**When to Use Semicolons ;**

In JavaScript a semicolon should be used to mark the end of a ***statement***. A JavaScript statement is an instruction to be executed. Semicolons separate JavaScript statements.

**Functions**

Functions also called subroutines/subprograms/methods, exist in most programming languages. Functions are blocks of code that have a name and perform a specific task. Simply by invoking or calling the name of the function, the code inside of it will run or be executed. Generally a function would contain code that serves a particular purpose/perform a specific task. Functions can take in some inputs and perform a task and/or return an output.

So far, every time that you have written JavaScript code it has been executed (top to bottom) when the page loads. Using functions we can delay when certain code will run or set it to only run after a certain thing has happened (an event) or make it run numerous times while feeding different inputs into the function.

Using functions makes the code easier to read and allows you to reuse the code each time you need to perform the same task. This is possible because all you need to do is call the name of the function whenever you want.

Reasons to use functions:

1. Provides abstraction.
2. Allows multiple programmers to work on a problem.
3. Allows you to reuse your work.
4. Makes it easier to identify structures.

Functions also go by the name of method or subroutine or procedure.

A basic function written in JavaScript would look like this. It is a common syntax for defining a function. In the below example a function is being **declared/defined**.

**function myFuncName() {**

**//** *Code for function goes here*

**//** *between the curly brackets*

**}**

Below is an example of the above function being **called** or **invoked** or **executed** meaning that the code inside its code block will be run.

**myFuncName();**

Here it is again but with the specific parts highlighted.

**function myFuncName ( ) {**

**//** *Code for function goes here*

**}**

* **-function-** is the keyword used to indicate that a function is being declared.
* **-myFuncName-** is the name assigned to the function. You can make this whatever you want but do not start with a **$** and try to camelCase. In order to execute the code inside the function, this specific name must be “called” or referenced. The parenthesis after the function name must also be placed after the name when calling the function.
* These parenthesis -**( )-** can contain parameters which when used will make functions more dynamic by adding additional inputs to the code. It is not mandatory to add parameters to a function (we’ll explain these later) however adding parenthesis after a function name is necessary.
* The grey section between the curly brackets is the function’s body or code block. Everything in here will be executed when the function is called. It is a collection of statements that perform the function’s function. This part can be as big or small as you like.

Once you have defined/declared a function and added your code inside its body, nothing will happen until you call/invoke the function. The code inside its code block will not run unless you call it.

Now to call/invoke/execute a function one must simply write the function name. Remember to always include the parenthesis even if they are empty and drop the keyword **function** from the start of it! To call the above we would write:

**myFuncName(); //** *When this is read, all code inside function will run*

Because of a phenomenon called **hoisting** (which we’ll explain later on in the course) *certain* JavaScript functions can be invoked/called before it is declared (unlike a variable) without any consequences. The below is perfectly fine as a result.

**myFuncName(); //** *Calling the function*

**function myFuncName ( ) { //** *Declaring the function*

**//** *Code for function goes here*

**}**

**A brief look at functions in other languages**

**-Java- | -C++-**

**public void myFuncName() { | int myFuncName() {**

**//** *Code for function* **|****//** *Code for function*

**} | }**

**|**

**-Python- | -PHP-**

**def myFuncName() : | function myFuncName() {**

**//** *Code for function* **|****//** *Code for function*

**return | }**

In certain languages a function must/should equate to a value. That is to say when the code inside the function is finished, it should return as something like a number, string, array or some other datatype. In the **C++** example above, **int** is used to specify the variable type that the function will return as. In the **Java** example the word **void** is used to let it be known that the function does not equate to value.

**Return**

In JavaScript, if you want a function to return as a value you must use the **return** keyword. When the interpreter reads this **return** keyword, it exits the function straight away and whatever is placed to the right of it, becomes the value of the function. It’s important to note that even if there is code placed on a new line after the **return** statement, this will not be read or executed. Strings, numbers, arrays, objects etc., all other data types can be returned from a function.

**function myFullName() {**

**const firstName = "James";**

**const lastName = "Finn";**

**return firstName + " " + lastName;**

**alert("***This alert will never happen***");**

**}**

Now when we call this function it will return as a string “James Finn”. The alert statement below will show an alert box with “James Finn” in it.

**alert( myFullName() );**

Functions do not have to return a specific value. If you omit the **return** keyword the JavaScript function will return the default **undefined**. However, using return allows you to assign variables using the returned values or use them in other parts of your code like so:

**const fullName =**  **myFullName();** *// fullName is now "James Finn"*

or

**alert( "My name is " + myFullName() );** *// "My name is James Finn"*

**Parameters/Arguments**

You know the way we keep having to place parenthesis **()** after functions even though they don’t seem to do anything. Well they actually do come in handy because of things called **parameters** and **arguments**. Functions can have parameters if you want. These allow you to add new values into a function in order to return new or different outputs.

* These parameters go inside the parenthesis of the function declaration. Here you just give them a name (**para1** and **para2**) which will be referenced in the code block. You can have as many parameters as you want but they must be separated by a comma ( **,** ).

**function myFuncWithPara( para1, para2 ) {**

**alert("My name is " + para1 + " " + para2);**

**}**

* The parameters are assigned values when the function is called by placing the values in the correct position and order within the parenthesis. So for instance we want the first parameter (**para1**) to be a string “James” and the second (**para2**) to be a string “Finn”. They can be any type of data type you want.

