Functions

Lesson Objectives

- Understand idea and uses of functions
- Learn how to use built-in functions (methods)
- Learn how to write functions in JavaScript
- Understand function call and return in relation to stack frames
- Understand scope and scope chain

Function

- Modular organization of related code, to perform a specific task.
- A function can be a program by itself, but usually a program is composed of number of functions. i.e., in most cases, a function is a "subprogram“

- We’ve already seen examples of built-in functions, like `alert(message)`, `prompt(message, default)`.But we can create functions of our own as well.

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.
- See examples:
  - lecture_codes/lesson5/func_say_hi.js
  - Lecutre_codes/lesson5/func_test_odd.js

Calling a function

- A function by itself won't do anything, unless you call/invoke it.
- How a function is called depends on functions header/signature
  - To call a function, you simply 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 present in the function definition.
- Function arguments are the real values that are passed to the function and received by them.

```
// 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 a parameter is not provided, 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 back into the calling code as the result.
- The directive `return` can be in 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.

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 argument to the function is a prime number, otherwise returns false.
  - Now call the function to test if user input is prime or not.

Main point

- Functions are subprograms and a computer program usually is composed of number of smaller functions. Functions makes programming modular, reusable and easier to understand. When a program starts to get complex, we must break it into smaller functions in order to handle it better. To be a better programmer we should not only be able to solve a problem at hand, but also need to be able to break it into smaller, meaningful, reusable functions. *Science of consciousness, With the regular experience of pure consciousness through practice of TM, one develop ability to have fine focus on small 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
  alert( message );
}
showMessage(); // Hello, I'm JavaScript!
alert( 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, to be covered in 303)

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

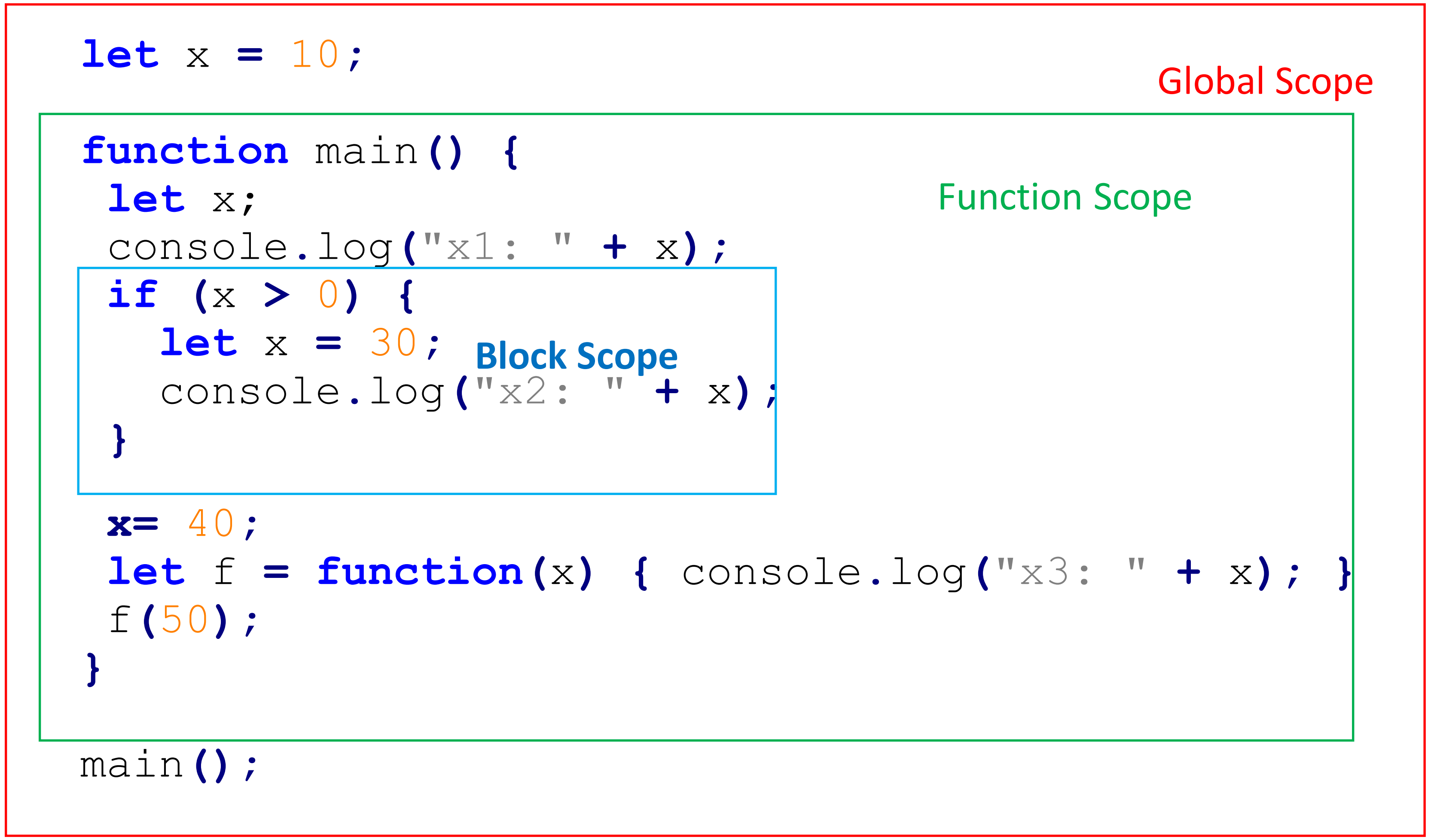
```
let userName = 'John’;

