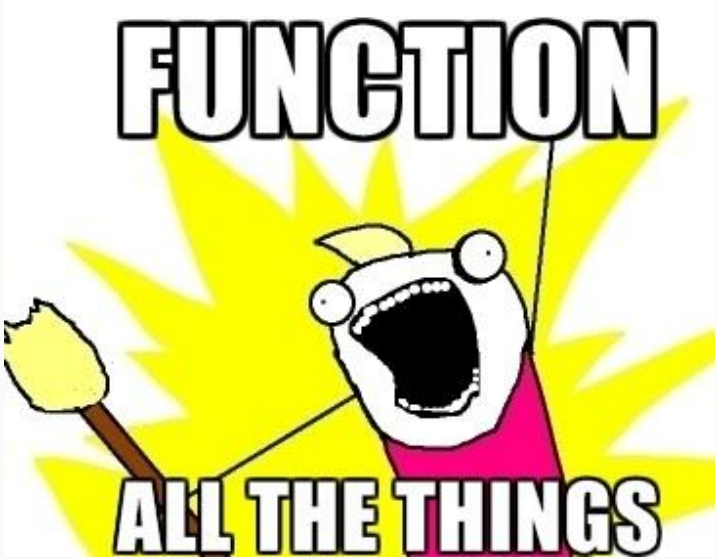
Școala
informală
de IT

Functions and Scopes

Agenda

- What a function is and why it’s useful
- Defining and using functions
- Function arguments & return values
- Scope
- Closure



