# Variables

## Variables

A named storage for storing data.

There are couple way of creating a variable

## var keyword

This is the relic of the past, back when JavaScript first came out it was the only way of declaring variables with the `var` keyword.

Variable declared with `var` is either globally scoped or function scoped.

A `var` variable defined in function can be used only the function and is functionally scope.

A `var` variable defined outside of a function is global scoped, can be used in any function.

```
var greeter = "hey hi";

function newFunction() {
  var hello = "hello";
}
```

`greeter` is globally scoped, `hello` is function scoped, hence if you try to access `hello` outside of the `newFunction` it will be a error since it is undefined.

### funky var keyword

You are able to re-declare and updated `var` variables.

```
var greeter = "hello";
var greeter = "hello world!";
```

This is perfectly valid JavaScript code.

```
var greeter = "hi";
greeter = "haha xd";
```

This is of course allowed.

### Hoisting of var

A mechanism where variables and function declarations are moved to the top of their scope before code execution.

```
console.log(greeter);
var greeter = "say hello";
```

So the code above is interpreted as if it was like this

```
var greeter;
console.log(greeter); // greeter is undefined
greeter = "say hello";
```

### Problem with var

The major problem with using `var` is that if you are going to redefine a variable that is already been defined, it will be hard to tell. To make this point more clear, here is an example:

```
var greeter = "hey hi";
var times = 4;

if (times > 3) {
  var greeter = "say Hello instead";
}

console.log(greeter) // "say Hello instead"
```

Here, you can see that if `times` is greater than 3 which it is in this case, it will redefine `greeter` to be `"say Hello instead"`. Now in this short snippet maybe you can tell by yourself that `greeter` has been redefined, but what if it is many lines down. You won't know you be overwriting the original `greeter` variable since they are so far apart.

# let keyword

The let keyword is the de facto standard for declaring a variable instead of `var`. Variable declare with `let` are block-scoped.

You can update variable that's declared as `let`, but you cannot redeclare it.

Redeclaring the same variable in different scope { }, is fine because those two `greeting` are treated as different variables since they are in different scope.

```
let greeting = "say Hi";
if (true) {
  let greeting = "say Hello instead";
  console.log(greeting); // "say Hello instead"
}
console.log(greeting); // "say Hi"
```

It also solves the problem with using `var`.

### No redeclaring

With `let` you cannot redeclare a variable like `var`.

```
var a = 5;
var a = 3; // This is fine

let a = 5;
let a = 3; // This is error
```

### No hoisting

`var` keyword will basically move the declaration of a variable even if you assigned it on the same line to the top of the program.

```
console.log(a);
var a; // undefined (not an error)
```

However, variable declared with `let` have no hoisting.

```
console.log(a);
let a; // Uncaught ReferenceError: a is not defined
```

## const keyword

Variable that's declared with `const` have const values. They are block scoped just like `let`. However, they cannot be updated or redeclared once they are declared.

```
const greeting = "Hi";
greeting = "Hello"; // error, you cannot assign to a const variable
```

In addition, they must have an initialization value.

Object declared with `const` can be updated, but cannot be reassigned.

```
const greeting = {
    message: "say Hi",
    times: 4
}

greeting.message = "Oh hi!"; // This is allowed!
```

```
const greeting = {
    message: "say Hi",
    times: 4
}

greeting = {
another: "message",
  xd: "please"
}
// This is not allowed! Reassigning is not allowed!
```

## No hoisting

Just like `let` the `const` keyword also doesn't allow hoisting.

---

Revision #4
Created 17 December 2022 00:24:33 by Tamarine
Updated 18 December 2022 16:14:51 by Tamarine