SECTION 3: JavaScript Fundamentals - Part 2

Lecture 2: Activating Strict Mode

JavaScript's strict mode is an important feature that helps developers avoid accidental errors. To activate it, you simply add `'use strict';` at the very beginning of your JavaScript file, before any other code (comments are allowed).

```
'use strict';

// Now strict mode is activated for the entire script
```

Strict mode creates visible errors where JavaScript would otherwise fail silently. For example:

```
let hasDriversLicense = false;
const passTest = true;

if (passTest) hasDriverLicense = true; // Typo! Missing 's'
if (hasDriversLicense) console.log('I can drive 😃 ');
```

Without strict mode, this code would silently create a new global variable `hasDriverLicense` (with the typo) and the condition would never be true. With strict mode, JavaScript throws an error, making the bug obvious.

Strict mode also reserves certain keywords for future JavaScript features and prevents using them as variable names, like:

```
const interface = 'Audio'; // Error in strict mode
const private = 534; // Error in strict mode
```

Lecture 3: Functions

Functions are fundamental building blocks in JavaScript. They are reusable pieces of code that can be executed whenever needed.

The simplest form of a function is a **function declaration**:

```javascript
function logger() {
  console.log('My name is Jonas!');
}

// Calling/running/invoking the function
logger();
logger();
```

Functions can receive input data (parameters) and return output data:

```javascript
function fruitProcessor(apples, oranges) {
  console.log(apples, oranges);
  const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
  return juice;
}

// Using the function with arguments 5 and 0
const appleJuice = fruitProcessor(5, 0);
console.log(appleJuice);

// Reusing the function with different inputs
const appleOrangeJuice = fruitProcessor(2, 4);
console.log(appleOrangeJuice);
```

In this example:

- `apples` and `oranges` are parameters (placeholders)
- `5`, `0`, `2`, and `4` are arguments (actual values)
- The function processes these inputs and returns a string

Functions help us write DRY code (Don't Repeat Yourself), which is an important clean code principle.

Lecture 4: Function Declarations vs. Expressions

JavaScript offers multiple ways to create functions. The two main types are function declarations and function expressions.

Function Declaration:

```
function calcAge1(birthYear) {
  return 2037 - birthYear;
}

const age1 = calcAge1(1991);
```

Function Expression:

```
const calcAge2 = function (birthYear) {
  return 2037 - birthYear;
};

const age2 = calcAge2(1991);
console.log(age1, age2); // Both produce the same result
```

The main difference between them is that function declarations can be called before they're defined in the code (due to a process called hoisting), while function expressions cannot:

```
// This works
const age1 = calcAge1(1991);

function calcAge1(birthYear) {
  return 2037 - birthYear;
}

// This would cause an error if placed before the function expression
const age2 = calcAge2(1991);

const calcAge2 = function (birthYear) {
  return 2037 - birthYear;
};
```

Both have their place in JavaScript, and which one you use often comes down to personal preference.

Lecture 5: Arrow Functions

Arrow functions were introduced in ES6 as a shorter form of function expressions, especially useful for simple one-line functions:

```
// Arrow function with one parameter and implicit return
const calcAge3 = birthYear => 2037 - birthYear;
const age3 = calcAge3(1991);
console.log(age3);

// Arrow function with multiple parameters and multiple lines of code
const yearsUntilRetirement = (birthYear, firstName) => {
  const age = 2037 - birthYear;
  const retirement = 65 - age;
  return `${firstName} retires in ${retirement} years!`;
};

console.log(yearsUntilRetirement(1991, 'Jonas'));
```

Key points about arrow functions:

1. For one parameter and one-line body, you can omit parentheses around parameters and curly braces
2. With a one-liner without curly braces, the return is implicit
3. For multiple parameters, you need parentheses: `(param1, param2) => expression`
4. For multiple lines of code, you need curly braces and an explicit `return` statement
5. Arrow functions don't have their own `this` keyword (important for more advanced JavaScript)

Lecture 6: Functions Calling Other Functions

In JavaScript, it's common for one function to call another function. This allows us to break down complex problems into smaller, reusable parts.

```javascript
// Function to cut fruit into pieces
const cutPieces = function (fruit) {
  return fruit * 4;
};

// Function that uses the cutPieces function
const fruitProcessor = function (apples, oranges) {
  const applePieces = cutPieces(apples);
  const orangePieces = cutPieces(oranges);

  const juice = `Juice with ${applePieces} pieces of apple and
${orangePieces} pieces of orange.`;
  return juice;
};

console.log(fruitProcessor(2, 3));
```

In this example, the `fruitProcessor` function calls the `cutPieces` function twice. This approach follows the DRY principle - if we later need to change how fruits are cut (e.g., cut into 3 pieces instead of 4), we only need to change it in one place.

