FUNCTIONS

Lesson Objectives

- Learn how to write functions in JavaScript
- Understand function call and return in relation to stack frames
- Understand scope and scope chain

Function declaration (function statement)

- The first line of function is called the signature, and it includes the keyword `function`, the function name and the optional parameter list.

```
function sum(num1, num2){
  return num1+num2;
}
```

```
function greet(){
  console.log("Hi, from a function");
}
```

- The statements inside a function are called the body of a function.
  - Function returns `undefined`, when return is not explicit.

Calling a function

- To call a function, write a function name followed by a set of parentheses;
- optionally passing matching arguments for the corresponding parameters.

```
let total = sum(5,5); // call to function sum

greet(); // call to function greet
```

Parameters vs Arguments

- Function parameters are the names of variables in the function definition.
- Function arguments are the actual values passed to the function

```
// function sum has two parameters num1 and num2
function sum(num1, num2){
  return num1+num2;
}

let total = sum(5,10); // arguments 5 and 10 for num1 and num2 respectively
```

Default values

- If an argument is not provided for a parameter, then its value becomes undefined.
- If we want to use a “default” value instead, then we can specify it after `=`

```
function sum(num1=0, num2=0){
  return num1+num2;
}
```

- What would be the result of calling `sum()` if default parameters were not assigned?
  - Is it even a valid call?

Returning a value

- A function can return a value to the calling code
- The directive `return` can be any place of the function.
  - When the execution reaches it, the function stops, and the value is returned to the calling code.
  - There may be many occurrences of `return` in a single function.
  - It is also possible to use `return` without a value. That causes the function to exit immediately.
- A function with an empty `return` or without it returns `undefined`

```
function oddEven(num){
  if (!num) return;
  if(num%2==0) return "Even";
  else return "Odd"
}
```

Beware semicolon insertion

- For a long expression in return , it might be tempting to put it on a separate line

```
return
(some + long + expression + or + whatever * f(a) + f(b))
```

- JavaScript assumes a semicolon after return . That'll work the same as:

```
r e t u r n ;
(some + long + expression + or + whatever * f(a) + f(b))
```

- becomes an empty return.

- For debugging purposes it is best to put expressions before the return
  - Best for debugging if return a single value or variable that holds the result of a computation in the function

Function names

- Functions are actions. So their name is usually a verb.

```
showMessage(..) // shows a message
getAge(..) // returns the age (gets it somehow)
calcSum(..) // calculates a sum and returns the result
createForm(..) // creates a form (and usually returns it)
checkPermission(..) // checks a permission, returns true/false
```

- A function should do exactly what is suggested by its name, no more.
  - Two independent actions deserve two functions,
  - if usually called together make a 3rd function that calls those two
  - getAge –bad if shows an alert with the age (should only get).
  - createForm –bad if modifies the document, adding a form to it (should only create and return).
  - checkPermission –bad if displays access granted/denied message (should only perform check and return result).

Exercises

- Write a function named `testPrime` that returns true when the argument to the function is a prime number, otherwise returns false.
  - (Best practice to first write the steps in English)
    - E.g., “defining table”
  - Now call the function to test if user input is prime or not.

Main point

- Expert developers break complex problems into smaller, meaningful, reusable functions. Functions make programs modular, reusable and easier to understand. *Science of consciousness, with regular experience of pure consciousness, one develops the ability to have fine focus on details without losing the big picture*.

Local variables

- A variable declared inside a function is only visible inside that function.

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable
  console.log( message );
}
showMessage(); // Hello, I'm JavaScript!
console.log( message ); // <--
 Error! The variable is local to the function
```

Outer variables

- A function can access an outer variable as well, for example:

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

- function has full access to the outer variable. It can modify it as well.
- Avoid if possible
  - Breaks encapsulation
  - Sometimes necessary (closures)

Variable Shadowing

- If a same-named variable is declared inside the function, then it *shadows* the outer one.
  - For instance, in the code below the function uses the local `userName`. The outer one is ignored:
  - Shadowing is generally a bad practice since it can confuse humans

```
let userName = 'John’;

function showMessage() {
  let userName = "Bob"; // declare a local variable
  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

showMessage();

alert( userName ); // John, unchanged
```

Scope revisited

- The scope of a variable determines how long and where a variable can be used.
- With let and const JavaScript has block scope
  - Parameters are local to a function.
- let and const → block scope.

Lexical scope in JavaScript (ES6+)

- From ES6, in JavaScript every block ({}) defines a scope
  - Via let and const

Global Scope

```
let x = 10;

function main() {
 let x;
 console.log("x1: " + x);
 if (x > 0) {
   let x = 30;
   console.log("x2: " + x);
 }

 x= 40;
 let f = function(x) { console.log("x3: " + x); }
 f(50);
}

main();
```

Block Scope

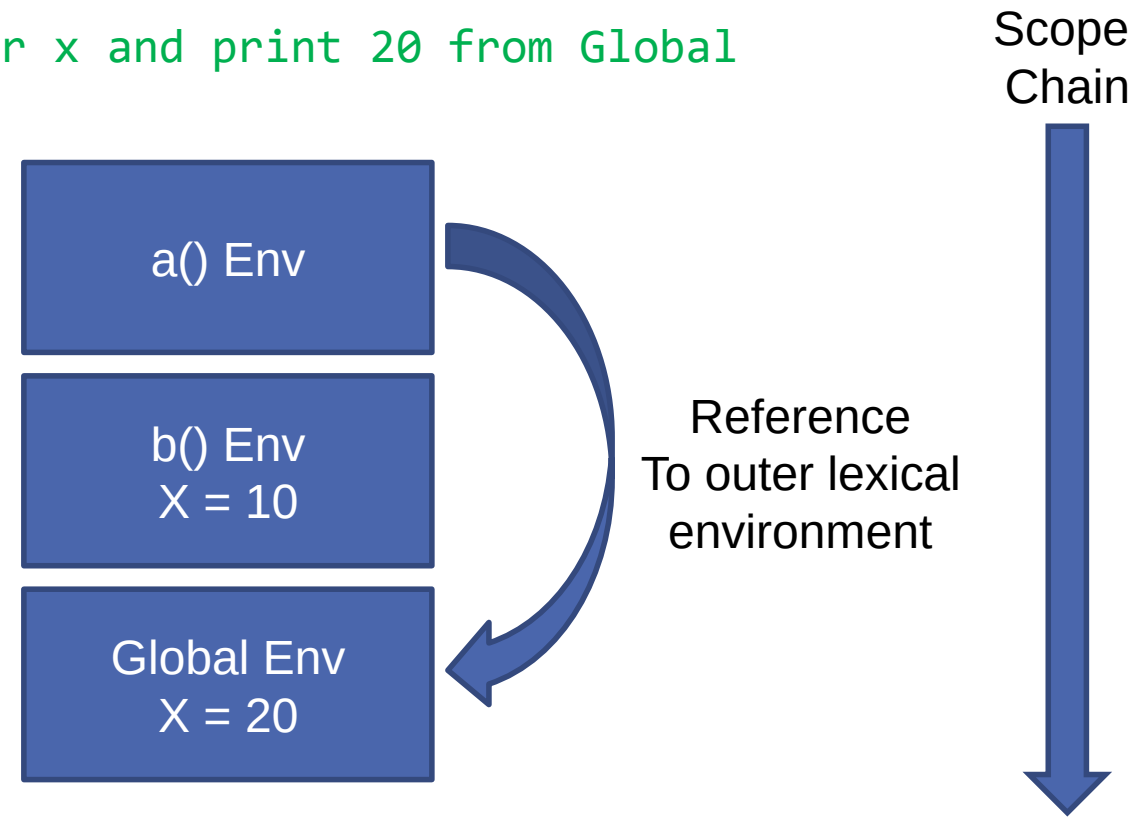
Block Scope

Scope chain

- When we refer to a variable in a program, JS engine will look for that variable in the current scope. If it doesn’t find it, it will consult its outer scope until it reach the global scope.

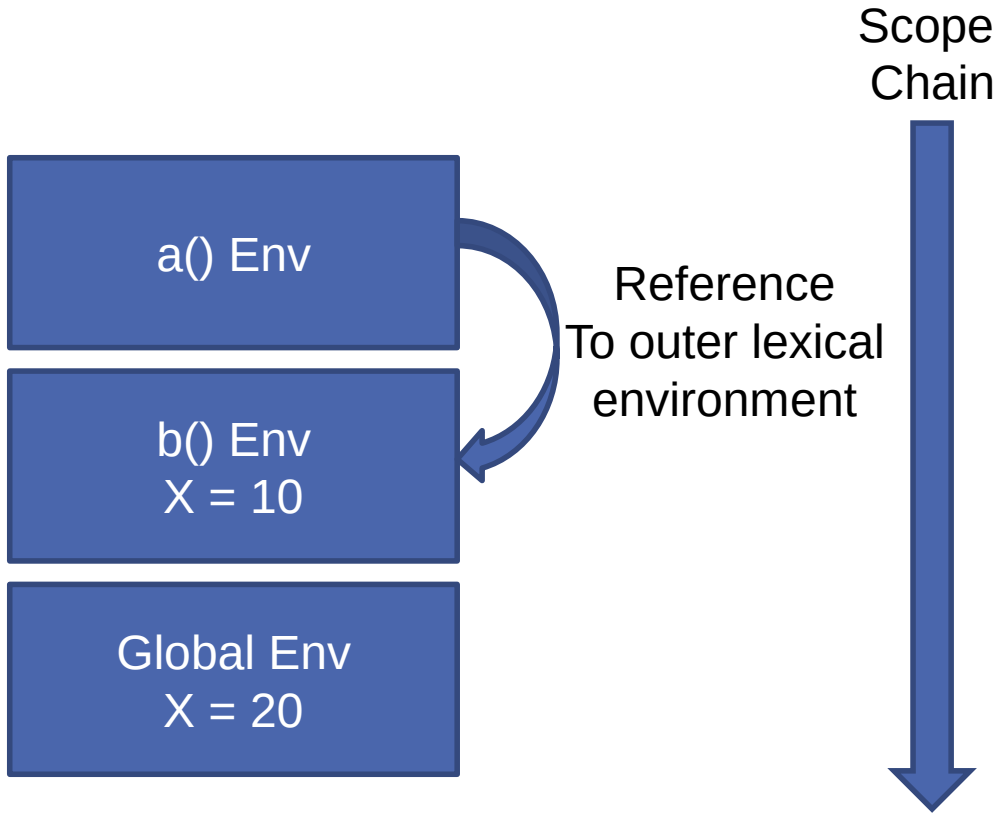
Scope Example

