Testing Objects for Properties

It is useful to check if the property of a given object exists or not. We can use the `.hasOwnProperty(propname)` method of objects to determine if that object has the given property name. `.hasOwnProperty()` returns `true` or `false` if the property is found or not.

Record Collection

You are given an object literal representing a part of your musical album collection. Each album has a unique id number as its key and several other properties. Not all albums have complete information.

You start with an `updateRecords` function that takes an object literal, `records`, containing the musical album collection, an `id`, a `prop` (like `artist` or `tracks`), and a `value`. Complete the function using the rules below to modify the object passed to the function.

- Your function must always return the entire record collection object.
- If `prop` isn't `tracks` and `value` isn't an empty string, update or set that album's `prop` to `value`.
- If `prop` is `tracks` but the album doesn't have a `tracks` property, create an empty array and add `value` to it.
- If `prop` is `tracks` and `value` isn't an empty string, add `value` to the end of the album's existing `tracks` array.
- If `value` is an empty string, delete the given `prop` property from the album.

Note: A copy of the `recordCollection` object is used for the tests.

After `updateRecords(recordCollection, 5439, "artist", "ABBA")`, `artist` should be the string `ABBA`

After `updateRecords(recordCollection, 5439, "tracks", "Take a Chance on Me")`, `tracks` should have the string `Take a Chance on Me` as the last element.

After `updateRecords(recordCollection, 2548, "artist", "")`, `artist` should not be set

After `updateRecords(recordCollection, 1245, "tracks", "Addicted to Love")`, `tracks` should have the string `Addicted to Love` as the last element.

After `updateRecords(recordCollection, 2468, "tracks", "Free")`, `tracks` should have the string `1999` as the first element.

After `updateRecords(recordCollection, 2548, "tracks", "")`, `tracks` should not be setAfter `updateRecords(recordCollection, 1245, "albumTitle", "Riptide")`, `albumTitle` should be the string `Riptide`

```
// Setup
const recordCollection = {
  2548: {
    albumTitle: 'Slippery When Wet',
    artist: 'Bon Jovi',
    tracks: ['Let It Rock', 'You Give Love a Bad Name']
  },
  2468: {
    albumTitle: '1999',
    artist: 'Prince',
    tracks: ['1999', 'Little Red Corvette']
  },
  1245: {
    artist: 'Robert Palmer',
    tracks: []
  },
  5439: {
    albumTitle: 'ABBA Gold'
  }
};

// Only change code below this line
function updateRecords(records, id, prop, value) {
  if(prop !=='tracks' && value!==''){
    records[id][prop] = value;
  } else if(prop === 'tracks' && records[id].hasOwnProperty('tracks') === false){
    records[id][prop] = [value]
  } else if( prop === 'tracks' && value !== '') {
    records[id][prop].push(value);
  } else if( value === '') {
    delete records[id][prop];
  }
  return records;
}

updateRecords(recordCollection, 5439, 'artist', 'ABBA');
```

REPLACE LOOPS USING RECURSION

Recursion is the concept that a function can be expressed in terms of itself. To help understand this, start by thinking about the following task: multiply the first `n` elements of an array to create the product of those elements. Using a `for` loop, you could do this:

```
function multiply(arr, n) {
  let product = 1;
  for (let i = 0; i < n; i++) {
    product *= arr[i];
  }
  return product;
}
```

However, notice that `multiply(arr, n) == multiply(arr, n - 1) * arr[n - 1]`. That means you can rewrite `multiply` in terms of itself and never need to use a loop.

```
function multiply(arr, n) {
  if (n <= 0) {
    return 1;
  } else {
    return multiply(arr, n - 1) * arr[n - 1];
  }
}
```

The recursive version of `multiply` breaks down like this. In the base case, where `n <= 0`, it returns 1. For larger values of `n`, it calls itself, but with `n - 1`. That function call is evaluated in the same way, calling `multiply` again until `n <= 0`. At this point, all the functions can return and the original `multiply` returns the answer.

Note: Recursive functions must have a base case when they return without calling the function again (in this example, when `n <= 0`), otherwise they can never finish executing.

Write a recursive function, `sum(arr, n)`, that returns the sum of the first `n` elements of an array `arr`.

