strict mode example

https://youtu.be/rbPKM4EkrGU

Use strict mode to enforce modern JavaScript syntax and catch errors early:

```
'use strict';
```

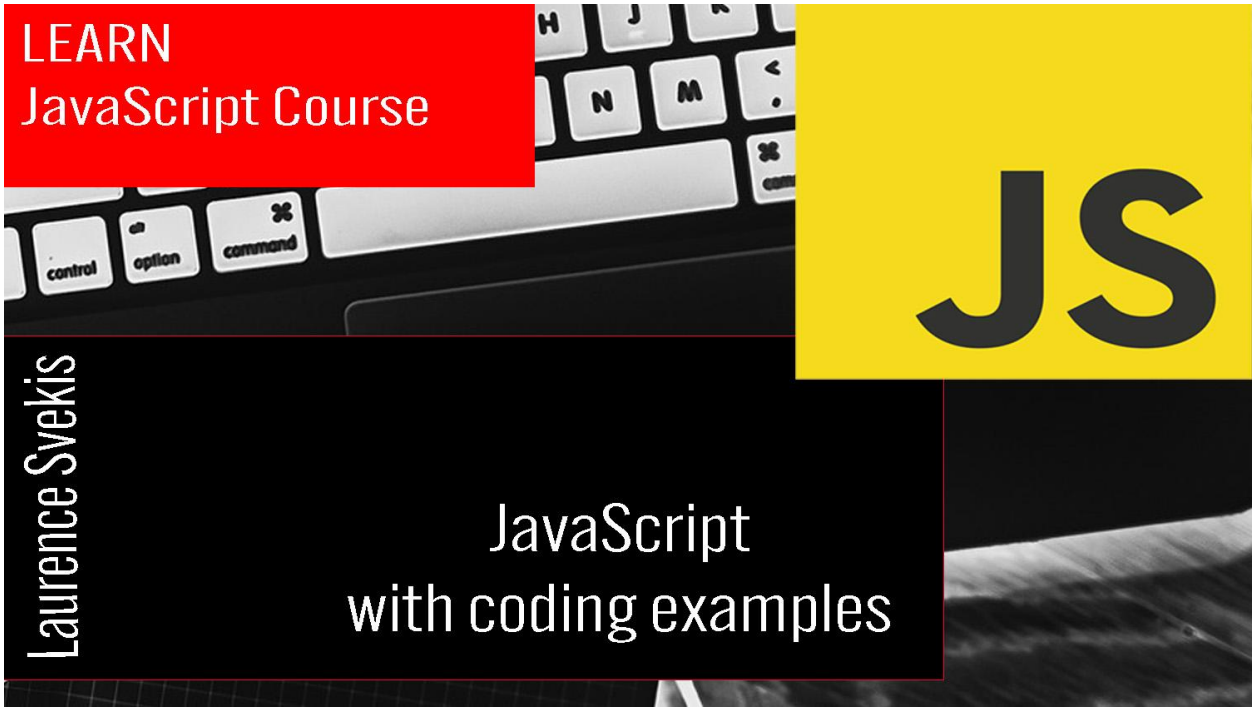
"use strict" is a directive in JavaScript that enables "strict mode". Strict mode is a way to opt in to a "secure" version of JavaScript. When in strict mode, JavaScript will validate your code more strictly and throw errors when it encounters "unsafe" or undefined behaviors.

Here's an example of using "use strict":

```
'use strict';

x = 10; // ReferenceError: x is not defined

(function() {
  'use strict';

  y = 20; // ReferenceError: y is not defined
})();
```

In the above example, both x and y are not declared with var, let, or const and are therefore considered global variables. When the code is run in strict mode, a ReferenceError is thrown, indicating that the variables are not defined.

It is recommended to use "use strict" at the beginning of every JavaScript file to ensure that the code is executed in strict mode and to take advantage of its security benefits.



Use const and let

https://youtu.be/yypZvAT6Gd0

Always declare variables with const or let, rather than var:

```
// Use let
let name = 'John Doe';

// Use const
const PI = 3.14;
```

var, let, and const are used to declare variables.

var is the oldest way to declare variables in JavaScript and has function scope. This means that a variable declared with var inside a function is only accessible within that function. If a var variable is declared outside of a function, it is considered a global variable and can be accessed from anywhere in the code.

```
var x = 10; // global scope

function example() {
  var y = 20; // function scope
}

console.log(x); // 10
console.log(y); // ReferenceError: y is not defined
```

let and const are introduced in ES6 and are block-scoped, meaning that a variable declared with let or const is only accessible within the block it was declared in.

```
let x = 10; // global scope

if (true) {
  let y = 20; // block scope
}

console.log(x); // 10
console.log(y); // ReferenceError: y is not defined
```

The difference between let and const is that let allows you to reassign the variable to a different value, while const does not.

Once a variable is declared with const, its value cannot be changed.

```
const x = 10;
x = 20; // TypeError: Assignment to constant variable.

const y = [1, 2, 3];
y.push(4); // allowed, since arrays are mutable in
JavaScript
```

It is recommended to use const by default, and only use let when you need to reassign the variable. This helps to ensure that your code is more readable, predictable, and secure.

Use Arrows functions

https://youtu.be/4jEFveT3IFw

Use arrow functions instead of function for cleaner and concise code:

```
// Function expression
const multiply = (a, b) => a * b;

// Implicit return
const square = x => x * x;
```

Arrow functions, also known as "fat arrow" functions, are a shorthand way to declare anonymous functions in JavaScript. They were introduced in ECMAScript 6 (ES6) and are a more

concise and less verbose alternative to regular function expressions.

Here's an example of a regular function expression:

```
const sum = function(a, b) {
  return a + b;
};

console.log(sum(1, 2)); // 3
```

And here's the equivalent arrow function:

```
const sum = (a, b) => {
  return a + b;
};

console.log(sum(1, 2)); // 3
```

Arrow functions have several advantages over regular function expressions:

Shorter syntax: Arrow functions are shorter and more concise, making your code easier to read and write.

Implicit return: If an arrow function only has one expression, the value of that expression is returned automatically. You can omit the return keyword and the curly braces.

```
const sum = (a, b) => a + b;

console.log(sum(1, 2)); // 3
```

Lexical this: Arrow functions have a lexical this binding, meaning that this refers to the surrounding scope and is not affected by the context in which the function is executed.

```
const person = {
  name: 'John Doe',
  greet: function() {
    setTimeout(function() {
      console.log(`Hello, my name is ${this.name}`);
    }, 1000);
  }
};
```

person.greet(); // Hello, my name is undefined
With arrow functions, this refers to the outer person object and the code works as expected:

```
const person = {
  name: 'John Doe',
  greet: function() {
    setTimeout(() => {
      console.log(`Hello, my name is ${this.name}`);
    }, 1000);
  }
};
```

person.greet(); // Hello, my name is John Doe
In conclusion, arrow functions are a powerful and convenient feature in JavaScript that allow you to write more concise, expressive, and predictable code.