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.
- See example: *lecture_codes/lesson5/scopes.js*
    - which lines will cause errors, why?

Lexical scope in JavaScript (ES6+)

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



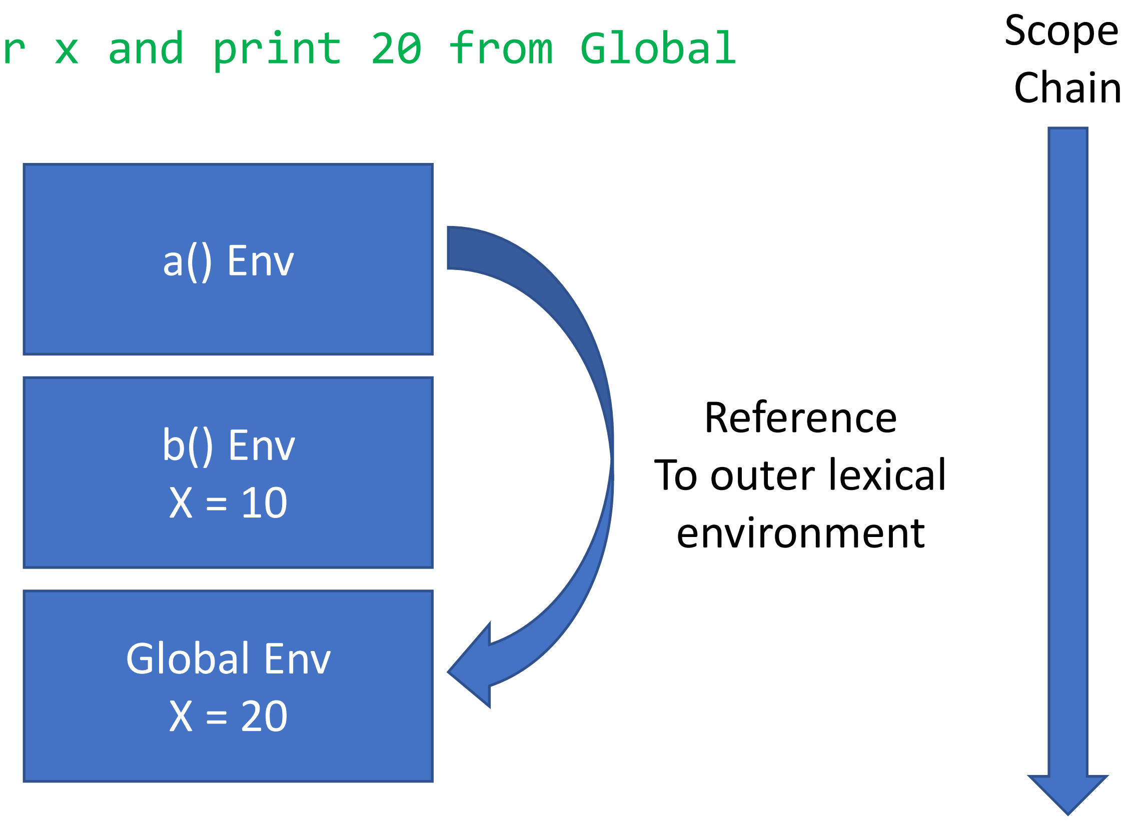
```
let x = 10;

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();
```

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.

