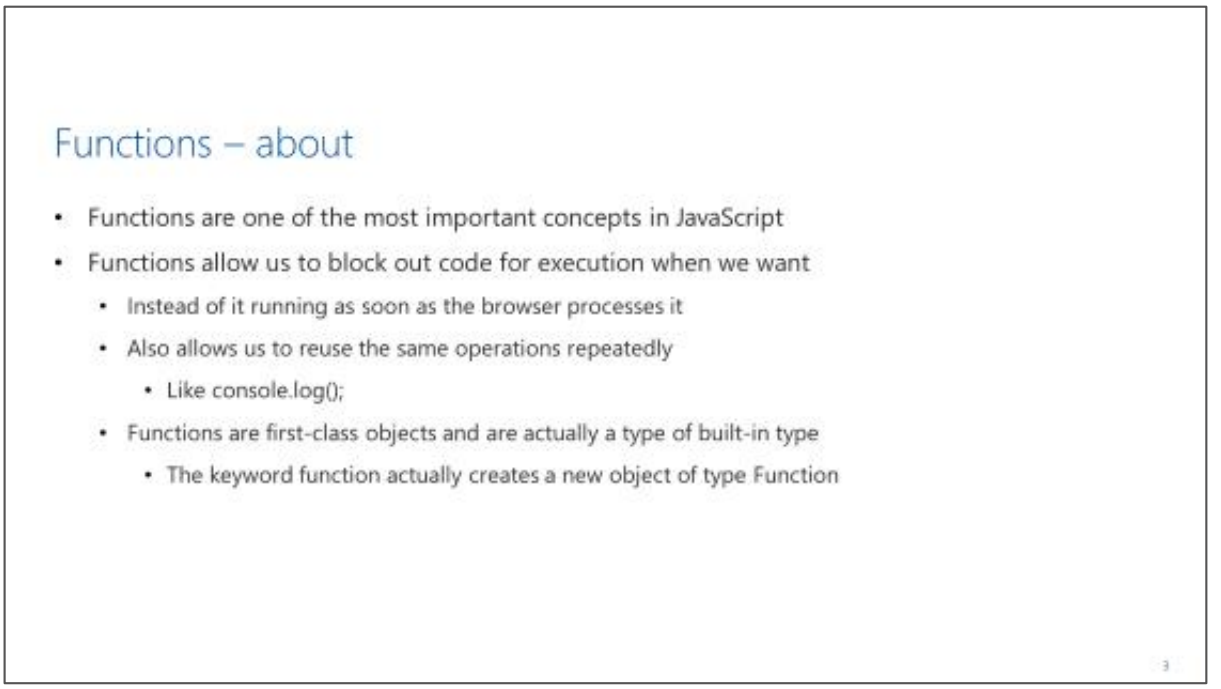
Functions

PROGRAMMING WITH JAVASCRIPT

Introduction

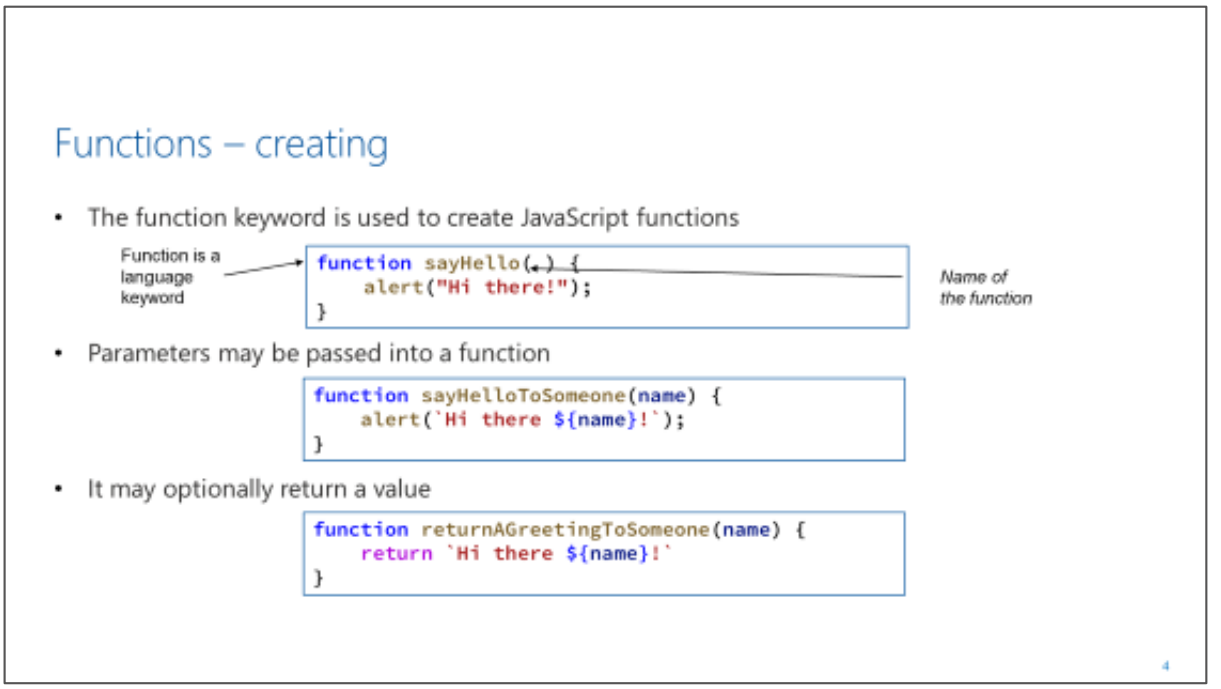
- Functions
  - What are functions?
  - Creating functions
  - Calling functions
- Scope
  - What is scope?
  - Functions and scope

2



In JavaScript a function is a type. It is a first-class object. Whenever we declare a function, we are actually declaring a variable of type function with the name of the function becoming the name of the variable.

