Week 9 WDD330

lunes, 12 de junio de 2023 11:11

Ruth Cordova!

VARIABLE SCOPE, CLOSURE

```
for (let i = 0; i < 3; i++) {
  // the variable i is only visible inside this for
  alert(i); // 0, then 1, then 2
}

alert(i); // Error, no such variable
```

→ LOCAL SCOPE

→ GLOBAL SCOPE

NESTED FUNCTIONS

WHEN A FUNCTION IS CREATED INSIDE ANOTHER FUNCTION

1)

```
function sayHiBye(firstName, lastName) {

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

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

}
```

2)

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

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

FUNCTIONS CAN BE RETURNED

LEXICAL ENVIRONMENT

EVERY RUNNING FUNCTION, CODE BLOCK, AND THE SCRIPT AS A WHOLE HAVE AN INTERNAL (HIDDEN) ASSOCIATED OBJECT KNOWN AS THE LEXICAL ENVIRONMENT

- IT ONLY EXISTS THEORETICALLY TO DESCRIBE HOW THINGS WORK

THE OBJECT CONSIST OF TWO PARTS:

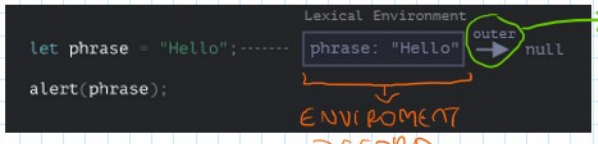
1 - ENVIRONMENT RECORDS → STORES ALL LOCAL VARIABLES AS ITS PROPERTIES

2 - A REFERENCE TO THE OUTER CODE

① VARIABLES:

- IT'S JUST A PROPERTY OF THE SPECIAL INTERNAL OBJECT, ENVIRONMENT RECORD.
- TO GET OR CHANGE A VARIABLE MEANS TO GET OR CHANGE A PROPERTY OF THAT OBJECT

GLOBAL LEXICAL ENVIRONMENT



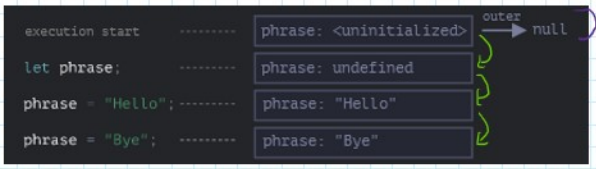
→ OUTER REFERENCE (global = NO REFERENCE)

ENVIRONMENT RECORD (VARIABLE STORE)

- LEXICAL ENVIRONMENT CHANGES WHILE CODE EXECUTION

→ LEXICAL ENVIRONMENT IS PRE-POPULATED

- LEXICAL ENVIROMENT CHANGES WHILE CODE EXECUTION

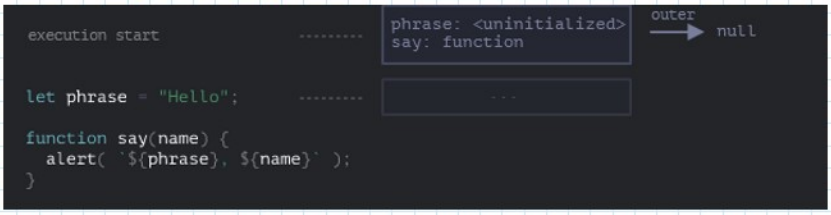


-> LEXICAL ENVIROMENT IS PRE-POPULATED WITH ALL DECLARED VARIABLES.
(THE ENGINE KNOWS ABOUT THEM BEFORE DECLARING)

- A VARIABLE IS A PROPERTY OF A SPECIAL INTERNAL OBJECT, ASSOCIATED WITH THE CURRENTLY EXECUTING BLOCK / FUNCTION / SCRIPT.
- WORKING WITH VARIABLES IS WORKING WITH THE PROPERTIES OF THAT OBJECT

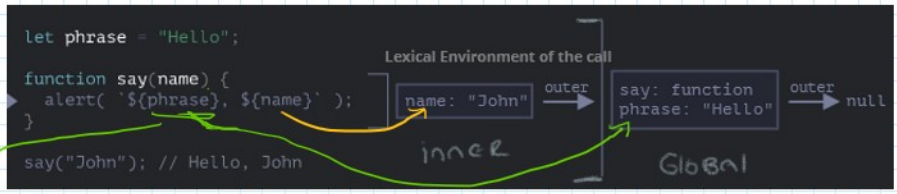
(2) FUNCTION DECLARATIONS

- THEY ARE INSTANTLY FULLY INITIALIZED (READY-TO-USE)



(3) INNER AND OUTER LEXICAL ENVIROMENT

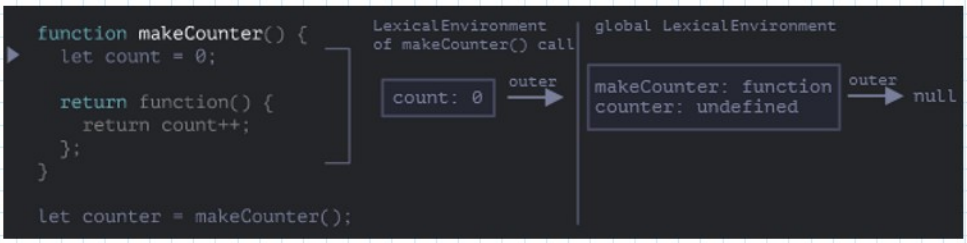
WHEN A FUNCTIONS RUNS, AT THE BEGINNING OF THE CALL, A NEW LEXICAL ENVIROMENT IS CREATED AUTOMATICALLY TO STORE LOCAL VARIABLES AND PARAMETERS OF THE CALLED



THE VARIABLE IS SEARCH FIRST IN THE INNER LEX. ENV.
THEN THE OUTTER ONE, THEN THE MORE OUTER ONE
AND SO UNTIL THE GLOBAL ONE

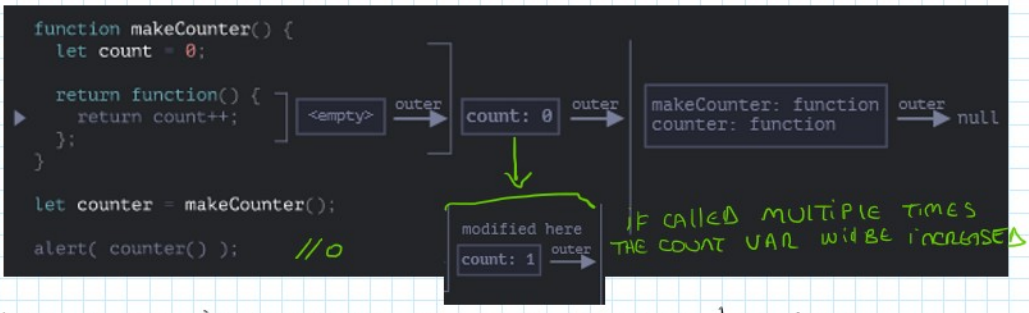
- IF VARIABLE IS NOT FOUND, GIVES AN ERROR in STRICT MODE

(4) RETURNING A FUNCTION



- ALL FUNCTIONS REMEMBER THE LEX. ENV IN WHICH THEY WERE MADE.
- ALL FUNCTIONS HAVE A PROPERTY NAME [[Environment]]

- COUNTER.[[ENVIRONMENT]] HAS A REFERENCE TO {COUNT: 0} LEX ENV.



- WHEN COUTER IS CALLED IS CREATED A NEW LEX. ENV., AND THE OUTER REFERENCE IS TAKEN FROM COUNTER.[[ENVIRONMENT]]

- A VARIABLE IS UPDATED IN THE LEXICAL ENVIROMENT WHERE IT LIVES

CLOSURE = A FUNCTION THAT REMEMBERS ITS OUTER VARIABLES AND CAN ACCESS THEM.

- IN JS, ALL FUNCTIONS ARE NATURALLY CLOSURES, THEY AUTOMATICALLY REMEMBER WHERE THEY WERE CREATED USING A HIDDEN `[[Environment]]` PROPERTY.

GARBAGE COLLECTION

- LEX ENV IS REMOVED FROM MEMORY WITH ALL VARIABLES AFTER THE FUCTION CALL FINISHED, BECAUSE THERE ARE NO REFERENCES TO IT.

- IN NESTED FUNCTION THAT IS STILL REACHABLE AFTER THE END OF A FUNCTION (STAYS ALIVE)

```
1  function f() {
2    let value = 123;
3
4    return function() {
5      alert(value);
6    }
7  }
8
9  let g = f(); // g.[[Environment]] stores a reference to the Lexical Environment
10 // of the corresponding f() call
```

→ IF f() IS CALLED MANY TIMES THE RESULTING FUNCTIONS ARE SAVED.

- A LEX.ENV. OBS. DIES WHEN IT BECOMES UNREACHABLE

CURRYING

- ITS A TRANSFORMATION OF FUNCTIONS

$$F(a,b,c) \longrightarrow F(a)(b)(c)$$

```
function curry(f) { // curry(f) does the currying transform
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// usage
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3
```

- The result of `curry(func)` is a wrapper `function(a)`.
- When it is called like `curriedSum(1)`, the argument is saved in the Lexical Environment, and a new wrapper is returned `function(b)`.
- Then this wrapper is called with `2` as an argument, and it passes the call to the original `sum`.

FROM lodash LIBRARY WE CAN USE _.CURRY, THAT RETURNS A WRAPPER THAT ALLOWS A FUNCTION TO BE CALLED BOTH NORMALLY AND PARTIALLY

```
function sum(a, b) {
  return a + b;
}

let curriedSum = _.curry(sum); // using _.curry from lodash library

alert( curriedSum(1, 2) ); // 3, still callable normally
alert( curriedSum(1)(2) ); // 3, called partially
```

FOR MULTI-ARGUMENT FUNCTIONS, THERE IS AN ADVANCE CURRY IMPLEMENTATION

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {   // (1)
      return func.apply(this, args);
    } else {
      return function(...args2) {       // (2)
        return curried.apply(this, args.concat(args2));
      }
    }
  };

}
```

EXAMPLE:

```
function sum(a, b, c) {
  return a + b + c;
}

```

```
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, still callable normally
alert( curriedSum(1)(2,3) ); // 6, currying of 1st arg
alert( curriedSum(1)(2)(3) ); // 6, full currying
```

- THE RESULT OF CURRY(FUNC) CALL IS THE WRAPPER CURRIED

- THE CURRYING REQUIRES THE FUNCTION TO HAVE A FIXED NUMBER OF ARGUMENTS
- THE FUNCTION CAN'T BE CURRIED IF TAKES REST PARAMETERS (F(...ARGS))

WEB APIs: DRAWING GRAPHICS

THE CANVAS API: USE FOR DRAW ON HTML

TO CREATE A 2D OR 3D SCENE ON A WEB PAGE YOU NEED TO START WITH AN HTML <CANVAS> ELEMENT, TO DEFINE THE AREA ON THE PAGE.

```
<canvas width="320" height="240">
  <p>Description of the canvas for
those unable to view it.</p>
</canvas>
```

FALLBACK CONTENT TO DESCRIBE THE CANVAS CONTENT TO USERS OF BROWSERS THAT DON'T SUPPORT CANVAS. (SCREEN READERS)

THE CANVAS TEMPLATE

HTML

```
<canvas class="myCanvas">
  <p>Add suitable fallback here.</p>
</canvas>
```

JS

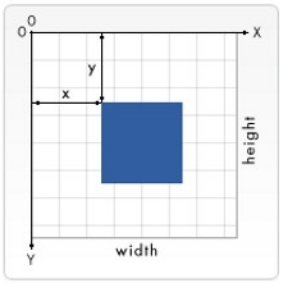
```
const canvas = document.querySelector(".myCanvas");
const width = (canvas.width = window.innerWidth);
const height = (canvas.height = window.innerHeight);

const ctx = canvas.getContext("2d");
```

THE TYPE OF CONTENT YOU WANT TO RETRIVE

```
ctx.fillStyle = "rgb(0, 0, 0)";
ctx.fillRect(0, 0, width, height);
```

- 2D CANVAS OPERATIONS ARE DONE WITH A CanvasRenderingContext2D OBJECT (ctx)

- MANY OPERATIONS NEEDS THE GIVEN COORDINATES FROM THE X AND Y AXIS (X, Y)
- (0,0) IS THE TOP LEFT OF THE CANVAS



SIMPLE RECTANGLES

```
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.fillRect(50, 50, 100, 150);
```

COLOR AREA

CAN ALSO USE ALPHA CHANNEL (TRANSPARENCY)

X, Y

WIDTH

HEIGHT

STROKES AND LINE WIDTHS

```
ctx.strokeStyle = "rgb(255, 255, 255)";
ctx.strokeRect(25, 25, 175, 200);
```

```
ctx.lineWidth = 5;
```

DRAWING PATHS

WRITING CODE TO SPECIFY THE EXACTIX PATH OF THE PEN AND TRACE THE SHAPE YOU WANT TO DRAW.

METHODS

- `beginPath()` — start drawing a path at the point where the pen currently is on the canvas. On a new canvas, the pen starts out at (0, 0).
- `moveTo()` — move the pen to a different point on the canvas, without recording or tracing the line; the pen "jumps" to the new position.
- `fill()` — draw a filled shape by filling in the path you've traced so far.
- `stroke()` — draw an outline shape by drawing a stroke along the path you've drawn so far.
- You can also use features like `lineWidth` and `fillStyle` / `strokeStyle` with paths as well as rectangles.

EXAMPLES

```
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.beginPath();
ctx.moveTo(50, 50);
// draw your path
ctx.fill();
```

TRIANGLE

DEGREES TO RADIANS

```
function degToRad(degrees) {
  return (degrees * Math.PI) / 180;
```

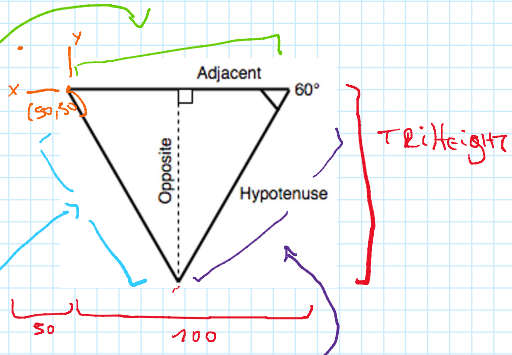
Triangle

DEGREES TO RADIANS

```
function degToRad(degrees) {
  return (degrees * Math.PI) / 180;
}
```

```
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.beginPath();
ctx.moveTo(50, 50);
```

```
ctx.lineTo(150, 50);
const triHeight = 50 *
Math.tan(degToRad(60));
ctx.lineTo(100, 50 + triHeight);
ctx.lineTo(50, 50);
ctx.fill();
```



Circle

```
ctx.fillStyle = "rgb(0, 0, 255)";
ctx.beginPath();
ctx.arc(150, 106, 50, degToRad(0),
degToRad(360), false);
ctx.fill();
```

x y radius — start angle — end angle

FALSE = CLOCKWISE
TRUE = COUNTERCLOCKWISE

Text

```
ctx.strokeStyle = "white";
ctx.lineWidth = 1;
ctx.font = "36px arial";
ctx.strokeText("Canvas text", 50, 50);

ctx.fillStyle = "red";
ctx.font = "48px georgia";
ctx.fillText("Canvas text", 50, 150);

canvas.setAttribute("aria-label",
"Canvas text");
```

- fillText() draws filled text
- strokeText() draws outline (stroke) text.

Images onto Canvas

```
const image = new Image();
image.src = "firefox.png";
```

```
image.addEventListener("load", ()
=> ctx.drawImage(image, 20, 20));
```

To display only a part of the image

```
ctx.drawImage(image, 20, 20, 185,
175, 50, 50, 185, 175);
```

x1 y1 w1 — starting points
h1 — w2 h2
where to

WHERE TO CUT K_2 Y_2

```
canvas.setAttribute("aria-label",
"Firefox Logo");
```

RESULT =>

Loops an Animations

EXAMPLE USING A FOR LOOP (FROM MDN)

https://github.com/mdn/learning-area/tree/main/javascript/apis/drawing-graphics/loops_animation/6_canvas_for_loop

SOME PROPERTIES USED:

TRANSLATE() MOVES THE ORIGIN POINT OF THE CANVAS

DEGTORAD() FUNCTION TO TRANSFORM DEGREES TO RADS

RAND() FUNCTION THAT RETURNS A RANDOM NUMBER

FOR ANIMATIONS: THERE ARE JS FUCTIONS THAT WILL ALLOW YOU TO RUN FUNCTIONS REPEATEDLY.

- WINDOW.REQUESTANIMATIONFRAME (1 PARAM) TAKES THE NAME OF THE FUNCTION TO RUN EACH FRAME
- WINDOW.CANCELANIMATIONFRAME() CALL IT WHEN YOU ARE DONE WITH THE ANIMATION LOOP (IS A GOOD PRACTICE TO USE THIS FUNCTION)

- STEPS FOR DOING A CANVAS ANIMATION

1. Clear the canvas contents (e.g. with `fillRect()` or `clearRect()`).
2. Save state (if necessary) using `save()` — this is needed when you want to save settings you've updated on the canvas before continuing, which is useful for more advanced applications.
3. Draw the graphics you are animating.
4. Restore the settings you saved in step 2, using `restore()`
5. Call `requestAnimationFrame()` to schedule drawing of the next frame of the animation.

WebGL API

- IT'S BASE ON OPENGL (OPEN GRAPHICS LIBRARY)
- ALLOWS YOU TO COMMUNICATE DIRECTLY WITH THE COMPUTER GPU

- It's base on OpenGL (Open Graphics library)
- Allows you to communicate directly with the computer GPU
- Writing raw WebGL is closer to low level languages
- Because of its complexity, most people write 3D graphics code using third party JS library (three.js, PlayCanvas, Babylon.js)

Example of a rotating cube with three.js library, from MDN

https://github.com/mdn/learning-area/tree/main/javascript/apis/drawing-graphics/threejs-cube