- `sum([1], 0)` should equal 0.
- `sum([2, 3, 4], 1)` should equal 2.
- `sum([2, 3, 4, 5], 3)` should equal 9.

Your code should not rely on any kind of loops (`for` or `while` or higher order functions such as `forEach`, `map`, `filter`, or `reduce`.).You should use recursion to solve this problem.

```
function sum(arr, n) {
  if(n <= 0) {
    return 0;
  } else {
    return sum(arr, n - 1) + arr[n - 1];
  }
}
```

PROFILE LOOKUP

We have an array of objects representing different people in our contacts lists.

A `lookUpProfile` function that takes `name` and a property (`prop`) as arguments has been pre-written for you.

The function should check if `name` is an actual contact's `firstName` and the given property (`prop`) is a property of that contact.

If both are true, then return the "value" of that property.

If `name` does not correspond to any contacts then return the string `No such contact`.

If `prop` does not correspond to any valid properties of a contact found to match `name` then return the string `No such property`.

```
// Setup
const contacts = [
  {
    firstName: "Akira",
    lastName: "Laine",
    number: "0543236543",
    likes: ["Pizza", "Coding", "Brownie Points"],
  },
  {
    firstName: "Harry",
    lastName: "Potter",
    number: "0994372684",
    likes: ["Hogwarts", "Magic", "Hagrid"],
  },
  {
    firstName: "Sherlock",
    lastName: "Holmes",
    number: "0487345643",
    likes: ["Intriguing Cases", "Violin"],
  },
  {
    firstName: "Kristian",
    lastName: "Vos",
    number: "unknown",
    likes: ["JavaScript", "Gaming", "Foxes"],
  },
];

function lookUpProfile(name, prop) {
  // Only change code below this line
for (let x = 0; x < contacts.length; x++) {
    if (contacts[x].firstName === name) {
      if (contacts[x].hasOwnProperty(prop)) {
        return contacts[x][prop];
      } else {
        return "No such property";
      }
    }
  }
  return "No such contact"
  // Only change code above this line
}

lookUpProfile("Akira", "likes");
```

GENERATE RANDOM WHOLE NUMBERS WITHIN A RANGE

Instead of generating a random whole number between zero and a given number like we did before, we can generate a random whole number that falls within a range of two specific numbers.

To do this, we'll define a minimum number `min` and a maximum number `max`.

Here's the formula we'll use. Take a moment to read it and try to understand what this code is doing:

```
Math.floor(Math.random() * (max - min + 1)) + min
```

Create a function called `randomRange` that takes a range `myMin` and `myMax` and returns a random whole number that's greater than or equal to `myMin`, and is less than or equal to `myMax`, inclusive.

```
function randomRange(myMin, myMax) {
  // Only change code below this line
  return Math.floor(Math.random() * (myMax - myMin + 1) + myMin);
  // Only change code above this line
}
```

USE THE PARSEINT FUNCTION WITH A RADIX

The `parseInt()` function parses a string and returns an integer. It takes a second argument for the radix, which specifies the base of the number in the string. The radix can be an integer between 2 and 36.

The function call looks like:

```
parseInt(string, radix);
```

And here's an example:

```
const a = parseInt("11", 2);
```

The radix variable says that `11` is in the binary system, or base 2. This example converts the string `11` to an integer `3`.

Use `parseInt()` in the `convertToInteger` function so it converts a binary number to an integer and returns it.

```
function convertToInteger(str) {
return parseInt(str, 2)
}

convertToInteger("10011");
```

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 `a ? b : c`, where `a` is the condition, `b` is the code to run when the condition returns `true`, and `c` is the code to run when the condition returns `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 or equal";
  }
}
```

This can be re-written using the conditional operator:

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

Use the conditional operator in the `checkEqual` function to check if two numbers are equal or not. The function should return either the string `Equal` or the string `Not Equal`

```
function checkEqual(a, b) {
return a === b ? "Equal" : "Not Equal"
}

checkEqual(1, 2);
```

USE RECURSION TO CREATE A COUNTDOWN

In a previous challenge, you learned how to use recursion to replace a `for` loop. Now, let's look at a more complex function that returns an array of consecutive integers starting with `1` through the number passed to the function.

As mentioned in the previous challenge, there will be a base case. The base case tells the recursive function when it no longer needs to call itself. It is a simple case where the return value is already known. There will also be a recursive call which executes the original function with different arguments. If the function is written correctly, eventually the base case will be reached.

For example, say you want to write a recursive function that returns an array containing the numbers `1` through `n`. This function will need to accept an argument, `n`, representing the final number. Then it will need to call itself with progressively smaller values of `n` until it reaches `1`. You could write the function as follows:

```
function countup(n) {
  if (n < 1) {
    return [];
  } else {
    const countArray = countup(n - 1);
    countArray.push(n);
    return countArray;
  }
}
console.log(countup(5));
```

The value `[1, 2, 3, 4, 5]` will be displayed in the console.

At first, this seems counterintuitive since the value of `n` *decreases*, but the values in the final array are *increasing*. This happens because the push happens last, after the recursive call has returned. At the point where `n` is pushed into the array, `countup(n - 1)` has already been evaluated and returned `[1, 2, ..., n - 1]`.

We have defined a function called `countdown` with one parameter (`n`). The function should use recursion to return an array containing the integers `n` through `1` based on the `n` parameter. If the function is called with a number less than 1, the function should return an empty array. For example, calling this function with `n = 5` should return the array `[5, 4, 3, 2, 1]`. Your function must use recursion by calling itself and must not use loops of any kind.

- `countdown(-1)` should return an empty array.
- countdown(10) `should return` (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
- countdown(5) `should return` (5, 4, 3, 2, 1)
- Your code should not rely on any kind of loops (`for` , `while` or higher order functions such as `forEach` , `map` , `filter` , and `reduce`).
- You should use recursion to solve this problem.

```
function countdown(n){
  if(n < 1){
    return [];
  } else {
    const array = countdown (n-1);
    array.unshift(n);
    return array
  }
}
```

USE RECURSION TO CREATE A RANGE OF NUMBERS

Continuing from the previous challenge, we provide you another opportunity to create a recursive function to solve a problem.

We have defined a function named `rangeOfNumbers` with two parameters. The function should return an array of integers which begins with a number represented by the `startNum` parameter and ends with a number represented by the `endNum` parameter. The starting number will always be less than or equal to the ending number. Your function must use recursion by calling itself and not use loops of any kind. It should also work for cases where both `startNum` and `endNum` are the same.

```javascript
function rangeOfNumbers(startNum, endNum) {
  if (endNum - startNum === 0) {
    return [startNum];
  } else {
    var numbers = rangeOfNumbers(startNum, endNum - 1);
    numbers.push(endNum);
    return numbers;
  }
}
```

PREVENT OBJECT MUTATION

As seen in the previous challenge, `const` declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function `Object.freeze` to prevent data mutation.

Any attempt at changing the object will be rejected, with an error thrown if the script is running in strict mode.

```
let obj = {
  name:"FreeCodeCamp",
  review:"Awesome"
};
Object.freeze(obj);
obj.review = "bad";
obj.newProp = "Test";
console.log(obj);
```

The `obj.review` and `obj.newProp` assignments will result in errors, because our editor runs in strict mode by default, and the console will display the value `{ name: "FreeCodeCamp", review: "Awesome" }`.

In this challenge you are going to use `Object.freeze` to prevent mathematical constants from changing. You need to freeze the `MATH_CONSTANTS` object so that no one is able to alter the value of `PI`, add, or delete properties.

```
function freezeObj() {
  const MATH_CONSTANTS = {
    PI: 3.14
  };
  // Only change code below this line
Object.freeze(MATH_CONSTANTS)

  // Only change code above this line
  try {
    MATH_CONSTANTS.PI = 99;
  } catch(ex) {
    console.log(ex);
  }
  return MATH_CONSTANTS.PI;
}
const PI = freezeObj();
```

PROFILE LOOKUP

We have an array of objects representing different people in our contacts lists.

A `lookUpProfile` function that takes `name` and a property (`prop`) as arguments has been pre-written for you.

The function should check if `name` is an actual contact's `firstName` and the given property (`prop`) is a property of that contact.

If both are true, then return the "value" of that property.

If `name` does not correspond to any contacts then return the string `No such contact`.

If `prop` does not correspond to any valid properties of a contact found to match `name` then return the string `No such property`.