Lecture 7: Reviewing Functions

Let's review the key concepts we've learned about functions:

```
const calcAge = function (birthYear) {
  return 2037 - birthYear;
};

const yearsUntilRetirement = function (birthYear, firstName) {
  const age = calcAge(birthYear);
  const retirement = 65 - age;

  if (retirement >= 0) {
    return `${firstName} retires in ${retirement} years!`;
  } else {
    return `${firstName} is already retired 🎉`;
  }
};

console.log(yearsUntilRetirement(1991, 'Jonas'));
console.log(yearsUntilRetirement(1969, 'Mark'));
```

Important points about functions:

1. Functions can call other functions (like `calcAge` inside `yearsUntilRetirement`)
2. Parameters are like local variables that only exist inside the function
3. Functions with the same parameter names don't interfere with each other
4. The `return` statement immediately exits the function - any code after it won't run
5. Functions can return different values based on conditions (like our retirement example)
6. Functions can be stored as values and accessed through the console

Functions are fundamental in JavaScript and understanding them well is critical for becoming proficient.

Lecture 9: Introduction to Arrays

Arrays allow us to store multiple related values in a single variable, rather than creating separate variables for each value:

```javascript
// Instead of this:
const friend1 = 'Michael';
const friend2 = 'Steven';
const friend3 = 'Peter';

// We can use an array:
const friends = ['Michael', 'Steven', 'Peter'];
console.log(friends);

// Alternative syntax (less common)
const years = new Array(1991, 1984, 2008, 2020);
```

Accessing array elements (using zero-based indexing):

```javascript
console.log(friends[0]); // Michael
console.log(friends[2]); // Peter

// Getting the array length
console.log(friends.length); // 3

// Getting the last element
console.log(friends[friends.length - 1]); // Peter
```

Arrays can be mutated even when declared with `const`:

```javascript
friends[1] = 'Jay'; // Replace 'Steven' with 'Jay'
console.log(friends); // ['Michael', 'Jay', 'Peter']

// However, this would not work:
// friends = ['Bob', 'Alice']; // Error! Cannot reassign const
variable
```

Arrays can hold values of different types:

```javascript
const jonas = ['Jonas', 'Schmedtmann', 2037 - 1991, 'teacher',
friends];
console.log(jonas);
```

Example of using arrays with functions:

```javascript
const calcAge = function (birthYear) {
  return 2037 - birthYear;
};

const years = [1990, 1967, 2002, 2010, 2018];

const ages = [
  calcAge(years[0]),
  calcAge(years[1]),
  calcAge(years[years.length - 1]),
];
console.log(ages); // [47, 70, 19]
```

Lecture 10: Basic Array Operations (Methods)

JavaScript provides built-in methods to manipulate arrays:

Adding elements:

```
const friends = ['Michael', 'Steven', 'Peter'];

// Add to end
friends.push('Jay');
console.log(friends); // ['Michael', 'Steven', 'Peter', 'Jay']

// Add to beginning
friends.unshift('John');
console.log(friends); // ['John', 'Michael', 'Steven', 'Peter', 'Jay']
```

Push and unshift both return the new array length:

```
const newLength = friends.push('Andrew');
console.log(newLength); // 6
```

Removing elements:

```
// Remove from end
const popped = friends.pop();
console.log(friends); // ['John', 'Michael', 'Steven', 'Peter', 'Jay']
console.log(popped); // 'Andrew'

// Remove from beginning
const shifted = friends.shift();
console.log(friends); // ['Michael', 'Steven', 'Peter', 'Jay']
console.log(shifted); // 'John'
```

Finding elements:

```
console.log(friends.indexOf('Steven')); // 1
console.log(friends.indexOf('Bob')); // -1 (not found)

// ES6 method - returns boolean
console.log(friends.includes('Steven')); // true
console.log(friends.includes('Bob')); // false
```

The `includes` method is useful for conditionals:

```
if (friends.includes('Peter')) {
  console.log('You have a friend called Peter!');
}
```

Lecture 12: Introduction to Objects

While arrays are great for ordered data, objects let us define key-value pairs:

```
const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  age: 2037 - 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven'],
};
```

This is called an object literal syntax - we're literally writing down the object content.

Objects allow us to:

- Give each piece of data a name (key or property name)
- Store unstructured and related data
- Access data by its name rather than position

Unlike arrays, the order of properties in objects doesn't matter when retrieving data.

Lecture 13: Dot vs. Bracket Notation

There are two ways to access object properties:

Dot Notation:

```
const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  age: 2037 - 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven'],
};

console.log(jonas.lastName); // 'Schmedtmann'
```

Bracket Notation:

```
console.log(jonas['lastName']); // 'Schmedtmann'
```

The main difference is that bracket notation allows us to use expressions:

```
const nameKey = 'Name';
console.log(jonas['first' + nameKey]); // 'Jonas'
console.log(jonas['last' + nameKey]); // 'Schmedtmann'

// This would NOT work with dot notation:
// console.log(jonas.'last' + nameKey); // Error
```

Bracket notation is useful when we don't know which property to access until runtime:

```
const interestedIn = prompt('What do you want to know about Jonas?');

if (jonas[interestedIn]) {
  console.log(jonas[interestedIn]);
} else {
  console.log(
    'Wrong request! Choose between firstName, lastName, age, job and
friends.',
  );
}
```