```
function a(){
        console.log(x); // consult Global for x and print 20 from Global
}

function b(){
        let x = 10;
        a(); // consult Global for a
        console.log(x);
}

let x = 20;
b();
```



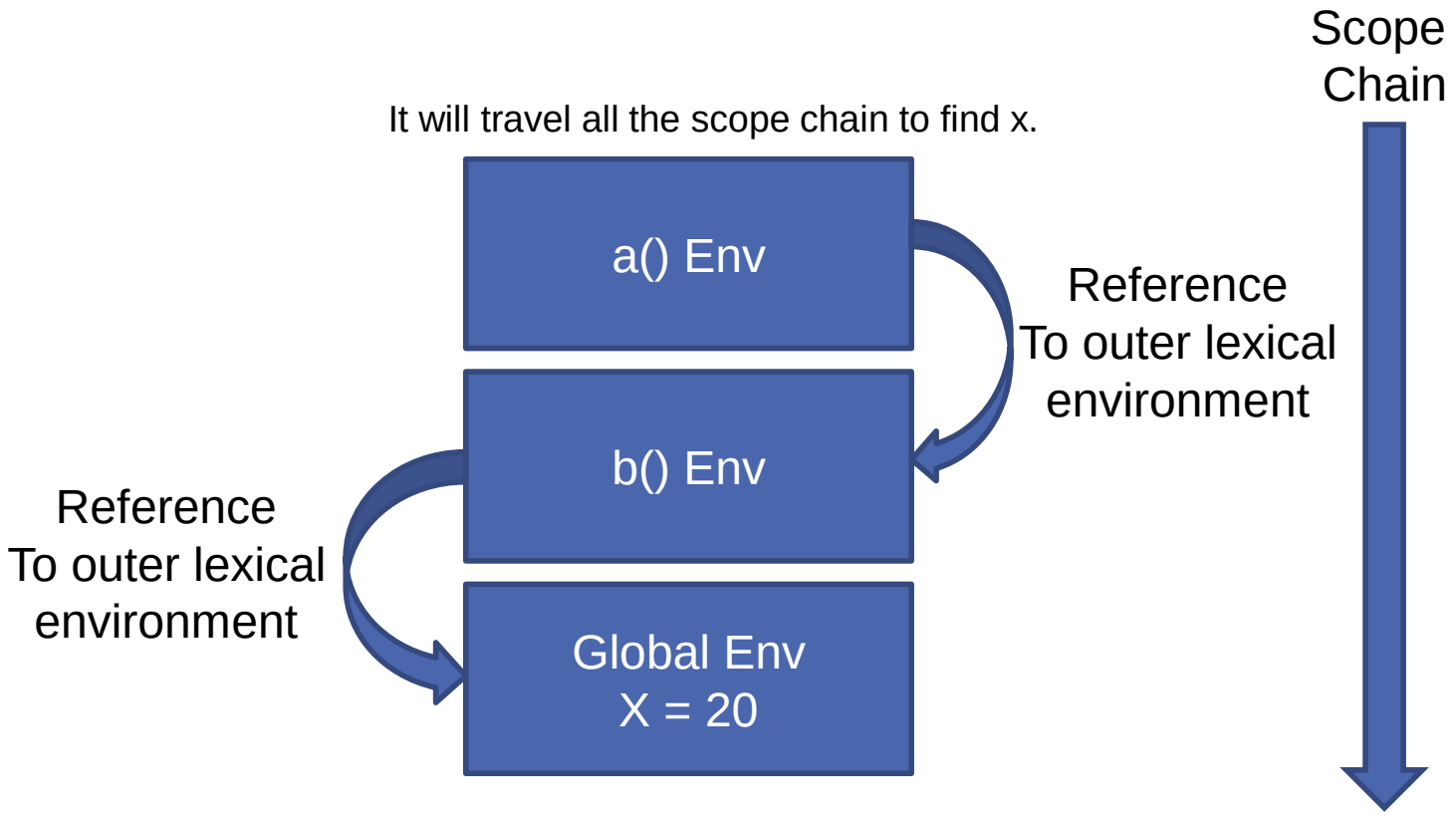
Scope Example

```
function b(){
        function a(){
                console.log(x);
        }
        let x = 10;
        a();
        console.log(x);
}

let x = 20;
b(); // 10
```



Scope Example

```
function b(){
        function a(){
                console.log(x);
        }
        a();
        console.log(x);
}

let x = 20;
b(); // 20
```



Scope Example

```
function f() {
        let a = 1, b = 20, c;
        console.log(a + " " + b + " " + c); // 1 20 undefined

        function g() {
           let b = 300, c = 4000;
           console.log(a + " " + b + " " + c); // 1 300 4000
           a = a + b + c;
           console.log(a + " " + b + " " + c); // 4301 300 4000
        }

        console.log(a + " " + b + " " + c); // 1 20 undefined
        g();
        console.log(a + " " + b + " " + c); // 4301 20 undefined
}
f();
```

Exercise

```
let x = 10;
function main() {
    let x = 0;
    console.log("x1 is " + x);
    x = 20;
    console.log("x2 is " + x);

    if (x > 0) {
        x = 30;
        console.log("x3 is " + x);
    }
    console.log("x4 is " + x);

    function f(x) {
        console.log("x5 is " + x);
    }
    f(50);
    console.log("x6 is " + x);
}
main();
console.log("x7 is " + x);
//Draw the scope chain
```

Main Point

Scope chain and execution context

When we refer a variable in a program, JavaScript will look for that variable in the current scope. If it doesn't find it, it will consult its outer scopes until it reach the global scope. *Science of consciousness, During the process of transcending we naturally proceed from local awareness to more subtle levels of awareness to unbounded awareness*.

Exercise

- Write a function to compute area of a triangle based on the following formula
  - computeArea = $\sqrt{s(s-a)(s-b)(s-c)}$
    - where a, b and c are the lengths of the three side of a triangle and s is the semi-perimeter of the triangle defined by following formula
    - s = (a+b+c)/2;
    - write a separate function for computing semi-perimeter.
    - parameters for computeArea will be the lengths of the triangle sides: a, b, c
    - Start with a “defining table”

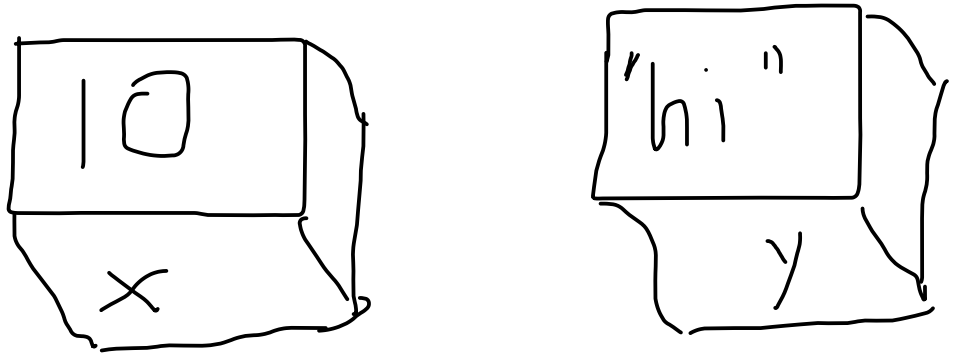
DATA TYPES, SELECTION AND ITERATION (PART II)

Variables

- Memory location referenced by some identifier like x and y.
  - Machine saves data on those memory locations.
  - Machine access/ manipulates variables in a program in order to compute results.

    const x = 10;

    const y = “hi”;



Variable naming

- There are two limitations on variable names in JavaScript
  - The name must contain only letters, digits, or the symbols $ and _
  - The first character must not be a digit.

```
// valid
let username, test123;

// invalid
let 2ndName, my-name;
```

- There is a list of reserved words, which cannot be used as variable names because they are used by the language itself.

- *Case sensitive*
- *Always write meaningful variable names.*

Modern JavaScript “use strict”;

- JavaScript has added many new features over the years
- Legacy code must also still run
  - Includes some poor language features
- Modern JavaScript removes many of the bad parts
  - Must run in strict mode to take advantage

```
//"use strict";
const msg = 'hello';
console.log(msg);
mgs = 'goodbye';
console.log (msg);
```

Good coding practice

- Use `const` if the value won't change after assignment
- Use `let` for variables that need to be reassigned
- favor const

- **Never use `var`**

The `typeof` operator

- The `typeof` operator returns the type of the argument.

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof 10n // "bigint"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
```

Conditional (ternary) operator

- The conditional (ternary) operator is the only JavaScript operator that takes three operands:
  - condition followed by a question mark (`?`),
  - expression to execute if the condition is truthy followed by a colon (`:`),
  - expression to execute if the condition is falsy.