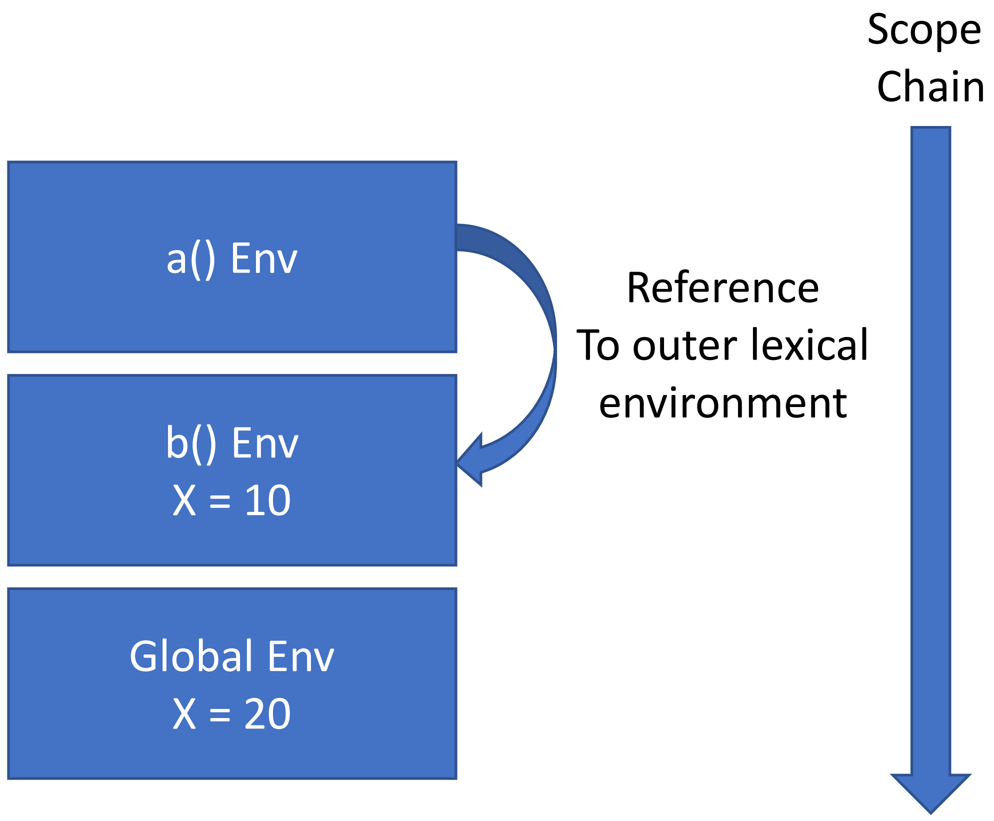
Scope Example

```
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();
```



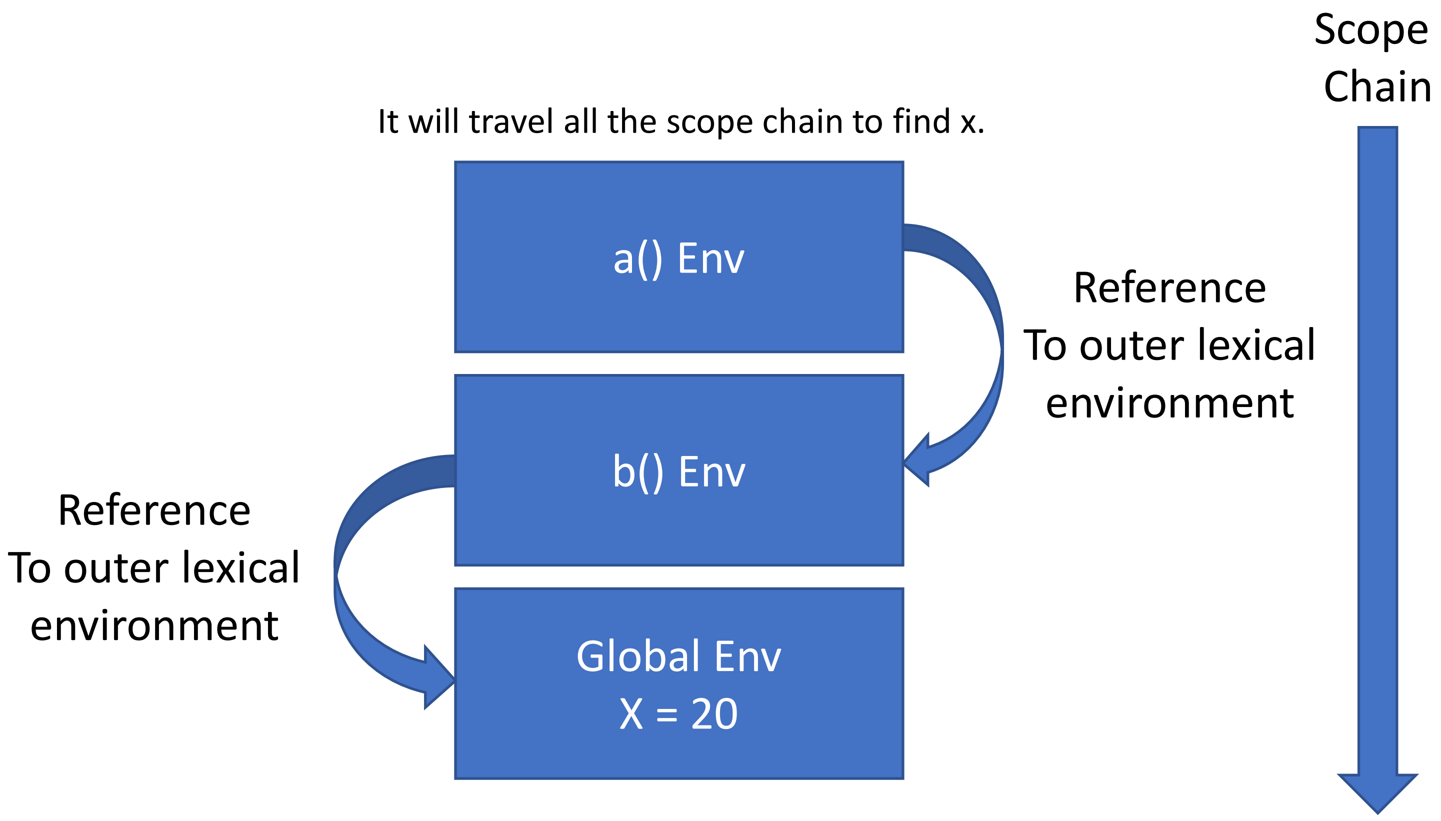
Scope Example

```
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 Chain
It will travel all the scope chain to find x.
a() Env
Reference To outer lexical environment
b() Env
Reference To outer lexical environment
Global Env
X = 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, JS engine 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 the unbounded awareness*.

Exercise

- Write a function to compute area of a triangle based on the following formula
  - Area = $\sqrt{s(s-a)(s-b)(s-c)}$
    - 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.

Software design principles for functions

- avoid globals
- avoid side effects
- a pure function takes arguments and returns a value
    - does not change arguments
    - returns a value
    - does not change any variables or state outside the function

- a function should be a command or a query, not both
    - tendency is for people to reuse functions that return values to get the value
    - if it also has a side effect (updating a database, printing, etc) can be unexpected and produce a bug

- 30 second rule: should take 30 seconds or less to read and understand, else too long
- functions should be self contained and only require user to know signature and return value

Avoid premature optimization

- avoid break and continue
- "premature optimization is the root of all evil (in programming)“
- 3 laws of optimization
  - don’t
  - later
  - only after profile

Function expression & Anonymous Function

- The syntax that we used before is called a *Function Declaration*
- There is another syntax for creating a function that is called a *Function Expression*.
  - A function keyword can be used to define a function inside an expression