```
// Setup
const contacts = [
  {
    firstName: "Akira",
    lastName: "Laine",
    number: "0543236543",
    likes: ["Pizza", "Coding", "Brownie Points"],
  },
  {
    firstName: "Harry",
    lastName: "Potter",
    number: "0994372684",
    likes: ["Hogwarts", "Magic", "Hagrid"],
  },
  {
    firstName: "Sherlock",
    lastName: "Holmes",
    number: "0487345643",
    likes: ["Intriguing Cases", "Violin"],
  },
  {
    firstName: "Kristian",
    lastName: "Vos",
    number: "unknown",
```

```javascript
    likes: ["JavaScript", "Gaming", "Foxes"],
  },
];

function lookUpProfile(name, prop) {
  // Only change code below this line
    for ( let i=0 ; i< contacts.length; i++){
      if(contacts[i].firstName === name){
          return contacts[i][prop] || "No such property";
      }
    }
    return "No such contact"
  // Only change code above this line
}

var data = lookUpProfile("Kristian", "number");
console.log(data)
```

USE DESTRUCTURING ASSIGNMENT TO ASSIGN VARIABLES FROM NESTED OBJECTS

You can use the same principles from the previous two lessons to destructure values from nested objects.

Using an object similar to previous examples:

```
const user = {
  johnDoe: {
    age: 34,
    email: 'johnDoe@freeCodeCamp.com'
  }
};
```

Here's how to extract the values of object properties and assign them to variables with the same name:

```
const { johnDoe: { age, email }} = user;
```

And here's how you can assign an object properties' values to variables with different names:

```
const { johnDoe: { age: userAge, email: userEmail }} = user;
```

Replace the two assignments with an equivalent destructuring assignment. It should still assign the variables `lowToday` and `highToday` the values of `today.low` and `today.high` from the `LOCAL_FORECAST` object.

```
const LOCAL_FORECAST = {
  yesterday: { low: 61, high: 75 },
  today: { low: 64, high: 77 },
  tomorrow: { low: 68, high: 80 }
};

// Only change code below this line
/*
const lowToday = LOCAL_FORECAST.today.low;
const highToday = LOCAL_FORECAST.today.high; */
const { today : { low: lowToday , high: highToday}}
= LOCAL_FORECAST;

// Only change code above this line
```

USE DESTRUCTURING ASSIGNMENT TO ASSIGN VARIABLES FROM ARRAYS

ES6 makes destructuring arrays as easy as destructuring objects.

One key difference between the spread operator and array destructuring is that the spread operator unpacks all contents of an array into a comma-separated list. Consequently, you cannot pick or choose which elements you want to assign to variables.

Destructuring an array lets us do exactly that:

```
const [a, b] = [1, 2, 3, 4, 5, 6];
console.log(a, b);
```

The console will display the values of `a` and `b` as `1, 2`.

The variable `a` is assigned the first value of the array, and `b` is assigned the second value of the array. We can also access the value at any index in an array with destructuring by using commas to reach the desired index:

```
const [a, b,,, c] = [1, 2, 3, 4, 5, 6];
console.log(a, b, c);
```

The console will display the values of `a`, `b`, and `c` as `1, 2, 5`.

Use destructuring assignment to swap the values of `a` and `b` so that `a` receives the value stored in `b`, and `b` receives the value stored in `a`.

```
let a = 8, b = 6;
// Only change code below this line
[a,b]=[b,a]
console.log(a)
```

USE DESTRUCTURING ASSIGNMENT WITH THE REST PARAMETER TO REASSIGN ARRAY ELEMENTS

In some situations involving array destructuring, we might want to collect the rest of the elements into a separate array.

The result is similar to `Array.prototype.slice()`, as shown below:

```
const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];
console.log(a, b);
console.log(arr);
```

The console would display the values `1, 2` and `[3, 4, 5, 7]`.

Variables `a` and `b` take the first and second values from the array. After that, because of the rest parameter's presence, `arr` gets the rest of the values in the form of an array. The rest element only works correctly as the last variable in the list. As in, you cannot use the rest parameter to catch a subarray that leaves out the last element of the original array.

Use destructuring assignment with the rest parameter to perform an effective `Array.prototype.slice()` so that `arr` is a sub-array of the original array `source` with the first two elements omitted.

