Debugging and Functions

Introduction

Goals for tonight

- Learn debugging strategies
- Review/extend knowledge of JS:
  - Functions
  - Callbacks

Debugging

- You are going to make mistakes!
- Let's examine ways to better debug
- Let's first examine some common errors

SyntaxError

- You've seen this one before!
- You have to fix these right away!

```
"awesome;

function first( {}

let = "nice!";
```

ReferenceError

- Thrown when you try to access a variable that is **not defined**
- This does not mean *undefined*

```
function sayHi() {
  let greeting = "hi!";
}

sayHi();

greeting;   // ReferenceError
```

TypeError

- Trying to do something with a type that you can not
- Accessing properties on *undefined* or *null*
- Invoking *("calling")* something that is not a function

```
"awesome".splice(); // TypeError

let obj = {};

obj.firstName.moreInfo; // TypeError
```

Two Kinds of Bugs

- An error is thrown — **easier**
- You didn't get what you wanted — **harder!**

A process for debugging

- Make assumptions
- Test assumptions
- Prove assumptions
- Repeat

console.log

- Be mindful about what you print out
- Great for a sanity check
- Even better when you add parameters

```
console.log("We made it!");

console.log("x =", x);
```

Challenge: indexVowelTotal

indexVowelTotal

Write a function, *indexVowelTotal*, which accepts an array of words and returns the sum of the indexes of each word when each character is a vowel.

```
indexVowelTotal([]); // 0

indexVowelTotal(['hello']); // 5

indexVowelTotal(['Joel', 'Elie', 'Kate', 'Brit']); // 14
```

```
function indexVowelTotal(words) {
  let sum = 0;

  for (let word of words) {
    sum + countVowelIndexes(word);
  }

  return sum;
}

function countVowelIndexes(string) {
  let sum = 0;

  for (let i = 0; i < string.length; i++) {
    let isCharVowel = isVowel(i);
    if (isCharVowel) {
      sum += string[i];
    }

    return sum;
  }
}

function isVowel(char) {
  return "aeiou".includes(char);
}
```

Solution

```
function indexVowelTotal(words) {
  let sum = 0;

  for (let word of words) {
    sum += countVowelIndexes(word);
  }

  return sum;
}

function countVowelIndexes(string) {
  let sum = 0;

  for (let i = 0; i < string.length; i++) {
    let isCharVowel = isVowel(string[i]);
    if (isCharVowel) {
      sum += i;
    }
  }
  return sum;
}

function isVowel(char) {
  return "aeiouAEIOU".includes(char);
}
```

Functions

- Allow us to execute an operation multiple times
- We can choose:
  - What input the function accepts
  - What the function outputs (using *return* keyword)
- Once we *return* we're done!
- Define using *function* keyword and execute by placing `()` right after

An example

```
function add(a,b) {
  return a + b;
}

add(10, 20); // 30
```

Arguments and parameters

In JS, these terms are often used interchangeably.

We're going to be more explicit:

Parameters
: Variables defined in the *definition* of a function

```
function add(a, b) { /* ... */ }
```

Arguments
: Values passed to the function *when it is invoked*

```
add(10, 20);
```

Too Many / Too Few Arguments

```
function add(x, y) {
  return x + y;
}
```

If you pass too many arguments, JS ignores the extra arguments:

```
add(2, 3, 4);  // 4 is ignored!
```

If you pass too few arguments, JS uses *undefined* for missing values:

```
add(2);  // `y` will be `undefined`
```

Callbacks

Functions can accept other functions

- Not all languages allow for this! JavaScript is special.
- A function *passed as an argument* to a function is called a ***callback***.

Why bother with callbacks?

- This helps us reduce duplication!
- You can pass in a defined function or make up one on the spot
- They can provide infinite levels of flexibility
- Let's see what we mean!

Write a function that accepts an array, and returns a new array with only even numbers

```
function filterOddNumbers(nums){
  let evenNums = [];
  for (let num of nums) {
    if (num % 2 === 0) {
      evenNums.push(num);
    }
  }
  return evenNums;
}
```

This works just fine, but what happens if we want to filter all odd numbers?

- What about other conditions?
- What about ranges?
- Here's where callbacks can help!

Refactoring to use a callback

```
function filter(array, callback) {
  let filteredArray = [];
  for (let val of array) {
    let result = callback(val)
    if (result === true) {
      filteredArray.push(val);
    }
  }
  return filteredArray;
}
```

```
filter([1,2,3,4,5,6], function(num) {
  return num % 2 === 0;
}); // [2,4,6]

filter([1,2,3,4,5,6], function(num) {
  return num > 2;
}); // [3,4,5,6]
```

This function already exists!

Useful functions with callbacks

Let's meet three functions we can call on arrays that use callbacks:

- *forEach*
- *map*
- *filter*

forEach

Iterate over items in array, running callback function for each:

```
let users = [
  { name: "Maya", hobby: "Swimming" },
  { name: "Malik", hobby: "Biking" },
  { name: "Anil", hobby: "Swimming" },
];

users.forEach(function (user) {
  console.log(user.hobby);
});
```

map

Create a new array by "transforming" each item in original array with callback function:

```
let users = [
  { name: "Maya", hobby: "Swimming" },
  { name: "Malik", hobby: "Biking" },
  { name: "Anil", hobby: "Swimming" },
];

users.map(function (user) {
  return user.hobby;
});
```

filter

Create a new array of only those items that the callback returns *true* for:

```
let users = [
  { name: "Maya", hobby: "Swimming" },
  { name: "Malik", hobby: "Biking" },
  { name: "Anil", hobby: "Swimming" },
];

users.filter(function (user) {
  return user.hobby === "Swimming";
});
```

- Always make sure to *return* in *map* and *filter*
- You can combine these functions together!
- You can practice more here! <https://www.rithmschool.com/courses/intermediate-javascript/javascript-iterators-foreach-map-filter>

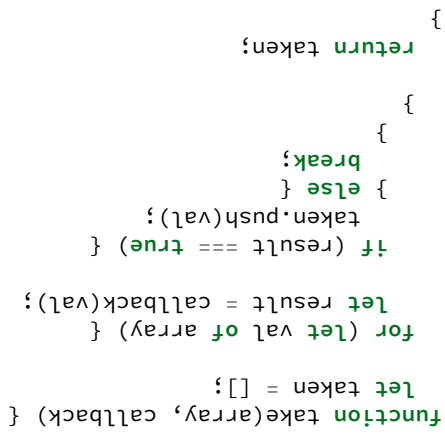
Challenge: take

Write a function called *take* which accepts an array of values and adds each value in the array to a new array **until** the callback returns *false*.

```
take([1, 3, 5, 8, 9], function (num) {
  return num % 2 !== 0;
});

// [1, 3, 5]
```

Solution



Challenge: arrayToObject

arrayToObject accepts an array of objects and a callback function.

It should return an object, where:

- each key is the result of the callback for each object in the array
- each value is the entire object in the array

The order of the keys/values in the object does not matter.

```
let data = [{ name: "Joel", favNum: 22 }, { name: "Elie", favNum: 44 }];

arrayToObject(data, function (value) {
  return value.name;
});

// { Elie: {name: "Elie", favNum: 44},
//   Joel: {name: "Joel", favNum: 22} }

arrayToObject(data, function (value) {
  return value.favNum;
});

// { 22: {name: "Joel", favNum: 22},
//   44: {name: "Elie", favNum: 44} }
```

Solution

```
function arrayToObject(array, generateKeyFn) {
  const container = {};

  for (let item of array) {
    const containerKey = generateKeyFn(item);
    container[containerKey] = item;
  }

  return container;
}
```

Arrow Functions

- Arrow functions are shorthand for anonymous functions
- They cannot be named and they only work as function expressions.

```
function sayHi() {
  return "Hi!";
}
```

is the same as

```
let sayHi = () => {
  return "Hi!";
};
```

Arrow Functions have an implicit return if you leave out the curly braces

```
function add (num1, num2) {
  return num1 + num2;
}
```

is the same as

```
let add = (num1, num2) => num1 + num2;
```

Where they are most useful

With callback functions like *map* and *filter*!

```
[1,2,3,4].map(function(num) {
  return num * 2;
});
```

is the same as

```
[1,2,3,4].map(num => num * 2);
```

Things you do not need to know *yet*

Closures

- A function that provides access to variables in its outer scope.

Recursion

- A recursive function is a function that calls itself.
- Useful for problems where the input is, itself, recursive