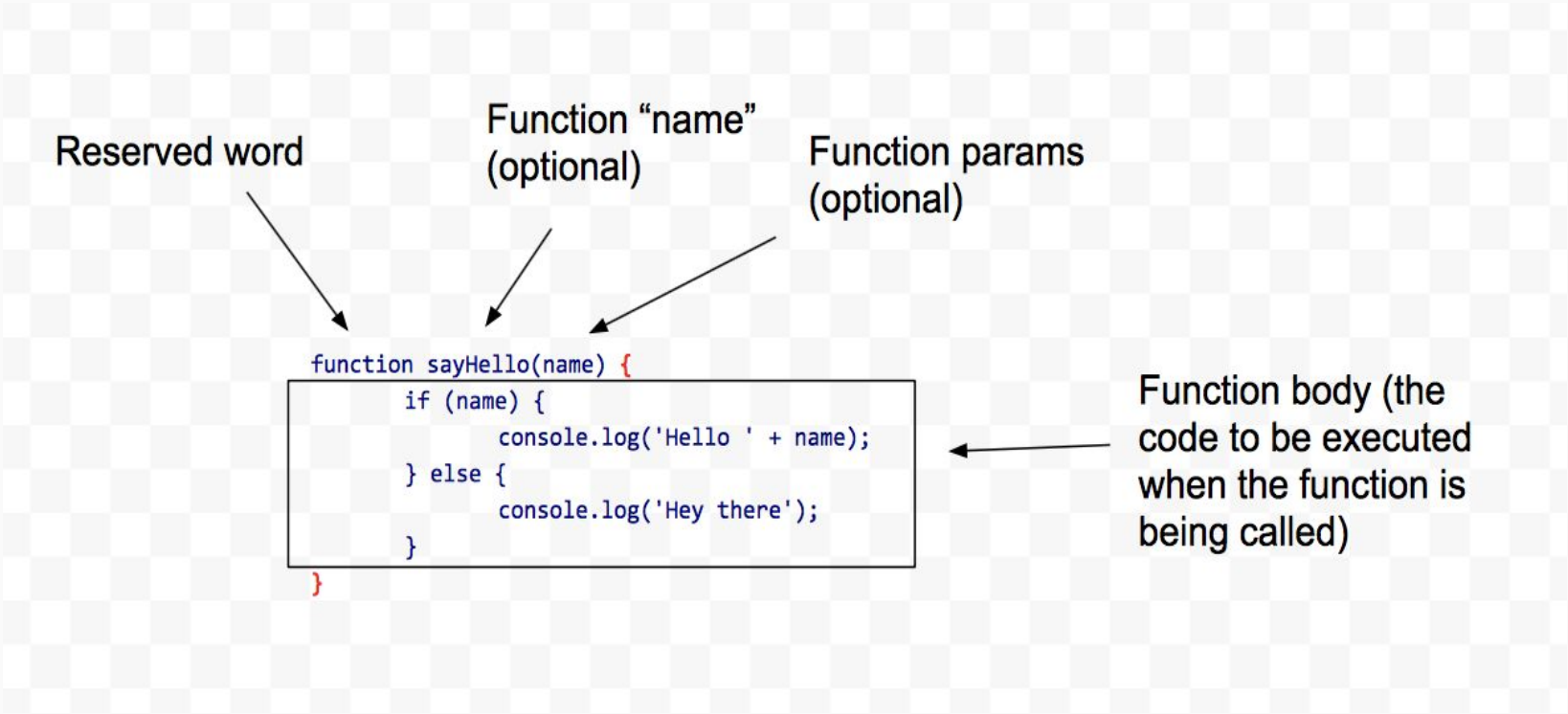
What a function is &
Why it’s useful

Functions

- = pieces of programs made of of statements that perform a task or calculate a value
- Why they are useful:
  - **reuse code / reduce repetition**
    - You can use the same code many times with different arguments, to produce different results.
  - **structure larger programs**
  - **associate names with subprograms**
  - **isolate these subprograms from each other**

Defining & Using Functions

Function declaration



Function expression

- The function keyword can be used to define a function inside an expression.
- The name of the function can be omitted.
  - Such a function is called anonymous.
- Note: just by defining functions the code inside them will not be executed until the function is called by its name!

```
// function expression
var sayHello = function sayHello(name) {
  if (name) {
    console.log('Hello ' + name);
  }
  else {
    console.log('Hey there');
  }
}

// function expression
// with anonymous function
var sayHelloAnonymous = function(name) {
  … // code here
}

// calling a function
sayHello(); // Hey there
```

Function invocation

- How can a function be invoked?
  - It is called using () from the Javascript code
  - It is called when an event occurs
  - It is called by itself; these functions are called: self invoked functions / immediately invoked function expressions (IIFE)
- *Recursivity

```
// called from the Javascript code
sayHello();

// when an event occurs
window.onload = sayHello;

// Immediately Invoked Function Expression
(function () {
  console.log("This is self invoked");
})();
```

Function Arguments / Parameters & Return Values

Function arguments & parameters

- **Function parameters are the names listed in the function definition.**
- **Function arguments are the real values passed to (and received by) the function.**
- **Arguments can be used in the function body**
  - Providing arguments directly when calling the function
  - Declaring the arguments in a variable before calling the function

```
// firstName and lastName are parameters
function fullName(firstName, lastName) {
  console.log( firstName + ' ' + lastName);
}

// 'Bruce' and 'Wayne' are arguments;
// they are passed directly when the
// function is called
fullName('Bruce', 'Wayne');

// arguments are previously-defined vars
var fName = 'Bruce';
var lName = 'Wayne';

fullName(fName, lName);
```

Sending parameters by value

- **Primitive parameters** (such as a number) are passed to functions **by value**
  - the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function
  - it's like creating a copy of that value inside the function

```
function sum(a, b) {
  console.log('sum', a+b);
  a = 0;
  console.log(a); // 0
}

var a = 2, b = 3;

sum(a, b); // 5, 0

console.log(a); // 2
```

Sending parameters by reference

- **Non-primitive parameters are passed to functions by reference**
  - **If you pass an object (such as an array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function**

```
function fullName(name) {
 console.log(name.first +' '+ name.last);
 name.last = 'Batman';
 console.log(name.last); // 'Batman'
}

var name = {
  first: 'Bruce',
  last: 'Wayne'
}

fullName(name); // 'Bruce Wayne', 'Batman'

console.log(name.last); // 'Batman'
```

Returning values from functions

- To return values from function we can use the reserved keyword return.
- If we don't specifically return nothing from a function *undefined* will be returned *automatically*.
- Note: After the return statement nothing will be executed.

```
function sum(a, b) {
  return a + b;
  // nothing will execute here
}

// combining function calls with expressions
var age = sum(10, 10) + sum(2, 3); // 25

// function calls inception!
var age = sum(sum(10, 10), sum(2, 3)) + 2; // 27

// recursive function
function factorial(n) {
  if ((n === 0) || (n === 1)) { return 1; }
  else { return (n * factorial(n - 1)); }
}
```

