[Lesson 1-2]

Roi Yehoshua 2018

[DAN.IT EDUCATION]

[Our targets for today]

- Destructuring
- Closures
- Function declaration
- Named functional expressions
- Immediately invoked functional expressions
- Functions call and apply

[DAN.IT EDUCATION]

Destructuring Assignment

→ Destructuring assignment allows for instantly “unpacking” arrays or objects into a bunch of variables, as sometimes they are more convenient

→ Destructuring also works great with complex functions that have many parameters

→ An example of how an array is destructured into variables:

```
// we have an array with first name and last name
let arr = ["Roi", "Yehoshua"];

// destructuring assignment
let [firstName, lastName] = arr;

// a shorter way for writing:
// let firstName = arr[0];
// let lastName = arr[1];

alert(firstName); // Roi
alert(lastName);   // Yehoshua
```

[DAN.IT EDUCATION]

Destructuring Assignment

→ Unwanted elements of the array can be thrown away via an extra comma:

```
// skipping the first and second elements, the third one is assigned to title,
// and the rest are also skipped
let [, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(title); // Consul
```

→ We can use destrucutring assignment with any iterable, not only arrays:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

→ We can assign to anything at the left side, e.g., an object property:

```
let user = {};
[user.firstName, user.lastName] = "John Smith".split(' ');

alert(user.firstName); // John
```

[DAN.IT] EDUCATION

Destructuring Assignment

→ We can use destructuring to loop over keys-and-values of a map:

```
let countryCodes = new Map();
countryCodes.set("US", "United States");
countryCodes.set("FR", "France");
countryCodes.set("IL", "Israel");

for (let [key, value] of countryCodes.entries()) {
    alert(`${key}:${value}`); // US: United States, FR: France, IL: Israel
}
```

Object Destructuring

→ The destructuring assignment also works with objects

→ The basic syntax is:

```
let {var1, var2} = {var1:..., var2...}
```

→ For example:

```
let options = {
      title: "Menu",
      width: 100,
      height: 200
};
let { title, width, height } = options;

alert(title);      // Menu
alert(width);      // 100
alert(height);     // 200
```

→ The properties options.title, options.width and options.height are assigned to the corresponding variables. The order of the variables on the left side does not matter.

[DAN.IT] EDUCATION

Object Destructuring

→ If we want to assign a property to a variable with another name, e.g., options.width to go into the variable named w, then we can set it using a colon:

```
let options = {
      title: "Menu",
      width: 100,
      height: 200
};

// { sourceProperty: targetVariable }
let { width: w, height: h, title } = options;

// width -> w
// height -> h
// title -> title

alert(title);      // Menu
alert(w);          // 100
alert(h);          // 200
```

Object Destructuring

→ For potentially missing properties we can set default values using "=", like this:

```
let options = {
    title: "Menu"
};
let { width = 100, height = 200, title } = options;

alert(title);      // Menu

alert(width);      // 100
alert(height); // 200
```

→ Just like with arrays or function parameters, default values can be any expressions or even function calls. They will be evaluated if the value is not provided.

Object Destructuring

→ We can use existing variables on the left side of the destructuring assignment

→ But there's a catch:

```
let title, width, height;

// error in this line
{ title, width, height } = { title: "Menu", width: 200, height: 100 };
```

→ The problem is that JavaScript treats {...} as a code block

→ To show JavaScript that it's not a code block, we need to wrap the whole assignment in brackets (...):

```
let title, width, height;

// okay now
({ title, width, height } = { title: "Menu", width: 200, height: 100 });

alert(title); // Menu
```

[DAN.IT] EDUCATION

Smart Function Parameters

→ There are times when a function has many parameters, most of which are optional

→ Imagine a function that creates a menu. It may have a width, a height, a title, items list and so on.

→ Here's a bad way to write such function:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {
    // ...
}
```

→ The problem is how to remember the order of arguments, and also how to call such a function when most parameters are ok by default. Like this?

