



# HOW TO Code Like A **Pro**

By Kieran.js

# Intro - A bit about me

- 🔥 DIT Year 2 Student (Software track)
- 🔥 Loves programming
- 🔥 Javascript is love, Javascript is life

# Why this *how to* session?

For FOP assignment, my project code was very long, ugly, inefficient, unorganized and prone to errors.

Whole program squeezed into 1 file (> 1000 lines)

## Benefits of session:

- 🔥 Shorten code (significantly!)
- 🔥 Increase efficiency
- 🔥 Increase speed & productivity
- 🔥 Makes programming more fun
- 🔥 Makes code more *aesthetic*

# Table of Content

- 1) Template literals ☆ (No more string concatenation)
- 2) Guard clauses
- 3) Truthy & Falsy
- 4) Ternary Operator ☆ (Short if-else statement)
- 5) Spread Syntax
- 6) Array Destructuring
- 7) Object Destructuring
- 8) Arrow Functions
- 9) Array Methods ☆ (Sweet with arrow functions!)

# Power & Beauty of... Syntactic Sugar

```
if (description === "sweet") {  
    var str = "sugar"  
}
```

```
else {  
    var str = "salt"  
}
```

*// 6 lines of code into 1*

```
var str = description === "sweet" ? "sugar" : "salt";
```

# Template Literals (``)

- Easy & Modern way of string concatenation
- Denoted and wrapped with backtick (`)



# Template Literals (Example)

Use `${}` to put variables inside the template string

```
const str = "World";  
const stringConcat = "Hello " + str;  
const templateLiteral = `Hello ${str}`;
```

Output: Hello World

// Use template literals instead of string concatenation

Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

# Guard Clauses

- Used in functions
  - Makes functional code much cleaner
  - By removing the **else** keyword when programming
  - Makes use of the **return** statement
- 
- Examples from internet:  
<https://refactoring.com/catalog/replaceNestedConditionalWithGuardClauses.html>



# Guard Clause (Example)

Without guard clause

```
function getEmotion(emoji) {  
  if (emoji === "😊") {  
    return "Happy";  
  }  
  else {  
    return "Not happy";  
  }  
}
```

With guard clause

```
function getEmotion(emoji) {  
  if (emoji === "😊") {  
    return "Happy";  
  }  
  return "Not happy";  
}
```

Returning exits function; else is not needed here

# Truthy & Falsy Values

- Good to know
- When encountered in a Boolean Context
  - **Falsy** values will be coerced (coverted) to **false**
  - **Truthy** values will be coerced to **true**
- Boolean context is basically where conditions are written
  - if (condition)
  - while (condition)
- Read more at <https://developer.mozilla.org/en-US/docs/Glossary/Truthy>

# Truthy & Falsy Values

## ■ Falsy values:

- false
- 0
- empty string ("")
- null
- undefined
- NaN

## ■ Truthy values:

- true
- All values that are not falsy

# Truthy & Falsy Values (Example)

- ```
If (0) {  
    console.log("zero");  
}
```
- ```
else {  
    console.log("one");  
}
```
- "one" gets printed out because 0 is a falsy value

# Truthy & Falsy Values (Example)

- ```
if ([]) {  
    console.log("teuroo");  
}
```
- ```
else {  
    console.log("phalse");  
}
```
- "teuroo" gets printed out because an array is a truthy value

# Ternary Operator (?)

- If-else statement in 1 line
- `age >= 21 ? "adult" : "child"`



I totally did not steal this image from youtube XD

# Ternary Operator (?)

- Can be chained (if, else if, else if...)
- Evaluates to an expression
- Can be used in
  - Function arguments
  - Arrays/Object literals
  - Anywhere that a value/expression is expected
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional\\_Operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)

# Spread Syntax (...)

- Denoted by 3 dots (...)
  - Expands an iterable (array/string)
  - Basically splits up an iterable and gives all individual elements
  - Simply think of it as spreading over the value's characters/elements
- 
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)



# Spread Syntax (Example)

```
const numbers = [1, 2, 3, 4];  
const fruit = "apple";  
console.log(...numbers) //output: 1, 2, 3, 4  
console.log(...fruit) // output: a p p l e
```

- Make a new copy of an array:
  - `const numbersCopy = [...numbers]` // much shorter than using splice or for loop
- Adding multiple items:
  - `const oneToNine = [0, ...numbers, 5, 6, 7, 8, 9]`

