ES6 Reference Guide

Background

"ECMAScript" (also called "ES") is the name of the international standard defining JavaScript. Released in 2015, ES6 contains numerous changes to the language syntax and is the recommended version of this standard.

Assigning Variables

`let`

`let` contains values that are re-assignable, but they are not accessible before they are declared. Note the error below:

```
function letLogger() {
  console.log(x);
  let x = "hello";
}

letLogger();
```

```
> Uncaught ReferenceError: x is not defined
  at letLogger (<anonymous>:2:17)
  at <anonymous>:1:1
```

To address the error, simply declare the variable prior to the `console.log` statement.

`const`

`const` stands for "constant" and values may not be reassigned. Similar to `let`, `const` does not declare or initialize values until that line of code is run.

Though the values cannot be reassigned, objects and arrays can be manipulated using methods such as `.pop()` and `.push()`:

```
const petNames = ["Cleo", "Jax", "Chance", "Buckaroo", "fishy"]
console.log("All of my pets: ", petNames);

// Remove the last value from the array
petNames.pop();

// View the array
console.log("Dogs and cats: ", petNames)
```

The array after using `.pop()`:

```
Dogs and cats: >(4) ["Cleo", "Jax", "Chance", "Buckaroo"]
```

`.forEach`

`.forEach` is used to call a function on each item in an array.

```
function printWithIndex(d, i) {
  console.log(i, d)
}

var arr = ["One", "Two", "Three", "Four"];

arr.forEach(printWithIndex);
```

```
0 "One"
1 "Two"
2 "Three"
3 "Four"
```

In the above example, `.forEach` is chained with the variable `arr`, returning both arguments (data and index) of the function.

Template Literals

`Template Literals` replace traditional string concatenation. Instead of quotes, backticks are utilized. Place holders are contained with `$` and the expression is wrapped in curly brackets:

```
let firstName = "John";
let lastName = "Doe";

const fullName = `${firstName} ${lastName}`;

console.log(fullName);

> John Doe
```

`.map`

This method creates a new array from an existing array while leaving the original unchanged.

Let's work through an example.

- First, create an example that will multiply a number by two:

```
function timesTwo(num) {
    return 2 * num;
}
```

- Next, call the `timesTwo` function on each element in the array using `.map`:

```
var doubleItems = [1, 2, 3, 4].map(timesTwo);
console.log(doubleItems);
```

```
> (4) [2, 4, 6, 8]
```

Here is another example using `.map`:

```
let students = [{name: "John", grade: 89}, {name: "Jane", grade: 91}];
function getGrades(student) {
    return student.grade;
}
let grades = students.map(getGrades);
console.log(grades)
```

```
> (2) [89, 91]
```

When used in conjunction with the `getGrades` function above, `.map` creates a new variable containing only the student grades. The original array remains untouched.

Arrow Functions

Arrow functions provide a new syntax for writing functions in JavaScript. Using arrow functions creates code that is more concise and streamlined.

Let's revisit the code from the `.map` example above:

```
// Original function
function getGrades(student) {
    return student.grade;
}
let grades = students.map(getGrades);
```

```
// The same function rewritten as an arrow function
students.map(student => student.grade)
```

The same block of code has been condensed into a single line with the use of an arrow function. Note that a "fat arrow" has replaced the word "function". Also, without the curly brackets, the return statement is implied.

To create an arrow function using a single parameter, parentheses and curly brackets are omitted completely:

```
var square = x => x * x;
```

Parentheses contain two parameters in an arrow function, but curly brackets are still omitted:

```
var multiply = (a, b) => a * b;
```
