and reuse code by breaking it up into separate modules. You can use named exports

to export multiple values from a module, and named imports to import specific values into another module.

```
// utils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// index.js
import { add, subtract } from './utils';
console.log(add(10, 5)); // 15
console.log(subtract(10, 5)); // 5
```