Lecture Template

Functions

Topics

- Declaring a Function
- Multiple Parameters
- Return Statements
- Return Values
- Function Scope
- Local Scope
- Global Scope

Functions

Declaring a Function

- **Functions** allow you to package up lines of code that you can use (and often reuse) in your programs.
- Sometimes they take **parameters** like the pizza button from the beginning of this lesson. `reheatPizza()` had one parameter: the number of slices.

```
function reheatPizza(numSlices) {
  // code that figures out reheat settings!
}
```

- The `reverseString()` function that you saw also had one parameter: the string to be reversed.

```
function reverseString(reverseMe) {
  // code to reverse a string!
}
```

Multiple Parameters

- In both cases, the parameter is listed as a variable after the function name, inside the parentheses. And, if there were multiple parameters, you would just separate them with commas.

```
function doubleGreeting(name, otherName)
```

- But, you can also have functions that don't have any parameters. Instead, they just package up some code and perform some task. In this case, you would just leave the parentheses empty. Take this one for example. Here's a simple function that just prints out "Hello!".

```
// accepts no parameters! parentheses are empty
function sayHello() {
  var message = "Hello!"
  console.log(message);
}
```

Return Statements

- In the `sayHello()` function above, a value is **printed** to the console with `console.log`, but not explicitly returned with a **return statement**. You can write a return statement by using the `return` keyword followed by the expression or value that you want to return.

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!"
  return message; // returns value instead of printing it
}
```

Now, to get your function to *do something*, you have to **invoke** or **call** the function using the function name, followed by parentheses with any **arguments** that are passed into it. Functions are like machines. You can build the machine, but it won't do anything unless you also turn it on. Here's how you would call the `sayHello()` function from before, and then use the return value to print to the console:

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!"
  return message; // returns value instead of printing it
}

// function returns "Hello!" and console.log prints the return value
console.log(sayHello());
```

> ***Prints:*** *"Hello!"*

How to run a Function

Parameters vs Arguments

- At first, it can be a bit tricky to know when something is either a parameter or an argument. The key difference is in where they show up in the code. A **parameter** is always going to be a *variable* name and appears in the function declaration. On the other hand, an **argument** is always going to be a *value* (i.e. any of the JavaScript data types - a number, a string, a boolean, etc.) and will always appear in the code when the function is called or invoked.

Function Recap

Functions Recap

- **Functions** package up code so you can easily use (and reuse) a block of code. **Parameters** are variables that are used to store the data that's passed into a function for the function to use. **Arguments** are the actual data that's passed into a function when it is invoked:

- The **function body** is enclosed inside curly brackets:

```
function add(x, y) {
  // function body!
}
```

```
// x and y are parameters in this function declaration
function add(x, y) {
  // function body
  // Here, `sum` variable has a scope within the function.
  // Such variables defined within a function are called Local variables
  // You can try giving it another name
  var sum = x + y;
  return sum; // return statement
}

// 1 and 2 are passed into the function as arguments,
// and the result returned by the function is stored in a new variable `sum`
// Here, `sum` is another variable, different from the one used inside the fun
var sum = add(1, 2);
```

Functions continued..

Return Statements

- The **function body** is enclosed inside curly brackets:

```
function add(x, y) {
  // function body!
}
```

- You **invoke** or **call** a function to have it do something:

```
add(1, 2);
```

> *Returns: 3*

Return Values

Return Values

- It's important to understand that **return** and **print** are not the same thing. Printing a value to the JavaScript console only displays a value (that you can view for debugging purposes), but the value it displays can't really be used for anything more than that. For this reason, you should remember to only use `console.log` to test your code in the JavaScript console.

```
function isThisWorking(input) {
  console.log("Printing: isThisWorking was called and " + input + " was passed
  return "Returning: I am returning this string!";
}

isThisWorking(3);
```

Prints: *"Printing: isThisWorking was called and 3 was passed in as an argument"*

Returns: *"Returning: I am returning this string!"*

Using Return Values

- Returning a value from a function is great, but what's the use of a return value if you're not going to use the value to do something?
- *A function's return value can be stored in a variable or reused throughout your program as a function argument.*

```
// returns the sum of two numbers
function add(x, y) {
  return x + y;
}
```

```
// returns the sum of two numbers
function add(x, y) {
  return x + y;
}


// returns the value of a number divided by 2
function divideByTwo(num) {
  return num / 2;
}


var sum = add(5, 7); // call the "add" function and store the returned value i
var average = divideByTwo(sum); // call the "divideByTwo" function and store t
```

Using Return Values cont...

Scope

Function Scope

In JavaScript there are two types of scope:

- Local scope
- Global scope

JavaScript has function scope: Each function creates a new scope.

Scope determines the accessibility (visibility) of these variables.

Variables defined inside a function are not accessible (visible) from outside the function.