```
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"]);
```

→ That's ugly, and becomes unreadable when we deal with more parameters

[DAN.IT] EDUCATION

Smart Function Parameters

→ Destructuring comes to the rescue!

→ We can pass parameters as an object, and the function immediately destructurizes them into variables:

```
// we pass object to function
let options = {
    title: "My menu",
    items: ["Item1", "Item2"]
};

// ...and it immediately expands it to variables
function showMenu({ title = "Untitled", width = 200, height = 100, items = [] }) {
    // title, items – taken from options, width, height – defaults used
    alert(`${title} ${width} ${height}`); // My Menu 200 100
    alert(items); // Item1, Item2
}

showMenu(options);
```

[DAN.IT EDUCATION]

Smart Function Parameters

→ We can also use more complex destructuring with nested objects and colon mappings:

```
let options = {  title:
    "My menu",
    items: ["Item1", "Item2"]
};

function showMenu({
    title = "Untitled",
    width:  w    =  100,  // width goes to w
    height: h = 200, // height goes to h
    items: [item1, item2] // items first element goes to item1, second to item2
}) {
    alert(`${title} ${w} ${h}`); // My Menu 100 200
    alert(item1); // Item1
    alert(item2); // Item2
}

showMenu(options);
```

Exercise (1)

→ We have an object:

```
let user = { name: "John", years: 30 };
```

→ Write the destructuring assignment that reads:

- → name property into the variable name
- → years property into the variable age
- → isAdmin property into the variable isAdmin (false if absent)

→ The values after the assignment should be:

```
let user = { name: "John", years: 30 };

// your code to the left side:
// ... = user;

alert(name); // John
alert(age); // 30
alert(isAdmin); // false
```

[Closure]

→ We know that a function can access variables outside of it

→ This feature is used quite often in JavaScript

→ But what happens when an outer variable changes? Does a function get the most recent value or the one that existed when the function was created?

```
let name = "John";

function sayHi() {
    alert("Hi, " + name);
}

name = "Adam";