When we invoke a function, we are actually calling a method of that function (a function on the function) called call – the invocation syntax is simply sugar.



Functions are created with the reserved word function. The newly created function block has mandatory curled braces and a function name. In the first example above, a simple function has been created that calls an alert and takes no input.

In example two, we are passing in a value this is known as a parameter or argument. It is at this point an optional parameter. If it was passed in blank, it would be undefined as a type. In the exercise, we will consider the idea of passing in the parameter and checking its value.

The first two functions just go off and do 'something'; they would often be known as 'sub routines' and are effectively a diversion from the main program allowing us to reuse code.

The third example uses the return keyword. It calculates or achieves some form of result and then returns this result to the program. It would normally be used in conjunction with a variable as we will see on the next slide.

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

5

The function must be declared before it is called. It must be declared in the current block, a previously rendered block or an external script file you have referenced, or else the function call will raise an error.

Parameters need to be passed in the correct order and remember that JavaScript has no type checking externally. We would need to be sure what is going into the function call is not garbage. Any parameters passed in are local copies and do not refer to the original variable.

The `return` statement has two purposes in a function. The first is as a method of flow control: when a return statement is encountered, the function exits immediately, without executing any code which follows the return statement. The second, and more important use of the return statement, returns a value to the caller by giving the function a value.

If a function contains no return statement, or the return statement does not specify a value, then its return value is the special undefined value, and attempting to use it in an arithmetic expression will cause an error.

Default Values & Rest Parameters

- Default values were a long standing problem with a fiddly solution
- Now we 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 it's 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]
```

6

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()
{
    var flag = false;
    return
}
```

true

true

7

The **var** keyword has additional semantics: it provides 'scoping' of variables. Variables declared outside the body of a function are always global as expected. Less obvious is that variables created inside a function body are also global. However, variables created explicitly using by the **var** keyword have scope limited to the enclosing function (no matter how deep within the function they are created).

Variables created outside of a function body using the **var** keyword are global. Note that means global to the html page, not just the current SCRIPT block.

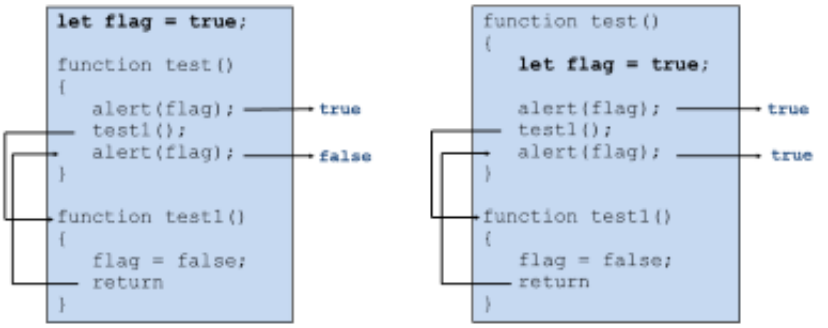
In the first example in the slide, setting the value of flag to false in function test1() has the effect of changing the value of the global variable and hence flag in function test().

In the second example, marking the variable flag as **var** in function test1() restricts its scope to that function only. This means that setting the value of the local flag variable does not effect the value of flag set in function test().

Improper use of scoping can be a source of major debugging headaches.

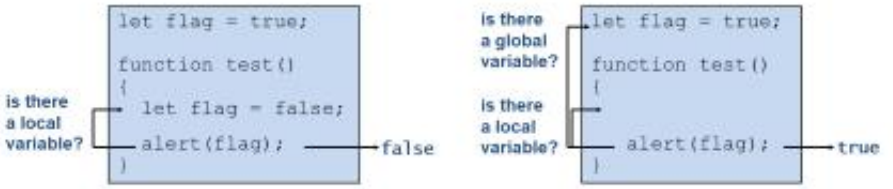
Functions – scope (2)

- 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?



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 scoping rules and the way identifiers are resolved follow an inside to outside flow in JavaScript.

If this code is executing in a function, then the first object to be checked is the function itself: local variables and parameters. After that, the 'global' store of variables is checked.

We discuss functions in greater detail later, but while we are talking about scope: variables declared in a formal parameter list are implicitly local.

```
function test(flag)     // This flag has local scope
{
  flag = false
}
```

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

10

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

11

Review

- Functions allow us to create re-usable 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

12

Exercise

- Create and use functions
- Returning data from a function
- Use functions as arguments to other functions

13