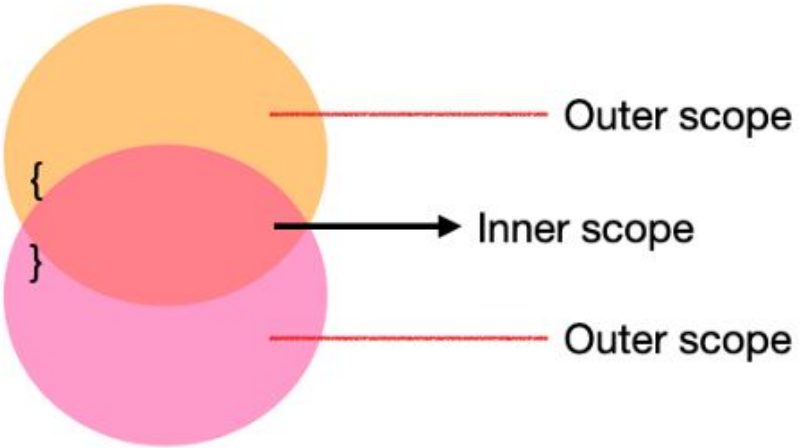
Local Scope

- Variables declared within a JavaScript function, become **LOCAL** to the function.
  Local variables have Function scope: They can only be accessed from within the function.
- Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.
- Local variables are created when a function starts, and deleted when the function is completed.

```
// code here can NOT use carName

function myFunction() {
  var carName = "Volvo";

  // code here CAN use carName

}
```

Shadowing

- What is ‘variable shadowing’? Variable shadowing occurs when a variable of an inner scope is defined with the same name as a variable in the outer scope. In the inner scope, both variables’ scope overlap.
- ‘Variable shadowing’ should be avoided



```
var bookTitle = "Le Petit Prince";
console.log(bookTitle);

function displayBookEnglish() {
  bookTitle = "The Little Prince";
  console.log(bookTitle);
}

displayBookEnglish();
console.log(bookTitle);
```

- same variable being assigned the same value in two different scopes.
- console.log is in global scope, not function scope.
- function scope has reassigned the global scope variable value, even though we exited function

Shadowing - Variable Overriding

```
var bookTitle = "Le Petit Prince";
console.log(bookTitle);

function displayBookEnglish() {
  var bookTitle = "The Little Prince";
  console.log(bookTitle);
}

displayBookEnglish();
console.log(bookTitle);
```

- To prevent shadowing from happening, we are going to simply declare a new variable inside the function
- We are going to create a different variable inside the function, this will be relative to function scope
- Then, the bookTitle that is declared in global scope will remain unchanged
- When we reach console.log, it will output as expected

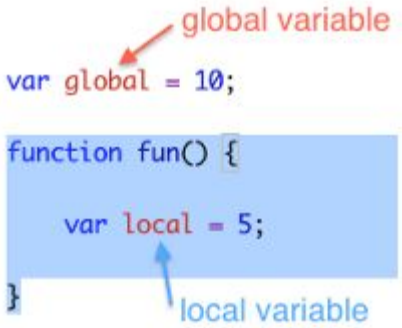
Shadowing - Variable Overriding cont..

Global Scope

- The global scope is the outermost scope. It is accessible from any inner (aka *local*) scope.
- A variable declared inside the global scope is named ***global*** variable. Global variables are accessible from any scope.
- `counter` is a global variable. This variable can be accessed from any place of the webpage's JavaScript.
- The global scope is a mechanism that lets the host of JavaScript (browser, Node) supply applications with host-specific functions as global variables.
- `window` and `document`, for example, are global variables supplied by the browser. In

```
// "global" scope
let counter = 1;
```

Scope Recap

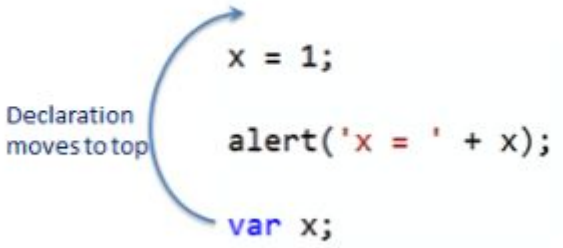


- If an identifier is declared in **global scope**, it's available *everywhere*.
- If an identifier is declared in **function scope**, it's available *in the function* it was declared in (even in functions declared inside the function).
- When trying to access an identifier, the JavaScript Engine will first look in the current function. If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there. It will keep doing this until it reaches the global scope.
- Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

Hoisting

Hoisting

- In most programming languages variables and functions need to be declared first, before you use them.
- Hoisting is a Js mechanism where variables and function declarations are moved to the top of their scope before code executes.
- This part is done when the program is being compiled (before the execution part).
- The code you write will remain the same, however the position of the declarations may change when compiled.



Hoisting Recap

- JavaScript hoists function declarations and variable declarations to the top of the current scope.
- Variable *assignments* are not hoisted.
- Declare functions and variables at the top of your scripts, so the syntax and behavior are consistent with each other.

```
var declaredLater;

// Outputs: undefined
console.log(declaredLater);

declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```