# Array Destructuring

- Unpack values from an array
  - Store them into variables
  - Removes the need for array[index] where suitable
  - Removes unnecessary and verbose lines
  - One liner for retrieving array elements
- 
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

# Array Destructuring (Example)

```
const array = ["A", 1];  
// Using index:  
const letter = array[0]; // A  
const number = array[1]; // 1  
  
// Destructuring:  
const [letter, number] = array;  
console.log(letter); // A  
console.log(number) // 1
```

# Object Destructuring

- Unpack values from an object
  - Store them into variables
  - Removes the need for repetitive typing of `object.property/object[property]`
  - One liner for retrieving object property values
- 
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment#object\\_destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#object_destructuring)

# Object Destructuring (Example)

```
const student = { id: "p1234567", age: 17 };  
// We usually use:  
console.log(`${student.id} is ${student.age} years old`);  
  
// With Destructuring:  
// Removes the need for repeating the word "person"  
const { id, age } = student;  
console.log(`${id} is ${age} years old`);
```

# Function Expressions

- Functions are one of the most used features of javascript
  - Different way of function declaration
  - Function expressions are functions where the name can be omitted to create anonymous functions
- 
- Read more at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function>

# Function Expressions (Example)

- Named

```
const print = function printHelloWorld() {  
  console.log("Hello World");  
}  
print() // Hello World
```

- Not named

```
const print = function() {  
  console.log("Hello World");  
}  
print() // Hello World
```

# Arrow Functions

- Same as normal functions with slight differences
  - Create new way of function declaration (uses `=>` instead of **function**)
  - **Can** be reduced into **1** line
  - Does not bind the **this** keyword
- 
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)



# Arrow Functions

- Assign arrow function to variable

```
const arrowFunction = () => {  
  return "This is an arrow function named as arrowFunction";  
}
```

- In one line:

```
const arrowFunction = () => "This is an arrow function named as arrowFunction";
```

# Array Methods

- In-built array methods that can help you reduce lines of code
- Omit the need to write for loops
- Methods covered:
  - Map
  - Filter
  - Reduce

# Array Methods - Map

- Takes in a callback function
  - Callback's First parameter = item of array in each iteration
  - Returns a new array
  - **Contains items mapped to return value of callback function**
- 
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

# Array Methods - Map (Example)

```
const scores = [17, 35, 50, 52, 60, 75, 90];
```

- Writing a named function:

```
function addOneToScore(score) {  
  return score + 1;  
}  
const newScores = scores.map(addOneToScore)
```

- Providing arrow callback directly:

```
const newScores = scores.map(score => score + 1);
```

# Array Methods - filter

- Takes in a callback function
  - Callback's First parameter = item of array in each iteration
  - Returns a new array
  - **Contains items that pass the condition in callback function**
  - Items that fail the condition will not be in the returned array
- 
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/filter](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)

# Array Methods - filter (Example)

```
const scores = [17, 35, 50, 52, 60, 75, 90];
```

- Writing a named function:

```
function getPasses(score) {  
  return score >= 50  
}  
const passingScores = scores.filter(getPasses)
```

- Providing arrow callback directly:

```
const passingScores = scores.filter(score => score >= 50)
```

# Array Methods - reduce

- Takes in a callback function
  - Callback's First parameter = Accumulated value
  - Callback's Second parameter = current item of array in each iteration
  - .reduce() takes in an optional 2<sup>nd</sup> argument
    - 2<sup>nd</sup> argument provided: accumulated value starts with argument provided
    - 2<sup>nd</sup> argument not provided: accumulated value starts with first item in array
  - **Returns the Accumulated value**
- 
- Read more at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/reduce](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce)

# Array Methods - reduce (Example)

```
const scores = [17, 35, 50, 52, 60, 75, 90];
```

- Writing a named function:

```
function getSum(acc, cur) {  
  return acc + cur;  
}  
const sumOfScores = scores.reduce(getSum)
```

- Providing arrow callback directly:

```
const sumOfScores = scores.reduce((acc, cur) => acc + cur)
```



# THANK YOU!

- Some of these concepts will definitely come in handy
- Highly recommended to look at the source codes provided in github repository! (<https://github.com/kspc100/how-to-code-like-a-pro>)
- Links to resources where you can read more are also provided in the slides