```
// function expression
let sayHi = function(){console.log("Hi");};
sayHi();
```

  - In JavaScript, a function is a value, so we can deal with it as a value.

- Function without a name is called *anonymous* function.

Arrow function

- New syntax introduced in ES6 to write a function in concise way

```
let isEven = (a) => {return a%2===0;}
console.log(isEven(4));

let isOdd = (a) => a%2 !== 0;
console.log(isOdd(7));

let sayHello = () => console.log('HI');
sayHello();
```

```
(arguments) => { return statement } // general syntax
  argument => { return statement } // one parameter
      argument => statement // implicit return
            () => statement // no parameter
```

The execution context and stack

- The information about the process of execution of a running function is stored in its *execution context*.
- The execution context is an internal data structure that contains details about the execution of a function: most importantly the current variables the function is using.
- One function call has exactly one execution context associated with it.
- When a function makes call to another function, the following happens
  - The current function is paused
  - The execution context associated with it is remembered in a special data structure called *execution context stack*.
  - Called function executes
  - After it ends, the calling function is resumed with prior saved *execution context*.

Function calling another function

```
// Output?

function A(){
    console.log("A is called");
    console.log("Before B is called");
    B();
    console.log("After B is called")
}

function B(){
    console.log("B is called");
    console.log("Before C is called");
    C();
    console.log("After C is called");
}

function C(){
    console.log("C is called");
}
A();
console.log("After A is called");
```

Example: Lets draw a stack

```
function funA(a,n){
    let something;
    something = "something.";
    funB(something, n);
}

function funB(a,b){
    let thing;
    thing = "a thing.";
    console.log("What is on the stack when we're here?");
}

function main(){
    let test;
    let n;
    test = "Hello";
    n = 5;
    funA(n, 10);
}

main();
```

Exercise: Draw the stack

```
function funX(a, b) {
    let c;
    c = 5;
    funY(a * c, "yes");
}

function funY(x, y) {
    let z;
    z = "I can see the sea";
    console.log("What is on the stack here?");
}

function main() {
    let a;
    let b;
    a = "Hello";
    funX(3, a);
    b = "World";
}

main();
```

References

- Functions (javascript.info)

- Function expressions (javascript.info)
- Arrow functions, the basics (javascript.info)