**myFuncWithPara( "James", "Finn" );**

* Now since **myFuncWithPara()** is called it will produce an alert stating the following:

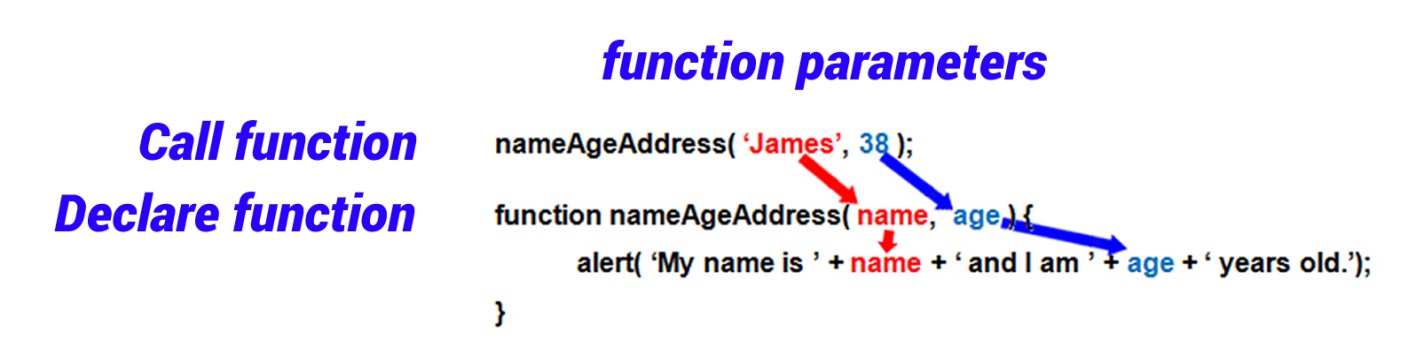
“My name is James Finn”

* But we can reuse this code and add in different parameters to get different results every time we call or invoke the function:

**myFuncWithPara( "Andrew", "Han" );** *// My name is Andrew Han*

**myFuncWithPara( "Michael", "Side" );** *// My name is Michael Side*

**myFuncWithPara( "Takaaki", "Fukuda" );** *// My name is Takaaki Fukuda*



Once again parameters don’t have to be limited to only strings. You can pass any data/variable type as a parameter into a function. You can even pass a function into another function as a parameter – these are called callbacks! **Just make sure to use quotation marks "" for strings** as this is often forgotten by students**.**

**Default Parameter/Argument Values**

You can set up a parameter to have a default value in case the function is called but no value is submitted for that parameter. To do this, in the function declaration use the **assignment operator =** to give the parameter a default **value** it will use in the event it doesn’t receive one.

**function sayName( name1 = "One", name2 = "Two" ) {**

**alert( "Hello " + name1 + " " + name2 );**

**}**

**sayName( "BOB" );** *// alerts "Hello BOB Two"*

In the above example, **name1** is assigned the value “BOB” which appears in the alert. Since only 1 parameter was provided in the function call, **name2** does not get assigned a new value and so it takes its default value of “Two”.

**Different Types of Functions in JavaScript**

There are several different ways of writing or defining functions in JavaScript. There are only slight differences between how these functions act or behave but it’s still relevant to know. One key differentiation is called **hoisting**. Hoisting is where JavaScript brings the declarations of things to the top of the scope in which they are. By doing this in the case of certain functions, it allows a function to be declared at the bottom of the code but be called at the top. Thanks to hoisting this doesn’t produce an error because the interpreter understands that the function exists and can jump to it.

**Function Statement** - These are hoisted

*addition( 1, 2 );* - *This function call works*

function addition(num1, num2) {

return num1 + num2;

}

**Function Expression**

* **Anonymous function expression** - Not hoisted

*addition( 1, 2 );* - *This function call doesn’t works*

var addition = function(num1, num2) {

return num1 + num2;

}

* **Named function expression** - Not hoisted

var addition = function addFunc(num1, num2) {

return num1 + num2;

}

* **IIFE Immediately Invoked Function Expression** - Not hoisted

( function (num1, num2) {- The function is called immediately

return num1 + num2; after it is declared

} ) ( 1, 2 );- Used only once

* ***Arrow functions*** - Not hoisted

var add = (num1, num2) => { - Used due to its concise nature

return num1 + num2; - Used a lot in modern inbuilt JavaScript

}; array methods

var add = (num1, num2) => num1 + num2; - If written in single line, the { }

and return keyword are omitted

**Events**

In JavaScript (and of course other languages) functions can be attached to events on a webpage. These functions will be called when the event occurs. For instance a specific function can be run when a user:

* clicks a button, **onClick**
* types into an input box, **onKeyPress**
* changes the value of a select box, **onChange**
* moves their mouse over an image etc. **onMouseOver**

It’s very simple to add an event to a HTML element. The most basic one is **onClick** which will call any JavaScript code inside its quotation marks **“ ”** to when the element is clicked. Below is an example of a button with code attached to it when it is clicked. Also be careful of mixing up your quotation marks. If you wrap something in **“ ”** then you must make sure that strings inside this have single quote marks **‘ ’**

**<button onClick = "sayHelloWorld( ‘Hello’, ‘World’, 77 )">** Press Me **</button>**

**<script>**

**function sayHelloWorld( para1, para2, para3 ) {**

**alert( para1 + " " + para2 + " " + para3 );**

**}**

**</script>**

Below is an example of code attached to a select box that will run when the value of the select box has been changed.

**<select onChange="showChange()">**

**<option value= “IE”>Ireland</option>**

**<option value= “CA”>Canada</option>**

**<option value= “US”>USA</option>**

**</select>**

**<script>**

**function showChange() {**

**alert( "The value has changed" );**

**}**

**</script>**