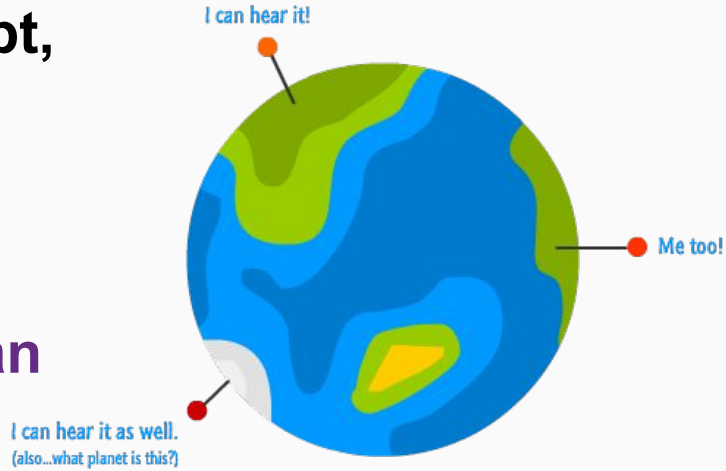
Function Scopes

Function Scopes

- In JavaScript, the set of identifiers that each environment has access to is called scope.
- In other words, scope determines where variables/functions are accessible.
- Scopes can be
  - global or
  - local.

Global Scope

- Before you write a line of JavaScript, you're in what we call the Global Scope. If we declare a variable, it's defined globally
- In other words, a global variable can be accessed from anywhere




```
/* global variables, accessible from everywhere */
var x = 1;
window.x = 1;
x = 1;
```

Local Scope

- There is typically one global scope, and each function defined has its own (nested) local scope.
- Any function defined within another function has a local scope which is linked to the outer function.

```
function myFunction() {
  var name = 'Todd';
  console.log(name); // Todd
};

/* Uncaught ReferenceError: name
is not defined */
console.log(name);
```

Function Scopes Rules

- Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.
- A function can access all variables and functions defined inside the scope in which it is defined
- A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access.

```
var x = 2;

function sum(a, b) {
  // can access a and be here
  console.log('sum', a+b);

  // can also access x here
  console.log(x);
}

// can't access a and be here
```

Closures

Scope Chain

- A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access. This is called scope chain
- The inner function has 3 scope chains:
  - it has access to its own scope (variables defined between its curly brackets),
  - it has access to the outer function's variables, and
  - it has access to the global variables.

```
var global = 'Global!';
function greetPerson(name) {
  function greet(greeting) {
    console.log(greeting + name);
    console.log(global);
  }
  greet('Hello');
};
```

Closures

Let's change the example a bit...

- **The scope chain is saved to the function object at the time of its creation.**
- **In other words, when an inner function is returned from an outer function, all the variables of the outer function are saved in the context of the inner function**

```
function greetPerson(name) {
  function greet(greeting) {
    console.log(greeting + ' ' + name);
  }
  return greet; // function!
};

var greetAna = greetPerson('Ana');
greetAna('Hello'); // Hello Ana
greetAna('Bye'); // Bye Ana

var greetBen = greetPerson('Ben');
greetBen('Hello'); // Hello Ben
```

Closures

- **Closures are functions together with an execution context*.**
  - ***all the variables in the scope chain**
- **A closure is created every time an enclosing outer function is called.**
- **Let's see some examples:**
  https://medium.freecodecamp.org/lets-learn-javascript-closures-66feb44f6a44#8178

```
function greetPerson(name) {
  function greet(greeting) {
    console.log(greeting + ' ' + name);
  }
  return greet; // function!
};

// closure #1
var greetAna = greetPerson('Ana');
greetAna('Hello'); // Hello Ana
greetAna('Bye'); // Bye Ana

// closure #2
var greetBen = greetPerson('Ben');
greetBen('Hello'); // Hello Ben
```

Resources

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

https://medium.freecodecamp.org/lets-learn-javascript-closures-66feb44f6a44

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures