web222

WEB222 - Week 2

Suggested Readings

- SpeakingJS, Chapter 15. Functions and Chapter 16. Variables: Scopes, Environments, and Closures
- Eloquent JavaScript, Chapter 3. Functions
- Functions Guide and Reference on MDN.

Functions

A function is a *subprogram*, or a smaller portion of code that can be called (i.e., invoked) by another part of your program, another function, or by the environment in response to some user or device action (e.g., clicking a button, a network request, the page closing). Functions *can* take values (i.e., arguments) and may *return* a value.

Functions are first-class members of JavaScript, and play a critical role in developing JavaScript programs. JavaScript functions can take other functions as arguments, can return functions as values, can be bound to variables or `Object` properties, and can even have their own properties. We'll talk about more of this when we visit JavaScript's object-oriented features.

Learning to write code in terms of functions takes practice. JavaScript supports functional programming. Web applications are composed of lots of small components that need to get wired together using functions, have to share data (i.e., state), and interoperate with other code built into the browser, or in third-party frameworks, libraries, and components.

We use JavaScript functions in a number of ways. First, we encapsulate a series of statements into higher-order logic, giving a name to a set of repeatable steps we can call in different ways and places in our code. Second, we use them to define actions to be performed in

response to events, whether user initiated or triggered by the browser. Third, we use them to define behaviours for objects, what is normally called a *member function* or *method*. Fourth, we use them to define *constructor* functions, which are used to create new objects. We'll look at all of these in the coming weeks.

Before we dive into that, we'll try to teach you that writing many smaller functions is often better than having a few large ones. Smaller code is easier to test, easier to understand, and generally has fewer bugs.

User-defined Functions

JavaScript has many built-in functions, which we'll get to below; however, it also allows you to write your own and/or use functions written by other developers (libraries, frameworks). These user-defined functions can take a number of forms.

Function Declarations

The first is the *function declaration*, which looks like this:

```
// The most basic function, a so-called NO OPERATION function
function noop() {
}

// square function accepts one parameter `n`, returns its value squared.
function square(n) {
    return n * n;
}

// add function accepts two parameters, `a` and `b`, returns their sum.
function add(a, b) {
    return a + b;
}
```

Here the `function` keyword initiates a *function declaration*, followed by a *name*, a *parameter list* in round parenthesis, and the function's *body* surrounded by curly braces. There is no semi-colon after the function body.

Function Expressions

The second way to create a function is using a *function expression*. Recall that expressions evaluate to a value: a function expression evaluates to a `function` Object. The resulting value is often bound (i.e., assigned) to a variable, or used as a parameter.

```
var noop = function() {};

var square = function(n) {
    return n * n;
};

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

A few things to note:

- The function's *name* is often omitted. Instead we return an *anonymous function* and bind it to a variable. We'll access it again via the variable name later. In the case of recursive functions, we sometimes include it to make it easier for functions to call themselves. You'll see it done both ways.
- We *did* use a semi-colon at the end of our function expression. We do this to signify the end of our assignment statement `var add = ... ;`.
- In general, *function declarations* are likely a better choice (when you can choose) due to subtle errors introduced with declaration order and hosting (see below); however, both are used widely and are useful.

> JavaScript version note: newer versions of JavaScript also include the new `=>` notation, which denotes an Arrow Function. When you see `var add = (a, b) => a + b;` it is short-hand for `var add = function(a, b) { return a + b; }`, where `=>` replaces the `function` keyword and comes *after* the parameter list, and the `return` keyword is optional when functions return a single value). Arrow functions also introduce some new semantics for the `this` keyword, which we'll address later.

Parameters and `arguments`

Function definitions in both cases take parameter lists, which can be empty, single, or multiple in length. Just as with variable declaration, no type information is given:

```
function emptyParamList() {
}

function singleParam(oneParameter) {
}

function multipleParams(one, two, three, four) {
}
```

A function can *accept* any number of arguments when it is called, including none. This would break in many other languages, but not JavaScript:

```
function log(a) {
    console.log(a);
}

log("correct");          // logs "correct"
log("also", "correct");  // logs "also"
log();                   // logs undefined
```

Because we can invoke a function with any number of arguments, we have to write our functions carefully, and test things before we make assumptions. How can we deal with a caller sending 2 vs. 10 values to our function?

One way we do this is using the built-in `arguments` Object. Every function has an implicit `arguments` variable available to it, which is an array-like object containing all the arguments passed to the function. We can use `arguments.length` to obtain the actual number of arguments passed to the function at runtime, and use array index notation (e.g., `arguments[0]`) to access an argument:

```
function log(a) {
    console.log(arguments.length, a, arguments[0]);
```

```
}

log("correct");          // 1, "correct", "correct"
log("also", "correct");  // 2, "also", "also"
log();                   // 0, undefined, undefined
```

We can use a loop to access all arguments, no matter the number passed:

```
function sum() {
    var count = arguments.length;
    var total = 0;
    for(var i = 0; i < count; i++) {
        total += arguments[i];
    }
    return total;
}

sum(1);          // 1
sum(1, 2);       // 3
sum(1, 2, 3, 4); // 10
```

You may have wondered previously how `console.log()` can work with one, two, three, or more arguments. The answer is that all JavaScript functions work this way, and you can use it to "overload" your functions with different argument patterns, making them useful in more than one scenario.

> JavaScript version note: in newer versions of JavaScript, we can also use Rest Parameters, which allow us to specify that all final arguments to a function, no matter how many, should appear within the function as an `Array`. There are some advantages to *not* using `arguments`, which rest parameters provide. We can convert the example above to:

```
function sum(...numbers) {
    var total = 0;
    for(var i = 0; i < numbers.length; i++) {
        total += numbers[i];
```

```
    }
    return total;
}
```

Dealing with Optional and Missing Arguments

Because we *can* change the number of arguments we pass to a function at runtime, we also have to deal with missing data, or optional parameters. Consider the case of a function to calculate a player's score in a video game. In some cases we may want to double a value, for example, as a bonus for doing some action a third time in a row:

```
function updateScore(currentScore, value, bonus) {
    return bonus ? currentScore + value * bonus : currentScore + value;
}

updateScore(10, 3);
updateScore(10, 3);
updateScore(10, 3, 2);
```

Here we call `updateScore` three different times, sometimes with 2 arguments, and once with 3. Our `updateScore` function has been written so it will work in both cases. We've used a conditional ternary operator to decide whether or not to add an extra bonus score. When we say `bonus ? ... : ...` we are checking to see if the `bonus` argument is *truthy* or *falsy*–did the caller provide a value for it? If they did, we do one thing, if not, we do another.

Here's another common way you'll see code like this written, using a default value:

```
function updateScore(currentScore, value, bonus) {
    // See if `bonus` is truthy (has a value or is undefined) and use it, or default to 1
    bonus = bonus || 1;
    return currentScore + value * bonus;
}
```

In this case, before we use the value of `bonus`, we do an extra check to see if it actually has a value or not. If it does, we use that value as is; but if it doesn't, we instead assign it a value of `1`. Then, our calculation will always work, since multiplying the value by `1` will be the same as not using a bonus.

The idiom `bonus = bonus || 1` is very common in JavaScript. It uses the Logical Or Operator `||` to test whether `bonus` evaluates to a value or not, and prefers that value if possible to the fallback default of `1`. We could also have written it out using an `if` statements like these:

```
function updateScore(currentScore, value, bonus) {
    if(bonus) {
        return currentScore + value * bonus;
    }
    return currentScore + value;
}

function updateScore(currentScore, value, bonus) {
    if(!bonus) {
        bonus = 1;
    }
    return currentScore + value * bonus;
}
```

JavaScript programmers tend to use the `bonus = bonus || 1` pattern because it is less repetitive, using less code, and therefore less likely to introduce bugs. We could shorten it even further to this:

```
function updateScore(currentScore, value, bonus) {
    return currentScore + value * (bonus || 1);
}
```

> JavaScript version note: newer versions of JavaScript also support Default Parameters, which allows us to specify a default value for any named parameter when declared. This frees us from having to check for, and set default values in the function body. Using default parameters, we could convert our code above to this:

```
function updateScore(currentScore, value, bonus = 1) {
    return currentScore + value * bonus;
}
```

Return Value

Functions always *return* a value, whether implicitly or explicitly. If the `return` keyword is used, the expression following it is returned from the function. If it is omitted, the function will return `undefined`:

```
function implicitReturnUndefined() {
    // no return keyword, the function will return `undefined` anyway
}

function explicitReturnUndefined() {
    return;
    // return keyword, but no expression given, which is also `undefined`
}

function explicitReturn() {
    return 1;
    // return keyword, followed by `Number` expression evalutes to `Number`
}

function explicitReturn2() {
    return "Hello" + " World!";
    // return keyword, followed by expression evaluating to a `String`
}
```

Function Naming

Functions are typically named using the same rules we learned for naming any variable: `camelCase` and using the set of valid letters, numbers, etc. and avoiding language keywords.

Function declarations always give a name to the function, while function expressions often omit it, using a variable name instead:

```
// Name goes after the `function` keyword in a declaration
function validateUser() {
    ...
}

// Name is used only at the level of the bound variable, function is anonymous
var validateUser = function() {
    ...
};

// Name is repeated, which is correct but not common. Used with recursive functions
var validateUser = function validateUser() {
    ...
};

// Names are different, which is also correct, but not common as it can lead to confusion
var validateUser = function validate() {
    // the validate name is only accessible here, within the function body
    ...
};
```

Because JavaScript allows us to bind function objects (i.e., result of function expressions) to variables, it is common to create functions without names, but immediately pass them to functions as arguments. The only way to use this function is via the argument name:

```
// The parameter `fn` will be a function, and `n` a number
function execute(fn, n) {
    // Call the function referred to by the argument (i.e, variable) `fn`, passing `n` as its argument
    return fn(n);
}

// 1. Call the `execute` function, passing an anonymous function, which squares its argument, and the value 3
execute(function(n) {
```

```
    return n * n;
}, 3);


// 2. Same thing as above, but with different formatting
execute(function(n) { return n * n;}, 3);

// 3. Using ES6 Arrow Function syntax
execute((n) => n * n, 3);

var doubleIt = function(num) {
    return num * 2;
}

// 4. Again call `execute`, but this time pass `doubleIt` as the function argument
execute(doubleIt, 3);
```

We can also use functions declared via function declarations used this way, and bind them to variables:

```
function greeting(greeting, name) {
    return greeting + " " + name;
}

var sayHi = greeting;  // also bind a reference to greeting to sayHi

// We can now call `greeting` either with `greeting()` or `sayHi()`
console.log(greeting("Hello", "Steven"));
console.log(sayHi("Hi", "Kim"));
```

JavaScript treats functions like other languages treat numbers or booleans, and lets you use them as values. This is a very powerful feature, but can cause some confusion as you get started with JavaScript.

You might ask why we would ever choose to define functions using variables. One common reason is to swap function implementations at runtime, depending on the state of the program. Consider the following code for displaying the user interface depending on whether the

user is logged in or not:

```
// Display partial UI for guests and non-authenticated users, hiding some features
function showUnauthenticatedUI() {
    ...
}

// Display full UI for authenticated users
function showAuthenticatedUI() {
    ...
}

// We will never call showUnauthenticatedUI or showAauthenticatedUI directly.
// Instead, we will use showUI to hold a reference to one or the other,
// and default to the unauthenticated version at first (i.e., until the user logs in).
var showUI = showUnauthenticatedUI;

...

// Later in the program, when a user logs in, we can swap the implemenation
// without touching any of our UI code.
function authenticate(user) {
    ...
    showUI = showAuthenticatedUI;
}

...

// Whenever we need to refresh/display the UI, we can always safely call
// whichever function is currently bound to `showUI`.
showUI();
```

Invoking Functions, the Execution Operator

In many of the examples above, we've been invoking (calling, running, executing) functions but haven't said much about it. We invoke a function by using the `()` operator:

```
var f = function() { console.log('f was invoked'); };
f();
```

In the code above, `f` is a variable that is assigned the value returned by a function expression. This means `f` is a regular variable, and we can use it like any other variable. For example, we could create another variable and share its value:

```
var f = function() { console.log('f was invoked'); };
var f2 = f;
f();       // invokes the function
f2();      // also invokes the function
```

Both `f` and `f2` refer to the the same function object. What is the difference between saying `f` vs. `f()` in the line `var f2 = f;` ? When we write `f()` we are really saying, "Get the value of `f` (the function referred to) and invoke it." However, when we write `f` (without `()`), we are saying, "Get the value of `f` (the function referred to)" so that we can do something with it (assign it to another variable, pass it to a function, etc).

The same thing is true of function declarations, which also produce `function` Objects:

```
function f() { console.log('f was invoked'); };
var f2 = f;
f2();      // also invokes the function
```

The distinction between referring to a function object via its bound variable name (`f`) vs invoking that same function (`f()`) is important, because JavaScript programs treat functions as *data*, just as you would a `Number` . Consider the following:

```
function checkUserName(userName, customValidationFn) {
    // If `customValidationFn` exists, and is a function, use that to validate `userName`
```

```
    if(customValidationFn && typeof customValidationFn === 'function') {
        return customValidationFn(userName);
    }
    // Otherwise, use a default validation function
    return defaultValidationFn(userName);
}
```

Here the `checkUserName` function takes two arguments: the first a `String` for a username; the second an optional (i.e., may not exist) function to use when validating this username. Depending on whether or not we are passed a function for `customValidationFn`, we will either use it, or use a default validation function (defined somewhere else).

Notice the line `if(customValidationFn && typeof customValidationFn === 'function') {` where `customValidationFn` is used like any other variable (accessing the value it refers to vs. doing an invocation), to check if it has a value, and if its value is actually a function. Only then is it save to invoke it.

It's important to remember that JavaScript functions aren't executed until they are called via the invocation operator, and may also be used as values without being called.

Built-in/Global Functions

JavaScript provides a number of built-in global functions for working with its data types, for example:

- `parseInt()`
- `parseFloat()`
- `isNaN()`
- `isFinite()`
- `decodeURI()`
- `decodeURIComponent()`
- `encodeURI()`
- `encodeURIComponent()`

There are also global functions that exist for historical reasons, but should be avoided for performance, usability, and/or security reasons:

- `eval()` dangerous to parse and run user-defined strings
- `prompt()` and `alert()` synchronous calls that block the UI thread.

Most of JavaScripts "standard library" comes in the form of *methods* on global objects vs. global functions. A *method* is a function that is bound to a variable belonging to an object, also known as a *property*. We'll be covering these in more depth later, but here are some examples

- `console.*` . There are quite a few worth learning, but here are some to get you started:
  - `console.log()` , `console.warn()` , and `console.error()`
  - `console.assert()`
  - `console.count()`
  - `console.dir()`
- `Math.*`
  - `Math.abs()`
  - `Math.max()`
  - `Math.min()`
  - `Math.random()`
  - `Math.round()`
- `Date.*`
  - `Date.now()`
  - `Date.getTime()`
  - `Date.getMonth()`
  - `Date.getDay()`
- `JSON.*`
  - `JSON.parse()`
  - `JSON.stringify()`

Scope

JavaScript variables are *declared* with the `var` keyword (or `let`, `const` in es6). We often *assign* a value when we *declare* it, though we don't have to do both at once:

```
var x;      // declared, no assignment (value is `undefined`)
x = 7;      // assignment of previously declared variable
var y = x;  // declaration and assignment combined
```

A variable always has a *scope*, which is the location(s) in the code where it is usable. Consider the variables `total` and `value`, as well as the `add` function below:

```
var total = 7;                    // global variable, accessible everywhere

function add(n) {
    var value = total + n;        // local variable, accessible within `add` function only
    return value;
}

console.log("Total is", total);   // Works, because `total` is in the same scope
console.log("Value is", value);   // `undefined`, since `value` isn't defined in this scope
console.log("New Total", add(16)) // Works, because `add` is defined in the same scope
```

Unlike most programming languages, which use *block scope*, JavaScript variables have *function scope*:

```
int main()
{
  {
      int x = 10;       // x is declared with block scope
  }
  {
      printf("%d", x);  // Error: x is not accessible here
```

```
    }
    return 0;
}
```

Now in JavaScript:

```
function main() {
    {
        var x = 10;     // x is declared in a block, but is scoped to `main`
    }
    {
        console.log(x); // works, because `x` is accessible everywhere in `main`
    }
}
```

In many languages, we are told to declare variables when we need them. However, in JavaScript we tend to define our variables at the top of our functions. We don't strictly need to do this, due to *hoisting*. JavaScript will *hoist* or raise all variable declarations it finds in a function to the top of their scope:

```
function f() {
    var y = x + 1;
    var x = 2;
}
```

At runtime, this will be transformed into the following:

```
function f() {
    var x;          // declaration is hoisted (but not assignment) to the top

    var y = x + 1;  // `NaN`, since `undefined` + 1 can't be resolved
    x = 2;          // note: `x` is not declared above, only the assignment is now here
```

This also happens when we forget to declare a local variable:

```
function f() {
    x = 2;          // `x` is assigned a value, but not declared
    return x + 1;
}
```

At runtime, this will be transformed into the following:

```
var x;              // `x` is not found in the scope of `f`, so it becomes global

function f() {
    x = 2;
    return x + 1;
}
```

The previous example introduces another important concept with JavaScript scopes, namely, that scopes can be *nested* within one another. Hoisting is moving variable declarations to the beginning of a scope. For example, function declarations are hoisted completely, which means we can call a function *before* we declare it.

```
f(); // this will work, as f's declaration gets hoisted
function f() {
}
f(); // this will also work, because f has been declared as you expect.

g(); // this will not work, since g's declaration will be hoisted, but not the assignment.
var g = function() {};
```

In general, declare and define things *before* you need them.

Overwriting Variables in Child Scopes

Since variables have function scope, and because functions can be nested, we have to be careful when naming our variables and arguments so as to not overwrite a variable in a parent scope. Or, we can use this to temporarily do exactly that. In both cases, we need to understand how nested scopes work. Consider the the following code, where a variable named `x` is used in three different scopes. What will be printed to the `console` when `child` is called?

```
var x = 1;

function parent() {
    var x = 2;

    function child(x) {
        console.log(x);
    }

    child(3);
}
```

The first declaration of `x` creates a global variable (i.e., available in every scope). Then, in `parent` we re-declare `x`, creating a new local variable, which overwrites (or hides) the global variable `x` in this scope (i.e., within the body of `parent`). Next, we define yet another scope for `child`, which also uses `x` as the name of its only argument (essentially another local variable). When we do `child(3)`, we are binding the value `3` to the `x` argument defined for the scope of `child`, and in so doing yet again overwriting the parent `x`. In the end, the console will show `3`.

We can do this in error as well, and cause unexpected behaviour:

```
var total = 100;

function increase(n) {
    var total = n + n;
}

increase(50);
console.log(total);
```

Here we expect to see `150` but instead will get `100` on the `console`. The problem is that we have redefined, and thus overwritten `total` inside the `increase` function. During the call to `increase`, the new local variable `total` will be used, and then go out of scope. After the function completes, the original global variable `total` will again be used.

Closures

A closure is a function that has *closed over* a scope, retaining it even after it would otherwise disappear through the normal rules of execution. In the following function, the variable `x` goes out of scope as soon as the function finishes executing:

```
function f() {
    var x = 7;
    return x * 2;
    // After this return, and f completes, `x` will no longer be available.
}
```

In JavaScript, functions have access not only to their own local variables, but also to any functions in their parents' scope. That is, if a function is used (referenced) but not declared in a function, JavaScript will visit the parent scope to find the variable. This can happen for any number of child/parent levels up to the global level.

The following is an example of this, and probably one you've seen before:

```
var x = 7;

function f() {
    return x * 2;  // `x` not declared here, JS will look in the parent scope (global)
}
```

Consider this example:

```
function parent() {
    var x = 7;

    function child() {
        return x * 2;
    }

    return child();
}
```

Here `x` is used in `child`, but declared in `parent`. The `child` function has access to all variables in its own scope, plus those in the `parent` scope. This nesting of scopes relies on JavaScript's function scope rules, and allows us to share data.

Sometimes we need to capture data in a parent scope, and retain it for a longer period of time than would otherwise be granted for a given invocation. Consider this example:

```
function createAccumulator(value) {
    return function(n) {
        value += n;
        return value;
    };
}

var add = createAccumulator(10);
add(1)   // returns 11
add(2)   // returns 13
```

Here the `createAccumulator` function takes an argument `value`, the initial value to use for an accumulator function. It returns an anonymous function which takes a value `n` (a `Number`) and adds it to the `value` before returning it. The `add` function is created by invoking `createAccumulator` with the initial `value` of `10`. The function that is returned by `createAccumulator` has access to `value` in its parent's scope. Normally, `value` would be destroyed as soon as `createAccumulator` finished executing. However, we have created a *closure* to capture the variable `value` in a scope that is now attached to the function we're creating and returning. As long as the returned

function exists (i.e., as long as `add` holds on to it), the variable `value` will continue to exist in our child function's scope: the variables that existed when this function was created continue to live on like a memory, attached to the lifetime of the returned function.

Closures make it possible to *associate* some *data* (i.e., the environment) with a function that can then operate on that data. We see similar strategies in pure object-oriented languages, where data (properties) can be associated with an object, and functions (methods) can then operate on that data. Closures play a somewhat similar role, however, they are more lightweight and allow for dynamic (i.e., runtime) associations.

By connecting data and functionality, closures help to reduce global variables, provide ways to "hide" data, allow a mechanism for creating private "methods", avoid overwriting other variables in unexpected ways.

As we go further with JavaScript and web programming, we will encounter many instances where closures can be used to manage variable lifetimes, and associated functions with specific objects. For now, be aware of their existence, and know that it is an advanced concept that will take some time to fully master. This is only our first exposure to it.

Another way we'll see closures used, is in conjunction with Immediately-Invoked Function Expressions (IIFE). Consider the following rewrite of the code above:

```
var add = (function(value) {
    return function(n) {
        value += n;
        return value;
    };
})(10);

add(1)   // returns 11
add(2)   // returns 13
```

Here we've declared `add` to be the value of invoking the anonymous function expression written between the first `(...)` parentheses. In essence, we have created a function that gets executed immediately, and which returns another function that we will use going forward in our program.

This is an advanced technique to be aware of at this point, but not one you need to master right away. We'll see it used, and use it ourselves, in later weeks to to avoid global variables, simulate block scope in JavaScript, and to choose or generate function implementations at runtime (e.g., polyfill).

Practice Exercises

For each of the following, write a function that takes the given arguments, and returns or produces (e.g., `console.log`) the given result.

1. Given `r` (radius) of a circle, calculate the area of a circle (A = π * r * r).
2. Simulate rolling a dice using `random()`. The function should allow the caller to specify any number of sides, but default to 6 if no side count is given: `roll()` (assume 6 sided, return random number between 1 and 6) vs. `roll(50)` (50 sided, return number between 1 and 50).
3. Write a function that converts values in Celcius to Farenheit: `convert(0)` should return `"32 F"`.
4. Modify your solution to the previous function to allow a second argument: `"F"` or `"C"`, and use that to determine what the scale of the value is, converting to the opposite: `convert(122, "F")` should return `"50 C"`.
5. Function taking any number of arguments (`Number`s), returning `true` if they are all less than 50: `isUnder50(1, 2, 3, 5, 4, 65)` should return `false`.
6. Function allowing any number of arguments (`Number`s), returning their sum: `sum(1, 2, 3)` should return `6`.
7. Function allowing any number of arguments of any type, returns `true` only if none of the arguments is falsy. `allExist(true, true, 1)` should return `true`, but `allExist(1, "1", 0)` should return `false`.
8. Function to create a JavaScript library name generator: `generateName("dog")` should return `"dog.js"`
9. Function to check if a number is a multiple of 3 (returns `true` or `false`)
10. Check if a number is between two other numbers, being inclusive if the final argument is true: `checkBetween(66, 1, 50, true)` should return `false`.
11. Function to calculate the HST (13%) on a purchase amount
12. Function to subtract a discount % from a total. If no % is given, return the original value.

13. Function that takes a number of seconds as a `Number`, returning a `String` formatted like `"X Days, Y Hours, Z Minutes"` rounded to the nearest minute.
14. Modify your solution above to only include units that make sense: `"1 Minute"` vs. `"3 Hours, 5 Minutes"` vs. `"1 Day, 1 Hour, 56 Minutes"` etc
15. Function that takes any number of arguments (`Number`s), and returns them in reverse order, concatenated together as a String: `flip(1, 2, 3)` should return `"321"`
16. Function that takes two `Number`s and returns their sum as an `Integer` value (i.e., no decimal portion): `intSum(1.6, 3.333333)` should return `4`
17. Function that returns the number of matches found for the first argument in the remaining arguments: `findMatches(66, 1, 345, 2334, 66, 67, 66)` should return `2`
18. Function to log all arguments larger than `255`: `showOutsideByteRange(1, 5, 233, 255, 256, 0)` should log `256` to the `console`
19. Function that takes a `String` and returns its value properly encoded for use in a URL. `prepareString("hello world")` should return `"hello%20world"`
20. Using the previous function, write an enclosing function that takes any number of `String` arguments and returns them in encoded form, concatenated together like so: `"?...&...&..."` where "..." are the encoded strings. `buildQueryString("hello world", "goodnight moon")` should return `"?hello%20world&goodnight%20moon"`
21. Function that takes a `Function` followed by any number of `Number`s, and applies the function to all the numbers, returning the total: `applyFn(function(x) { return x * x;}, 1, 2, 3)` should return 14.