sayHi(); // what will it show: "John" or "Adam"?
```

[DAN.IT] EDUCATION

[Lexical Environment]

→ In JavaScript, every running function, code block, and the script as a whole have an associated object known as the *Lexical Environment*

→ The **Lexical Environment** object consists of two parts:

  → *Environment Record* – an object that has all local variables as its properties (and some other information like the value of this)

  → A reference to the *outer lexical environment*, usually the one associated with the code lexically right outside of it (outside of the current curly brackets)

→ So, a “variable” is just a property of the special internal object, Environment Record

  → “To get or change a variable” means “to get or change a property of the Lexical Environment”

→ For instance, in this simple code, there is only one Lexical Environment

  → the global environment associated with the whole script

[DAN.IT EDUCATION]

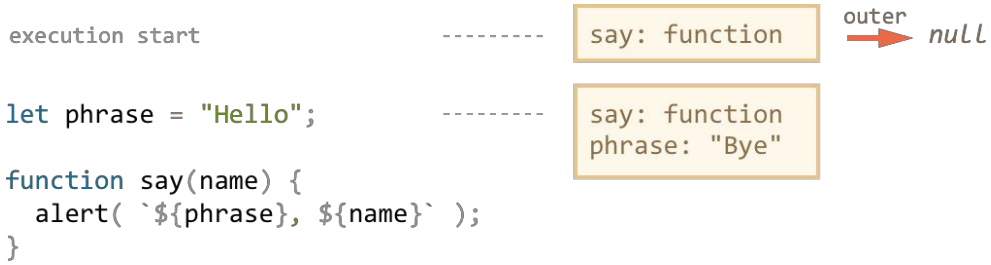
[Function Declaration]

→ Function declarations are special

→ Unlike let variables, they are processed not when the execution reaches them, but when a Lexical Environment is created

→ For the global Lexical Environment, it means the moment when the script is started

→ That is why we can call a function declaration before it is defined



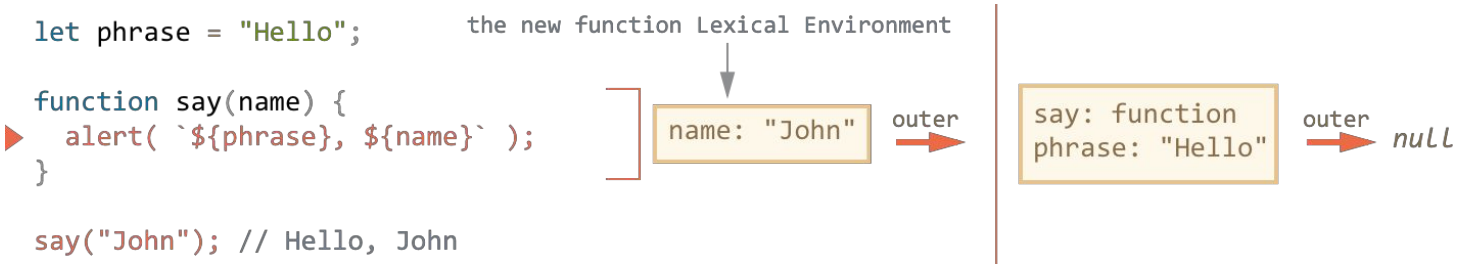
[DAN.IT EDUCATION]

Inner and Outer Lexical Environments

→ When a function runs, a new function Lexical Environment is created automatically

→ That Lexical Environment is used to store local variables and parameters of the call

→ Here's the picture of Lexical Environments when the execution is inside say("John"), at the line labeled with an arrow:

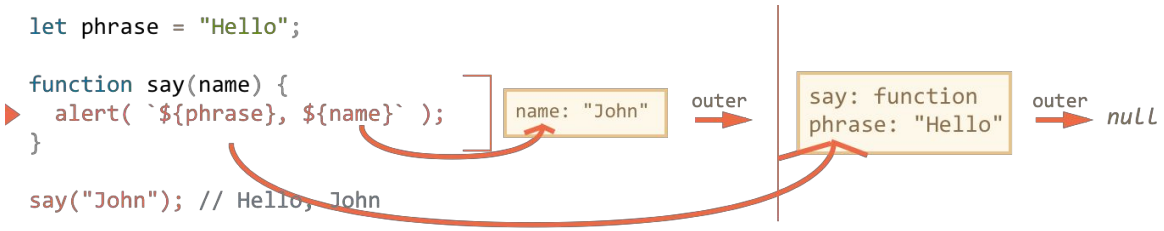


→ During the function call we have two Lexical Environments: the inner one (for the function call) and the outer one (global)

→ The inner Lexical Environment has the outer reference to the outer one

[Inner and Outer Lexical Environments]

→ When code wants to access a variable – it is first searched for in the inner Lexical Environment, then in the outer one, then the more outer one and so on until the end of the chain



→ If a function is called multiple times, then each invocation will have its own Lexical Environment, with local variables and parameters specific for that very run.

Inner and Outer Lexical Environments

→ Because of the described mechanism **a function gets outer variables as they are now**

→ It takes its current value from its own or an outer Lexical Environment

```
let name = "John";

function sayHi() {
    alert("Hi, " + name);
}
name = "Adam"; // (*)

sayHi(); // Hi, Adam
```

→ The global Lexical Environment has name: "John"

→ At the line (*) the global variable is changed, now it has name: "Adam"

→ When the function say(), is executed and takes name from outside. Here that's from the global Lexical Environment where it's already "Adam"

[DAN.IT EDUCATION]

[Nested Functions]

→ A function is called “nested” when it is created inside another function

→ We can use it to organize our code, like this:

```
function sayHiBye(firstName, lastName) {

    // helper nested function to use below
    function getFullName() {
        return firstName + " " + lastName;
    }

    alert("Hello, " + getFullName());
    alert("Bye, " + getFullName());
}
```

→ The nested function getFullName() can access the outer variables firstName and lastName

[DAN.IT] EDUCATION

[Nested Functions]

→ A nested function can be returned and then be used somewhere else

→ No matter where, it still has access to the same outer variables

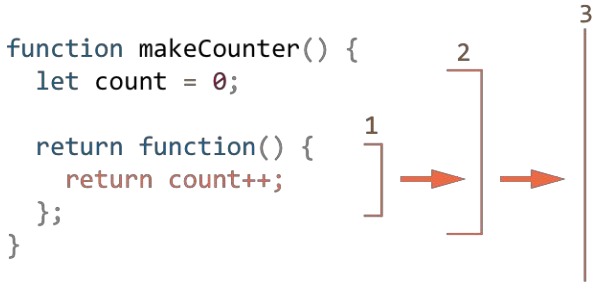
```
function makeCounter() {
    let count = 0;

