POWERING POTENTIAL

Functions

→ JavaScript Fundamentals

Functions

- What are functions?
- Creating functions
- Calling functions

Scope

- What is scope?
- Functions and scope

Functions – about

- Functions are one of the most important concepts in JavaScript
- Functions allow us to block out code for execution when we want, instead of it running as soon as the browser processes it
- Allow us to reuse the same operations repeatedly, like console.log();
- Functions are first-class objects and are actually a type of built-in type
- The keyword function actually creates a new object of type Function

Functions – creating

- The function keyword is used to create JavaScript functions

Function is a language keyword

```
function sayHello( ) {
    alert("Hi there!");
}
```

Name of the function

- Parameters may be passed into a function

```
function sayHelloToSomeone(name) {
    alert(`Hi there ${name}!`);
}
```

- It may optionally return a value

```
function returnAGreetingToSomeone(name) {
    return `Hi there ${name}!`
}
```

Functions – calling

- Functions, once created, can be called
- Use the function name
- Pass in any parameters, ensuring the order
- If the function returns, pass back result

```
sayHelloToSomeone("Dave");
let r = returnAGreetingToSomeone("Adrian");
```

- Parameters are passed in as value based
- The parameter copies the value of the variable
- For a primitive, this is the value itself
- For an object, this is a memory address

Arrow Functions

- Can be declared as **`const`** (or **`let`**) setting a variable name to be a function
- Syntax:

```
scope name = () => implicit return;                    // no arguments
scope name = arg => implicit return;                   // single argument
scope name = (arg1, arg2) => implicit return;          // multiple arguments
scope name = ( …args) => ( implicit return );          // rest arguments and bracketed return
scope name = () => {                                   // code block with optional defined
  // function block                                    // return
  // return if required
};
```

Arrow Functions - examples

```
const noArgFnImpRet = () => `Hello World`;          // returns `Hello World` when called

const noArgFnCodeBlk = () => { // Some implementation code with return if required };

const sglArgFn = arg => { console.log(arg); }       // outputs value of arg

const multiArgFn = (num1, num2) => ( num1 * num2 ); // outputs value when called
```

- Arrow functions can be used in place of anonymous functions too
- Useful where a callback function has to be supplied
- Need to be careful that the value of **this** is understood
- This will become clearer when defining event handling functions for the DOM

Default values & rest parameters

- Default values were a long-standing problem with a fiddly solution
- Can provide a value for the argument and if none is passed to the function, it will use the default

```
function doSomething(arg1, arg2, arg3=5) {
    return(arg1 + arg2 + arg3);
}
console.log(doSomething(5,5)); //15
```

- If the last named argument of a function is prefixed with… then its value and all further values passed to the function will be captured as an array:

```
function multiply(arg1, ...args) {
args.forEach((arg,i,array) => array[i] = arg*arg1);
return args;
}
console.log(multiply(5,2,5,10)); //[10,25,50], 5 = arg1, [2, 5, 10] = …args
```

Functions – scope (1)

- Scope defines where variables can be seen
- Use the let keyword to specify scope to the current block
- If you don't use let, then variable has 'global' scope

```
function test()
{
    flag = true;

    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    flag = false;
    return
}
```

true

false

```
function test()
{
    flag = true;

    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    let flag = false;
    return
}
```

true

true

Functions – scope (2)

- In the code sample to the left, the flag variable is explicitly defined at global level
- In the code sample to the right it is declared in the scope of test
- Can test1 see it?

```
let flag = true;

function test()
{
    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    flag = false;
    return
}
```

true

false

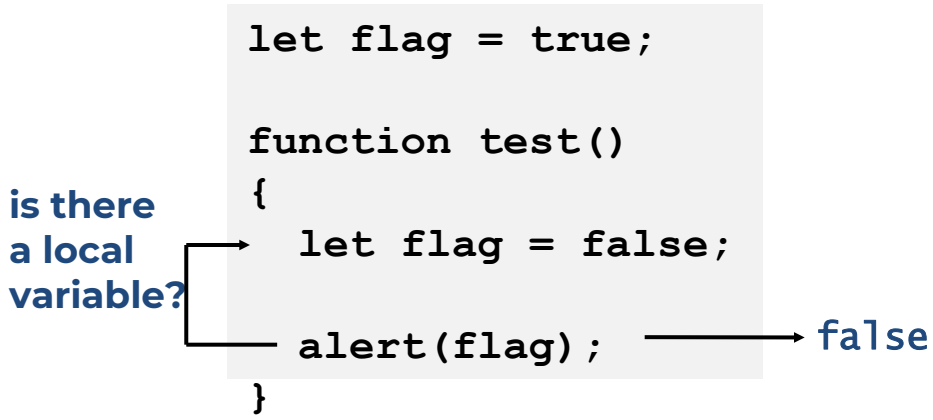
```
function test()
{
    let flag = true;

    alert(flag);
    test1();
    alert(flag);
}

function test1()
{
    flag = false;
    return
}
```

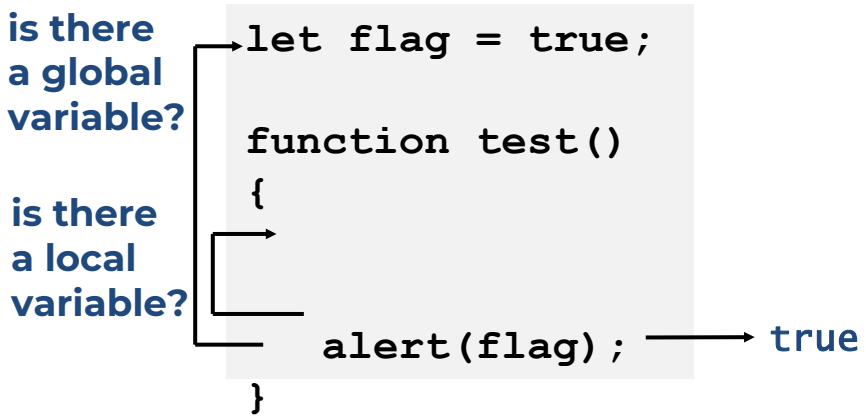
true

true

Functions – local vs. global scope

- Scope Chains define how an identifier is looked up
- Start from inside and work out





- What happens if there is not a local or global variable?
- One is added to global scope!

The global object

- Global object for client-side JavaScript is called window
- Global variables created using the **var** keyword are added as properties on the global object (window)
- Global functions are methods of the current window
- Current window reference is implicit

```
var a = 7;
alert(b);
```

```
window.a = 7;
window.alert(b);
```


These are equivalent

- Global variables created using the let keyword are NOT added as properties on the window

The global object

- Unless you create a variable within a function or block it is of global scope
- The scope chain in JavaScript is interesting
- JavaScript looks up the object hierarchy not the call stack
- This is not the case in many other languages
- If a variable is not seen in scope, it can be accidently added to global
- Like the example in the previous slide

QuickLab 17 - Functions

- Create and use functions
- Returning data from a function

REVIEW

- Functions allow us to create reusable blocks of code
- Scope is a critical concept to understand and utilise in your JavaScript programming career
- Functions are first-class objects, meaning we can pass them round as we would other objects and primitives