```
const source = [1,2,3,4,5,6,7,8,9,10];
function removeFirstTwo(list) {
  // Only change code below this line
  const [a, b, ...arr] = list; // Change this line
  // Only change code above this line
  return arr;
}
const arr = removeFirstTwo(source);
console.log(arr)
```

USE DESTRUCTURING ASSIGNMENT TO PASS AN OBJECT AS A FUNCTION'S PARAMETERS

In some cases, you can destructure the object in a function argument itself.

Consider the code below:

```
const profileUpdate = (profileData) => {
  const { name, age, nationality, location } = profileData;

}
```

This effectively destructures the object sent into the function. This can also be done in-place:

```
const profileUpdate = ({ name, age, nationality, location }) => {

}
```

When `profileData` is passed to the above function, the values are destructured from the function parameter for use within the function.

Use destructuring assignment within the argument to the function `half` to send only `max` and `min` inside the function.

```
const stats = {
  max: 56.78,
  standard_deviation: 4.34,
  median: 34.54,
  mode: 23.87,
  min: -0.75,
  average: 35.85
};

// Only change code below this line
//const half = (stats) => (stats.max + stats.min) / 2.0;
const half = ({min, max}) => (max + min) / 2.0;
console.log(half(stats))
```

```
// Only change code above this line

```

CREATE STRINGS USING TEMPLATE LITERALS

A new feature of ES6 is the template literal. This is a special type of string that makes creating complex strings easier.

Template literals allow you to create multi-line strings and to use string interpolation features to create strings.

Consider the code below:

```
const person = {
  name: "Zodiac Hasbro",
  age: 56
};

const greeting = `Hello, my name is ${person.name}!
I am ${person.age} years old.`;

console.log(greeting);
```

The console will display the strings `Hello, my name is Zodiac Hasbro!` and `I am 56 years old.`.

A lot of things happened there. Firstly, the example uses backticks (`` ` ``), not quotes (' or "), to wrap the string. Secondly, notice that the string is multi-line, both in the code and the output. This saves inserting \n` within strings. The {variable} syntax used above is a placeholder. Basically, you won't have to use concatenation with the + operator anymore. To add variables to strings, you just drop the variable in a template string and wrap it with { and }. Similarly, you can include other expressions in your string literal, for example ${a + b}. This new way of creating strings gives you more flexibility to create robust strings.

Use template literal syntax with backticks to create an array of list element (`li`) strings. Each list element's text should be one of the array elements from the `failure` property on the `result` object and have a `class` attribute with the value `text-warning`. The `makeList` function should return the array of list item strings.

Use an iterator method (any kind of loop) to get the desired output (shown below).

```
[
  '<li class="text-warning">no-var</li>',
  '<li class="text-warning">var-on-top</li>',
  '<li class="text-warning">linebreak</li>'
]
```

```
const result = {
  success: ["max-length", "no-amd", "prefer-arrow-functions"],
  failure: ["no-var", "var-on-top", "linebreak"],
  skipped: ["no-extra-semi", "no-dup-keys"]
};
```

```
function makeList(arr) {
  "use strict";
  // change code below this line
  const failureItems = [];
  for (let i = 0; i < arr.length; i++) {
    failureItems.push(`<li class="text-warning">${arr[i]}</li>`);
  }
  // change code above this line
  return failureItems;
}

const failuresList = makeList(result.failure);
```

WRITE CONCISE OBJECT LITERAL DECLARATIONS USING OBJECT PROPERTY SHORTHAND

ES6 adds some nice support for easily defining object literals.

Consider the following code:

```
const getMousePosition = (x, y) => ({
  x: x,
  y: y
});
```

`getMousePosition` is a simple function that returns an object containing two properties. ES6 provides the syntactic sugar to eliminate the redundancy of having to write `x: x`. You can simply write `x` once, and it will be converted to `x: x` (or something equivalent) under the hood. Here is the same function from above rewritten to use this new syntax:

```
const getMousePosition = (x, y) => ({ x, y });
```

Use object property shorthand with object literals to create and return an object with `name`, `age` and `gender` properties.