    return function () {
        return count++; // has access to the outer counter
    };
}
let counter = makeCounter();

alert(counter()); // 0

alert(counter()); // 1
alert(counter()); // 2
```

[DAN.IT EDUCATION]

[Nested Functions]

→ When the inner function runs, the variable in count++ is searched from inside out:

- → The locals of the nested function…
- → The variables of the outer function…
- → And so on until it reaches global variables.



→ In this example count is found on step 2, so count++ finds the outer variable and increases it in the Lexical Environment where it belongs

[DAN.IT] EDUCATION

Nested Functions

→ For every call to makeCounter() a new function Lexical Environment is created, with its own counter

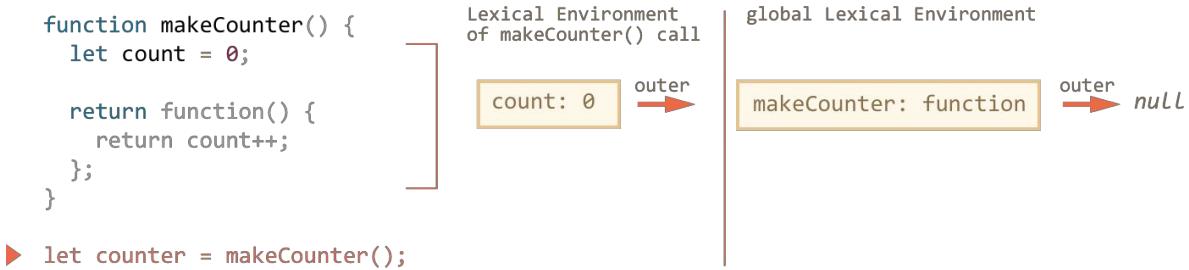
→ So the resulting counter functions are independent

```
function makeCounter() {
    let count = 0;
    return function () {
        return count++;
    };
}

let counter1 = makeCounter();
let counter2 = makeCounter();

alert(counter1());  // 0
alert(counter1());  // 1
alert(counter1());  // 2

alert(counter2());  // 0 (independent)
```

[Nested Functions]

→ Behind the scenes, all functions “on birth” receive a hidden property [[Environment]] with a reference to the Lexical Environment of their creation

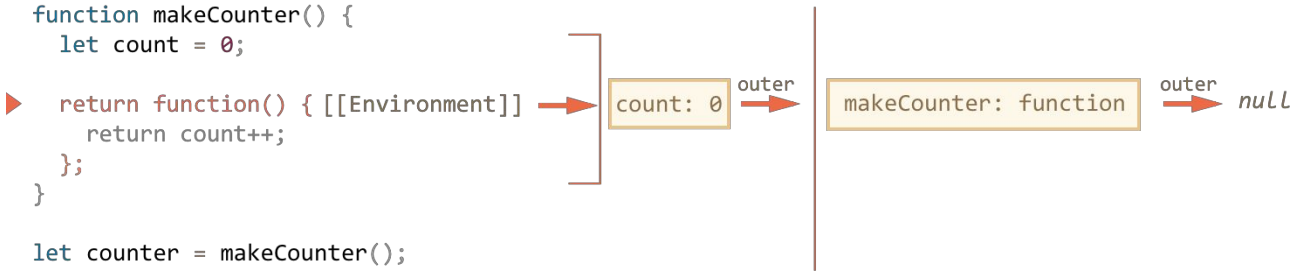
→ At the moment of the call of makeCounter(), the Lexical Environment is created, to hold its variables and argument



[Nested Functions]

→ During the execution of makeCounter(), a tiny nested function is created

→ The value of its [[Environment]] is the current Lexical Environment of make Counter() (where it was born):



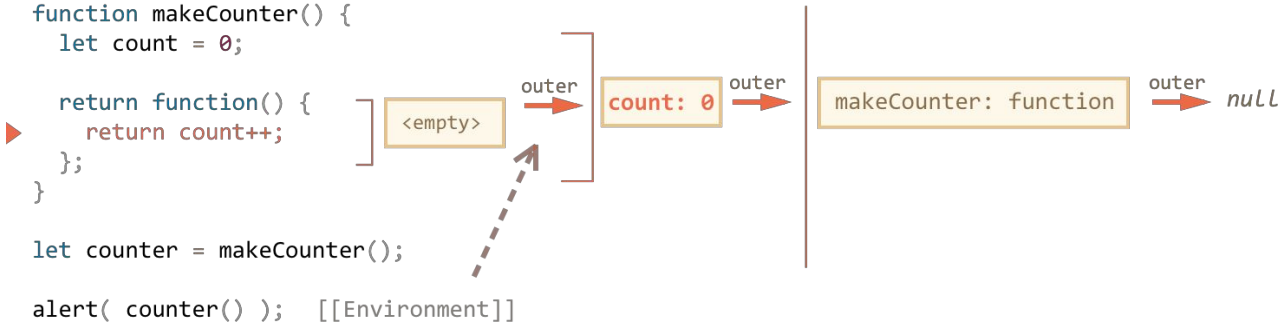
→ The result (the tiny nested function) is assigned to the global variable counter

[Nested Functions]

→ When the counter() is called, an “empty” Lexical Environment is created for it

→ It has no local variables by itself.

→ But the [[Environment]] of counter is used as the outer reference for it, so it has access to the variables of the former makeCounter() call where it was created:



→ When it looks for count, it finds it among the variables makeCounter, in the nearest outer Lexical Environment

[DAN.IT] EDUCATION

[Closures]

→ There is a general programming term “closure”, that developers should know

→ A closure is a function that remembers its outer variables and can access them

→ In some languages, that’s not possible, or a function should be written in a special way to make it happen

→ As explained above, in JavaScript all functions are naturally closures

→ That is: they automatically remember where they were created, and all of them can access outer variables

[DAN.IT EDUCATION]

Code Blocks

→ We also can use a “bare” code block {…} to isolate variables into a “local scope”

→ For instance, in a web browser all scripts share the same global area

→ So if we create a global variable in one script, it becomes available to others

→ That becomes a source of conflicts if two scripts use the same variable name

→ If we’d like to avoid that, we can use a code block to isolate the script:

```
{
    // do some job with local variables that should not be seen outside
    let message = "Hello";
    alert(message); // Hello
}

alert(message); // Error: message is not defined
```

→ The code outside of the block (or inside another script) doesn’t see variables inside the block, because the block has its own Lexical Environment

[DAN.IT EDUCATION]

[IIFE]

→ In old scripts, one can find “immediately-invoked function expressions” (IIFE) used for the same purpose

→ They look like this:

```
(function () {
    let message = "Hello";
    alert(message); // Hello
})();
```

→ Here a Function Expression is created and immediately called

→ So the code executes right away and has its own private variables

[DAN.IT EDUCATION]

[Exercise (2)]

→ What will be the output of the following function?

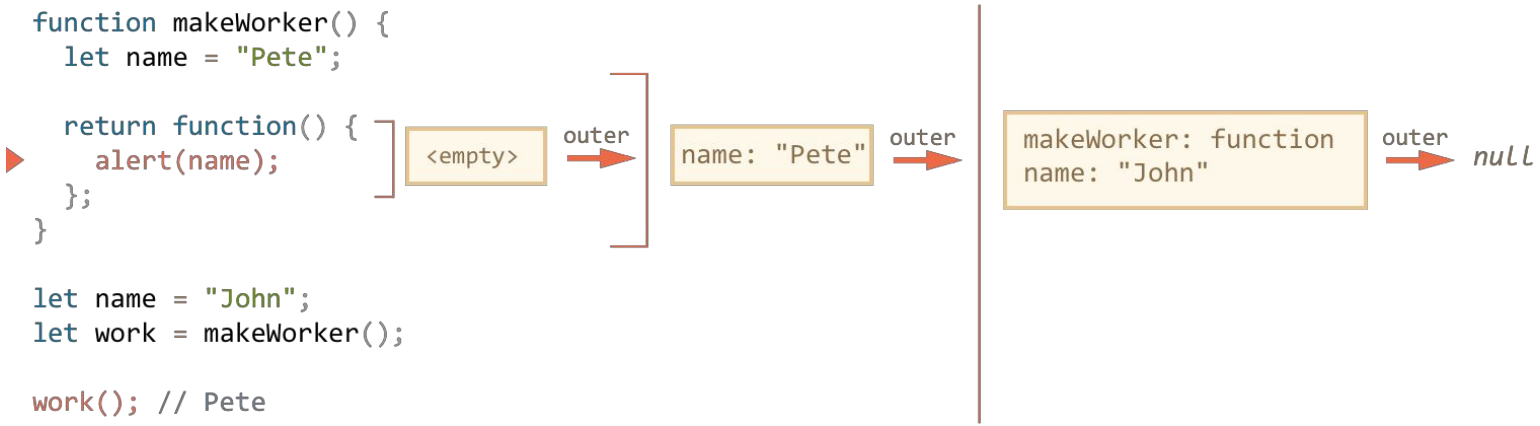
→ Draw a diagram of the lexical environments when execution reaches the line with (*)