We can also add new properties to objects using either notation:

```
jonas.location = 'Portugal';
jonas['twitter'] = '@jonasschmedtman';
console.log(jonas);
```

Lecture 14: Object Methods

Since functions are just values in JavaScript, we can add them as properties to objects:

```
const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  birthYear: 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven'],
  hasDriversLicense: true,

  // Method (function as property)
  calcAge: function () {
    // 'this' refers to the current object (jonas)
    this.age = 2037 - this.birthYear;
    return this.age;
  },

  getSummary: function () {
    return `${this.firstName} ${
      this.lastName
    } is a ${this.calcAge()}-year old ${this.job}. He has ${
      this.friends.length
    } friends and ${this.hasDriversLicense ? 'a' : 'no'} driver's
license.`;
  },
};
```

Functions attached to objects are called **methods**. We can call them like this:

```
console.log(jonas.calcAge()); // 46
console.log(jonas.age); // 46

// Using the getSummary method
console.log(jonas.getSummary());
// "Jonas Schmedtmann is a 46-year old teacher. He has 3 friends and a
driver's license."
```

In object methods, the `this` keyword refers to the object calling the method. This allows methods to access and manipulate the object's other properties.

The `calcAge` method not only returns the age but also creates a new `age` property on the object, saving us from recalculating it every time.

Lecture 16: Iteration: The for Loop

Loops allow us to automate repetitive tasks. The `for` loop is commonly used when you know exactly how many iterations you need:

```
for (let rep = 1; rep <= 10; rep++) {
  console.log(`Lifting weights repetition ${rep} 🏋️`);
}
```

A for loop has three parts:

1. **Initialization**: `let rep = 1` - sets the initial counter value
2. **Condition**: `rep <= 10` - the loop continues as long as this is true
3. **Update**: `rep++` - updates the counter after each iteration

The loop executes the code block for each iteration as long as the condition remains true.

Lecture 17: Looping Arrays, Breaking and Continuing

One of the most common uses for loops is iterating through arrays:

```
const jonasArray = [
  'Jonas',
  'Schmedtmann',
  2037 - 1991,
  'teacher',
  ['Michael', 'Peter', 'Steven'],
];
const types = [];

for (let i = 0; i < jonasArray.length; i++) {
  console.log(jonasArray[i], typeof jonasArray[i]);

  // Filling a new array
  types.push(typeof jonasArray[i]);
}

console.log(types);
```

We can also use loops to transform one array into another:

```
const years = [1991, 2007, 1969, 2020];
const ages = [];

for (let i = 0; i < years.length; i++) {
  ages.push(2037 - years[i]);
}
console.log(ages);
```

continue and break statements:

The `continue` statement skips the current iteration and continues with the next one:

```
console.log('----- ONLY STRINGS -----');
for (let i = 0; i < jonasArray.length; i++) {
  if (typeof jonasArray[i] !== 'string') continue;
  console.log(jonasArray[i]);
}
```

The `break` statement completely terminates the loop:

```
console.log('----- BREAK WITH NUMBER -----');
for (let i = 0; i < jonasArray.length; i++) {
  if (typeof jonasArray[i] === 'number') break;
  console.log(jonasArray[i]);
}
```

Lecture 18: Looping Backwards and Loops in Loops

Looping backwards through an array:

```
const jonasArray = [
  'Jonas',
  'Schmedtmann',
  2037 - 1991,
  'teacher',
  ['Michael', 'Peter', 'Steven'],
];

for (let i = jonasArray.length - 1; i >= 0; i--) {
  console.log(jonasArray[i]);
}
```

For this loop:

1. Start at the last element: `i = array.length - 1`
2. Continue while `i >= 0`
3. Decrease the counter: `i--`

Nested loops (a loop inside another loop):

```
for (let exercise = 1; exercise <= 3; exercise++) {
  console.log(`----- Starting exercise ${exercise} ------`);

  for (let rep = 1; rep <= 5; rep++) {
    console.log(`Exercise ${exercise}: Lifting weights repetition ${rep} 🏋️‍♀️`);
  }
}
```

In this example, the inner loop (with `rep`) executes completely for each iteration of the outer loop (with `exercise`), resulting in a total of 15 repetitions (3 exercises × 5 repetitions).

Lecture 19: The while Loop

The `while` loop is more versatile than the `for` loop because it only requires a condition:

```
let rep = 1;
while (rep <= 10) {
  console.log(`Lifting weights repetition ${rep} 🏋️`);
  rep++;
}
```

The `while` loop is particularly useful when you don't know in advance how many iterations you need:

```
let dice = Math.trunc(Math.random() * 6) + 1;

while (dice !== 6) {
  console.log(`You rolled a ${dice}`);
  dice = Math.trunc(Math.random() * 6) + 1;
}
```

In this example, we keep rolling a die until we get a 6. Since we can't predict when we'll roll a 6, a `while` loop is more appropriate than a `for` loop.

Key differences between `for` and `while` loops:

- Use `for` when you know the exact number of iterations
- Use `while` when you don't know how many iterations will be needed
- The `while` loop is more flexible but requires manual setup of the counter (if needed)