The Basics -2 (16 coding exercises)

"All skills are learnable." — B.Tracy

2.1 Hoisting

Exercises:

1. What is hoisting?
2. Examine the following function assignment. Will it get hoisted?

```
let calculateSurfaceArea = function(h, w, l) {
  let area = 2 * (h * w) + 2 *  (h * l) + 2 * (w * l);
   return area;
};
```

Answers

1. Adding variable and function declarations to memory during compilation before they get executed is called "hoisting". All variable declarations and function declarations are scanned for and placed in memory at the start of their enclosing scope `{}`.
2. No, it will not get hoisted as assignments are not hoisted. This includes function expressions which are functions assigned to variables.

2.1.2 Var hoisting

Exercises:

1. Explain why the two console.log statements return different values?

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

2. Examine the following code and analyze what is happening in reference to hoisting:

```
console.log(furniture);

var material = 'Bamboo';
```

3. Analyze the following code what is logged to the console?

```
for(var i = 0; i <= 4; i++) {

  console.log(i);

  setTimeout(function() {

    console.log('The number being logged is ' + i);

  },1000);

}
```

Answers

1. During hoisting **var** variables are stored in memory with a starting value of `undefined`. Therefore, the `console.log` statement will return `undefined`. And during execution the **var** variable's value is overwritten by the user defined value which is `100`. Therefore the second `console.log` statement returns `100`.

2. The variable `furniture` has not been declared, therefore it cannot be referenced. So a reference error is returned:

```
ReferenceError: furniture is not defined
```

3. The `console.log` statement outside the `setTimeout()` function will log numbers from `0 - 4`. Whereas, the `console.log` statement inside the `setTimeout()` function the entire loop has already iterated through every iteration that it should. Therefore, by the time (1 second) that the first `setTimeout()` function will execute, `i` is already 5. However, there is no way to reference `i`

```
0

1

2

3

4

"The number  being logged is 5"
```

```
"The number being logged is 5"
"The number being logged is 5"
"The number  being logged is 5"
"The number  being logged is 5"
```

You can circumvent this issue by using **let** which has block scope:

```
0
1
2
3
4
```

2.1.3 Let and const hoisting

Exercises:

1. Define the Temporal Dead Zone (**TDZ**)
2. Answer the following questions related to the preceding block of code:

```
console.log(giraffe);
const giraffe = 'Masai giraffe';
{
const giraffe = 'Nubian giraffe';
}
console.log(giraffe);
```

- What will the first console.log statement return?
- After commenting out the first `console.log` statement what will the second `console.log` statement return?

Answer

1. `let` and `const` variables are said to lie within the **Temporal Dead Zone** when they're declared, but not as yet initialized in memory. These variables get hoisted, but no value is assigned to them in memory, not even `undefined`.
2. Answer the following questions related to the preceding block of code:

    - The first `console.log` statement will return a `ReferenceError` as **const** and **let** variables cannot be accessed before they're declared as they lie within the temporal dead zone (**TDZ**)
    - The second `console.log` statement will return `"Masai giraffe"`. As the **const** variable that references this string value is in global scope. Hence, it can be accessed anywhere. The other **const** variable is within block scope and cannot be accessed outside its block scope

2.1.5 Precedence for variable and function hoisting

Exercises:

1. Have a look at the code snippet and explain why the number `20` is logged to the console:

```
sum(10,10);

function sum(x,y){

let add = x + y;

console.log(add);

}
```

2. A variable sum has been declared and assigned a value of `10`. What will the `console.log` statement return now?

```
function sum(x,y){

let add = x + y;

console.log(add);

}
```

```
sum(1, 2);

var sum = 100;

console.log(typeof(sum));
```

3. Will `fruit` be hoisted? If so what will the `typeof()` operator return?

```
console.log(typeof(fruit));

var fruit = function() {

 let fruitJam = true;

 let toast = true;

 if (fruitJam && toast) {

  console.log(`I like fruit Jam and toast`);

 }

}
```

Answers

1. Functions are hoisted along with a reference to the body of the entire function. Therefore, when the function `sum()` is called before it is declared, the code statements within the function execute and `console.log` statement within the function executes.

2. Variable assignment takes precedence over function declaration. Therefore, the `typeof(sum)` will return `"number"` as `var sum` is assigned a numeric value of `100`. And the variable assignment will be hoisted before the function `sum()` .

3. No, because only function declarations are hoisted. Function assignments are not hoisted. `Undefined.`

2.2 Error handling with try/catch/throw and finally

Exercises:

1. Mrs. Shear is a grade 3 teacher. She wants to ensure that students in her class learn in groups of 3. Write a function called grouping() which takes a number as a parameter. Within the function use a `try-catch` block to throw an error if the number passed into the function `grouping()` is not divisible by 3. If it is divisible

by 3 then return the number of groupings that are formed. The `finally` block must let Mrs. Shear know that the function is over.

2. Describe 3 types of errors that are encountered while coding
3. Explain what type of error is thrown and why?

```
let num= 1

console.log(s.toUpperCase());
```

Answers

1. The function `grouping()` is below:

```
function grouping(x) {
try {
if(x % 3 !== 0) {
throw new Error(` Not divisible by 3`)
} else {
let result = x / 3
console.log(result)
}
}  catch(e)  {
console.log(e.name + e.message)
} finally {
console.log(`Function is complete`);
}
}
grouping(9);
```

2. ReferenceError, typeError, and rangeError.
3. `TypeError` as you are trying to use a string method on the number data type.

2.4 Strict mode

Exercises:

1. What is **strict mode** and why should you use it?

2. What is happening here and how can we stop it?

```
x = 6;

console.log(x);
```

3. The following block of code is not in strict mode. What all from this block of code is not allowed while in strict mode?

```
customer = {};

function customerDetails() {

customer.nom = "Shadow Zilla";

customer.id = "808";

customer.job  =  'Candy';

return (customer);

}

Object.defineProperty(customer, 'job', {

value: 'Designer',

writable: false

});

console.log(`${customer.nom} is a great
${customer.job}`);
```

Answers

1. Strict mode is an ES5 feature which enforces a semantically stricter version of JavaScript. Strict mode reduces the number of silent errors by adhering to a stricter mode of itself.
2. The assignment of `x` to `6` creates a variable in the global scope which is hoisted to the top of its scope irrespective of what line assignment takes place. Use `strict mode` to prevent this
3. The following is not allowed while in strict mode:

- Assignment to an undeclared variable
- Writing to a read only property of an object
- Deleting a property of an object