```
const createPerson = (name, age, gender) => {
  // Only change code below this line
  return({
    name,
    age,
    gender
  });
  // Only change code above this line
};
```

WRITE CONCISE DECLARATIVE FUNCTIONS WITH ES6

When defining functions within objects in ES5, we have to use the keyword `function` as follows:

```javascript
const person = {
  name: "Taylor",
  sayHello: function() {
    return `Hello! My name is ${this.name}.`;
  }
};
```

With ES6, you can remove the `function` keyword and colon altogether when defining functions in objects. Here's an example of this syntax:

```javascript
const person = {
  name: "Taylor",
  sayHello() {
    return `Hello! My name is ${this.name}.`;
  }
};
```

Refactor the function `setGear` inside the object `bicycle` to use the shorthand syntax described above.

```javascript
// Only change code below this line
const bicycle = {
  gear: 2,
  //setGear: function(newGear) {
    setGear (newGear) {
     this.gear = newGear;
   }
};
// Only change code above this line
bicycle.setGear(3);
console.log(bicycle.gear);
```

USE CLASS SYNTAX TO DEFINE A CONSTRUCTOR FUNCTION

ES6 provides a new syntax to create objects, using the class keyword.

It should be noted that the `class` syntax is just syntax, and not a full-fledged class-based implementation of an object-oriented paradigm, unlike in languages such as Java, Python, Ruby, etc.

In ES5, we usually define a `constructor` function and use the `new` keyword to instantiate an object.

```
var SpaceShuttle = function(targetPlanet){
  this.targetPlanet = targetPlanet;
}
var zeus = new SpaceShuttle('Jupiter');
```

The `class` syntax simply replaces the `constructor` function creation:

```
class SpaceShuttle {
  constructor(targetPlanet) {
    this.targetPlanet = targetPlanet;
  }
}
const zeus = new SpaceShuttle('Jupiter');
```

It should be noted that the `class` keyword declares a new function, to which a constructor is added. This constructor is invoked when `new` is called to create a new object.

Note: UpperCamelCase should be used by convention for ES6 class names, as in `SpaceShuttle` used above.

The `constructor` method is a special method for creating and initializing an object created with a class. You will learn more about it in the Object Oriented Programming section of the JavaScript Algorithms And Data Structures Certification.

Use the `class` keyword and write a `constructor` to create the `Vegetable` class.

The `Vegetable` class allows you to create a vegetable object with a property `name` that gets passed to the `constructor`.

```
// Only change code below this line
class Vegetable  {
  constructor(name) {
    this.name = name;
  }

}
// Only change code above this line

const carrot = new Vegetable('carrot');
console.log(carrot.name); // Should display 'carrot'
```

USE GETTERS AND SETTERS TO CONTROL ACCESS TO AN OBJECT

You can obtain values from an object and set the value of a property within an object.

These are classically called getters and setters.

Getter functions are meant to simply return (get) the value of an object's private variable to the user without the user directly accessing the private variable.

Setter functions are meant to modify (set) the value of an object's private variable based on the value passed into the setter function. This change could involve calculations, or even overwriting the previous value completely.

```
class Book {
  constructor(author) {
    this._author = author;
  }
  // getter
  get writer() {
    return this._author;
  }
  // setter
  set writer(updatedAuthor) {
    this._author = updatedAuthor;
  }
}
const novel = new Book('anonymous');
console.log(novel.writer);
novel.writer = 'newAuthor';
console.log(novel.writer);
```

The console would display the strings `anonymous` and `newAuthor`.

Notice the syntax used to invoke the getter and setter. They do not even look like functions. Getters and setters are important because they hide internal implementation details.

Note: It is convention to precede the name of a private variable with an underscore (`_`). However, the practice itself does not make a variable private.

Use the `class` keyword to create a `Thermostat` class. The `constructor` accepts a Fahrenheit temperature.

In the class, create a `getter` to obtain the temperature in Celsius and a `setter` to set the temperature in Celsius.

Remember that `C = 5/9 * (F - 32)` and `F = C * 9.0 / 5 + 32`, where `F` is the value of temperature in Fahrenheit, and `C` is the value of the same temperature in Celsius.

Note: When you implement this, you will track the temperature inside the class in one scale, either Fahrenheit or Celsius.

This is the power of a getter and a setter. You are creating an API for another user, who can get the correct result regardless of which one you track.

In other words, you are abstracting implementation details from the user.