```
function makeWorker() {
    let name = "Pete";

    return function () {
        alert(name); // (*)
    };
}

let name = "John";

// create a function
let work = makeWorker();

// call it
work(); // what will it show? "Pete" (name where created) or "John" (name where called)?
```

[DAN.IT] EDUCATION

[Solution]

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "John";
let work = makeWorker();

work(); // Pete
```



Exercise (3)

→ Here a counter object is made with the help of the constructor function

→ Will it work? What will it show?

```
function Counter() {
    let count = 0;

    this.up = function () {
        return ++count;
    };
    this.down = function () {
        return --count;
    };
}
let counter = new Counter();

alert(counter.up()); // ?

alert(counter.up());  // ?
alert(counter.down()); // ?
```

[Exercise (4)]

→ We have a built-in method arr.filter(f) for arrays

→ It filters all elements through the function f. If f returns true, then that element is returned in the resulting array.

→ Make a set of “ready to use” filters:

→ inBetween(a, b) – between a and b or equal to them (inclusively)

→ inArray([...]) – in the given array

→ For instance:

```
/* .. your code for inBetween and inArray */
let arr = [1, 2, 3, 4, 5, 6, 7];

alert(arr.filter(inBetween(3,  6))); // 3,4,5,6

alert(arr.filter(inArray([1, 2,  10]))); // 1,2
```

[DAN.IT] EDUCATION

Call and apply

→ There's a special built-in function method **func.call()** that allows to call a function explicitly setting **this**

→ The syntax is: `func.call(context, arg1, arg2, ...)`

→ It runs func providing the first argument as this, and the next as the arguments

→ As an example, in the code below we call sayHi in the context of different objects

```
function sayHi() {
    alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// use call to pass different objects as "this"
sayHi.call(user); // this = John
sayHi.call(admin); // this = Admin
```

[DAN.IT] EDUCATION

Call and apply

→ And here we use call to call say with the given context and phrase:

```
function say(time, phrase) {
    alert(`[${time}] ${this.name}: ${phrase}`);
}

let user = { name: "John" };
say.call(user, '10:00', 'Hello'); // [10:00] John: Hello (this=user)
```

→ There is another built-in method **func.apply()** that works almost the same as func.call(), but takes an array-like object instead of a list of arguments:

```
function say(time, phrase) {
    alert(`[${time}] ${this.name}: ${phrase}`);
}

let user = { name: "John" };
let messageData = ['10:00', 'Hello']; // become time and phrase

// user becomes this, messageData is passed as a list of arguments (time, phrase)
say.apply(user, messageData); // [10:00] John: Hello (this=user)
```

[DAN.IT] EDUCATION

Call and apply

→ There is another built-in method func.apply() that works almost the same as func.call()

```
func.apply(context, args)
```

→ The syntax is:

→ The only syntax difference between call and apply is that call expects a list of arguments, while apply takes an array-like object with them

```
function say(phrase) {
    alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// user becomes this, and "Hello" becomes the first
argument  say.call(user, "Hello"); // John: Hello
```

Control questions

1. What is Destructuring Assignment and when it should be used?
2. What are Smart Function Parameters and when they should be used?
3. What is Lexical environment?
4. What is Closure?
5. How do you use closures?
6. What happens when we put a function inside another function?
7. What is scope and what defines one?
8. What is Immediately invoked functional expression?