Remove Items Using splice()

Ok, so we've learned how to remove elements from the beginning and end of arrays using `shift()` and `pop()`, but what if we want to remove an element from somewhere in the middle? Or remove more than one element at once? Well, that's where `splice()` comes in. `splice()` allows us to do just that: **remove any number of consecutive elements** from anywhere in an array.

`splice()` can take up to 3 parameters, but for now, we'll focus on just the first 2. The first two parameters of `splice()` are integers which represent indexes, or positions, of the array that `splice()` is being called upon. And remember, arrays are *zero-indexed*, so to indicate the first element of an array, we would use `0`. `splice()`'s first parameter represents the index on the array from which to begin removing elements, while the second parameter indicates the number of elements to delete. For example:

```
let array = ['today', 'was', 'not', 'so', 'great'];


array.splice(2, 2);
// remove 2 elements beginning with the 3rd element
// array now equals ['today', 'was', 'great']
```

`splice()` not only modifies the array it's being called on, but it also returns a new array containing the value of the removed elements:

```
let array = ['I', 'am', 'feeling', 'really', 'happy'];


let newArray = array.splice(3, 2);
// newArray equals ['really', 'happy']
```

Add Items Using splice()

Remember in the last challenge we mentioned that `splice()` can take up to three parameters? Well, you can use the third parameter, comprised of one or more element(s), to add to the array. This can be incredibly useful for quickly switching out an element, or a set of elements, for another.

```
const numbers = [10, 11, 12, 12, 15];
const startIndex = 3;
const amountToDelete = 1;


numbers.splice(startIndex, amountToDelete, 13, 14);
```

```
// the second entry of 12 is removed, and we add 13 and 14 at the same index

console.log(numbers);

// returns [ 10, 11, 12, 13, 14, 15 ]
```

Here, we begin with an array of numbers. Then, we pass the following to `splice()`: The index at which to begin deleting elements (3), the number of elements to be deleted (1), and the remaining arguments (13, 14) will be inserted starting at that same index. Note that there can be any number of elements (separated by commas) following `amountToDelete`, each of which gets inserted.

Copy Array Items Using slice()

The next method we will cover is `slice()`. Rather than modifying an array, `slice()` copies or *extracts* a given number of elements to a new array, leaving the array it is called upon untouched. `slice()` takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (extraction will occur up to, but not including the element at this index). Consider this:

```
let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear'];


let todaysWeather = weatherConditions.slice(1, 3);

// todaysWeather equals ['snow', 'sleet'];

// weatherConditions still equals ['rain', 'snow', 'sleet', 'hail', 'clear']
```

In effect, we have created a new array by extracting elements from an existing array.

Use the Conditional (Ternary) Operator

The *conditional operator*, also called the *ternary operator*, can be used as a one line if-else expression.

The syntax is:

```
condition ? expression-if-true : expression-if-false;
```

The following function uses an if-else statement to check a condition:

```
function findGreater(a, b) {
  if(a > b) {
    return "a is greater";
  }
  else {
    return "b is greater";
  }
}
```

This can be re-written using the `conditional operator`:

```
function findGreater(a, b) {
  return a > b ? "a is greater" : "b is greater";
}
```

Use Multiple Conditional (Ternary) Operators

In the previous challenge, you used a single conditional operator. You can also chain them together to check for multiple conditions.

The following function uses if, else if, and else statements to check multiple conditions:

```
function findGreaterOrEqual(a, b) {
  if (a === b) {
    return "a and b are equal";
  }
  else if (a > b) {
    return "a is greater";
  }
  else {
    return "b is greater";
  }
}
```

The above function can be re-written using multiple conditional operators:

```
function findGreaterOrEqual(a, b) {
  return (a === b) ? "a and b are equal"
```

```
    : (a > b) ? "a is greater"
    : "b is greater";
}
```

It is considered best practice to format multiple conditional operators such that each condition is on a separate line, as shown above. Using multiple conditional operators without proper indentation may make your code hard to read. For example:

```
function findGreaterOrEqual(a, b) {
  return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";
```

Use Arrow Functions to Write Concise Anonymous Functions

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we do not reuse them anywhere else.

To achieve this, we often use the following syntax:

```
const myFunc = function() {
  const myVar = "value";
  return myVar;
}
```

ES6 provides us with the syntactic sugar to not have to write anonymous functions this way. Instead, you can use **arrow function syntax**:

```
const myFunc = () => {
  const myVar = "value";
  return myVar;
}
```

When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword `return` as well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:

```
const myFunc = () => "value";
```

This code will still return the string `value` by default.

Write Arrow Functions with Parameters

Just like a regular function, you can pass arguments into an arrow function.

```
// doubles input value and returns it
const doubler = (item) => item * 2;
doubler(4); // returns 8
```

If an arrow function has a single parameter, the parentheses enclosing the parameter may be omitted.

```
// the same function, without the parameter parentheses
const doubler = item => item * 2;
```

It is possible to pass more than one argument into an arrow function.

```
// multiplies the first input value by the second and returns it
const multiplier = (item, multi) => item * multi;
multiplier(4, 2); // returns 8
```

Set Default Parameters for Your Functions

In order to help us create more flexible functions, ES6 introduces *default parameters* for functions.

Check out this code:

```
const greeting = (name = "Anonymous") => "Hello " + name;


console.log(greeting("John")); // Hello John
console.log(greeting()); // Hello Anonymous
```

The default parameter kicks in when the argument is not specified (it is undefined). As you can see in the example above, the parameter `name` will receive its default value `"Anonymous"` when you do not provide a value for the parameter. You can add default values for as many parameters as you want.

Use the Rest Parameter with Function Parameters

In order to help us create more flexible functions, ES6 introduces the *rest parameter* for function parameters. With the rest parameter, you can create functions that take a variable number of arguments. These arguments are stored in an array that can be accessed later from inside the function.

Check out this code:

```
function howMany(...args) {
  return "You have passed " + args.length + " arguments.";
}
console.log(howMany(0, 1, 2)); // You have passed 3 arguments.
console.log(howMany("string", null, [1, 2, 3], { })); // You have passed 4 arguments.
```

The rest parameter eliminates the need to check the `args` array and allows us to apply `map()`, `filter()` and `reduce()` on the parameters array.