```
// Only change code below this line
class Thermostat {

  constructor(fahrenheit) {
    this.fahrenheit = fahrenheit;
    }
  get temperature(){
    return (5/9) * (this.fahrenheit - 32);
    }

  set temperature(celsius){
      this.fahrenheit = (celsius * 9) / 5+32
    }
  }
// Only change code above this line

const thermos = new Thermostat(76); // Setting in
Fahrenheit scale
let temp = thermos.temperature; // 24.44 in Celsius
thermos.temperature = 26;
temp = thermos.temperature; // 26 in Celsius
```

USE * TO IMPORT EVERYTHING FROM A FILE

Suppose you have a file and you wish to import all of its contents into the current file. This can be done with the `import * as` syntax. Here's an example where the contents of a file named `math_functions.js` are imported into a file in the same directory:

```
import * as myMathModule from "./math_functions.js";
```

The above `import` statement will create an object called `myMathModule`. This is just a variable name, you can name it anything. The object will contain all of the exports from `math_functions.js` in it, so you can access the functions like you would any other object property. Here's how you can use the `add` and `subtract` functions that were imported:

```
myMathModule.add(2,3);
myMathModule.subtract(5,3);
```

The code in this file requires the contents of the file: `string_functions.js`, that is in the same directory as the current file. Use the `import * as` syntax to import everything from the file into an object called `stringFunctions`.

```
import * as stringFunctions from
'./string_functions.js';
// Only change code above this line

stringFunctions.uppercaseString("hello");
stringFunctions.lowercaseString("WORLD!");
```

CREATE AN EXPORT FALLBACK WITH EXPORT DEFAULT

In the `export` lesson, you learned about the syntax referred to as a named export. This allowed you to make multiple functions and variables available for use in other files.

There is another `export` syntax you need to know, known as export default. Usually you will use this syntax if only one value is being exported from a file. It is also used to create a fallback value for a file or module.

Below are examples using `export default`:

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

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

The first is a named function, and the second is an anonymous function.

Since `export default` is used to declare a fallback value for a module or file, you can only have one value be a default export in each module or file. Additionally, you cannot use `export default` with `var`, `let`, or `const`

The following function should be the fallback value for the module. Please add the necessary code to do so.

```
export default function subtract(x, y) {
  return x - y;
}
```

CREATE A JAVASCRIPT PROMISE

A promise in JavaScript is exactly what it sounds like - you use it to make a promise to do something, usually asynchronously. When the task completes, you either fulfill your promise or fail to do so. `Promise` is a constructor function, so you need to use the `new` keyword to create one. It takes a function, as its argument, with two parameters - `resolve` and `reject`. These are methods used to determine the outcome of the promise. The syntax looks like this:

```
const myPromise = new Promise((resolve, reject) => {

});
```

Create a new promise called `makeServerRequest`. Pass in a function with `resolve` and `reject` parameters to the constructor.

```
const makeServerRequest = new Promise ((resolve,
reject) => {

});
```

COMPLETE A PROMISE WITH RESOLVE AND REJECT

A promise has three states: `pending`, `fulfilled`, and `rejected`. The promise you created in the last challenge is forever stuck in the `pending` state because you did not add a way to complete the promise. The `resolve` and `reject` parameters given to the promise argument are used to do this. `resolve` is used when you want your promise to succeed, and `reject` is used when you want it to fail. These are methods that take an argument, as seen below.

```
const myPromise = new Promise((resolve, reject) => {
  if(condition here) {
    resolve("Promise was fulfilled");
  } else {
    reject("Promise was rejected");
  }
});
```

The example above uses strings for the argument of these functions, but it can really be anything. Often, it might be an object, that you would use data from, to put on your website or elsewhere.

Make the promise handle success and failure. If `responseFromServer` is `true`, call the `resolve` method to successfully complete the promise. Pass `resolve` a string with the value `We got the data`. If `responseFromServer` is `false`, use the `reject` method instead and pass it the string: `Data not received`.

```javascript
const makeServerRequest = new Promise((resolve, reject) => {
  // responseFromServer represents a response from a server
  let responseFromServer;

  if(responseFromServer) {
    // Change this line
    resolve(' We got the data')
  } else {
    // Change this line
    reject ('Data not received')
  }
});
```