- *frequently used as a shortcut for the if statement*.

  - value of the evaluated expression is returned
  - const message = day > 5 ? "Happy weekend! " : "Happy weekday!";

- Only use this for very simple conditions
  - Probably best to avoid until become experienced and comfortable with standard if else

nullish coalescing operator ??

- ?? provides a short way to choose the first “defined” value from a list.

- It’s used to assign default values to variables:

```
// set height=100, if height is null or undefined
height = height ?? 100;
```

Switch

- A switch statement can replace multiple if (or else if) checks
- The `switch` has one or more `case` blocks and an optional `default`.

- Using else if – See example: demos\d11iteration\selection\using_else_if.js
- Using switch – See example: *demos\d11iteration\selection\using_switch.js*

- value is checked for strict equality to value of first case then second …
  - if equal execute code from corresponding case, until nearest break (or until the end of switch).
  - If no case matched then default code is executed (if it exists).

Exercise

- Write a program that asks user to enter number between 1 to 5 and prints out how the number is spelled.
  - First, write using else if
  - Then, refactor it to use switch

Breaking the loop

- Normally, a loop exits when its condition becomes falsy.
- But we can force the exit at any time using the special break directive.

```
let sum = 0;

while (true) {
  let value = +prompt("Enter a number", '');
  if (!value) break; // (*)
  sum += value;
}
alert('Sum: ' + sum);
```

- The break directive is activated at the line (*) if the user enters an empty line or cancels the input. It stops the loop immediately

Continue to the next iteration

- The `continue` directive is a “lighter version” of `break`.
  - It doesn’t stop the whole loop.
  - Instead, it stops the current iteration and forces the loop to start a new one (if the condition allows).

```
for (let i = 0; i < 10; i++) {
  // if true, skip the remaining part of the body
  if (i % 2 == 0) continue;
  alert(i); // 1, then 3, 5, 7, 9
}
```

  - Refactor above code without using the continue statement.

break and continue

- The break statement ends the entire loop statement prematurely.
  - Example - finding if a number is prime without using break statement.
    - See *demos\d11iteration\loops\test_prime_no_break.js*
    - Example - finding if a number is prime using break statement.
    - See *demos\d11iteration\loops\test_prime_using_break.js*

- The continue statement "jumps over" one iteration in the loop.
  - See example: demos\d11iteration\loops\continue_keyword.js

- break and continue are rarely required in well designed code
  - best practice to avoid unless have a specific reason to use
  - typical use case involves performance optimization of stopping a loop upon some condition
  - can generally achieve same effect with a sentinel while loop
    - Exercise: change the following to be a while loop with sentinel and no break
    - demos\d11iteration\loops\test_prime_using_break.js

Defining Table

An **algorithm** is simply a list of steps to perform some task. A large computer program contains many algorithms. Before creating an algorithm to solve a problem, you must be sure that you understand the problem. If you don't, you will probably create an algorithm that solves the wrong problem. A **defining table** is a useful tool to help you better understand a problem before you develop an algorithm to solve it. A defining table has three sections: input, processing, and output. To create a defining table, simply draw a table with the three sections. Then as you read and re-read the problem, put the parts of the problem into their

correct section in the table.

Example 1

You work for a large construction company. Your boss has asked you to write a computer program that will read a list of window openings for a building and compute, and output the total cost of all the windows. The window openings are entered in inches with the width first and the height second. The cost of a window is computed by multiplying the area of the window in square feet by $35.

Defining Table

Input	Processing	Output
A list of window openings For each window • width in inches • height in inches	For each window • compute area in sq. ft. • multiply area by $35 • add cost of this window to the total cost	total cost of all windows

Exercise: setup GitHub repository

Create a GitHub account and then setup your course repository according to the instructions in Sakai > Resources > lab helpers > setupGithubRepository.pdf