Strings and Variables

To understand primitives, we have to first understand variables.

Variable is a way to store some value which you can use later in the program.

To create a variable, we will write `let` and then the `variable name` , then `equals to` then a string in single quotes.

What this will do is, it create a binding between variable name and its value. Now, from now on this variable name can be used if you want to use the value it is pointing to.

To print this, we will use `console.log(variable_name)` `console.log` will print the value pointed by this variable.

There are some rules to write variable names.

- can not start with a number. (can have number in between)
- can not include any special character (except `$` and `_`) can
- contain any any alphabet can not give space between
- characters.
- can not use any keywords as variable names.

There is a convention for variable names for multiple words.

```
let firstname = 'Lionel'; //normal
let firstName = 'Lionel'; // camel case
let first_name = 'Lionel'; // snake case
let first-name = 'Lionel'; // kebab case (this you can not use as variable names can not have special characters)
```

In JS, there is a convention to use camel case.

What will be the output of below code.

```
let firstName = 'Lionel';
let anotherName = firstName;

console.log(anotherName); // Lionel
```

As we know `firstName` is pointing to `'Lionel'` in memory. `anotherName` will also point to `'Lionel'` now.

So the output will be `'Lionel'`

What will happen, if we add two strings?

```
let firstName = 'Lionel';

let lastName = 'Messi';

let anotherName = firstName + lastName;

console.log(anotherName); // 'LionelMessi';
```

If we add two strings, it will create a new string with one string concatenated to another.

Now, `anotherName` is pointing to `'LionelMessi'`

What if we have to add space in between?

One way is to create a new variable for space.

```
let firstName = 'Lionel';

let lastName = 'Messi';

let space = ' ';

let anotherName = firstName + space + lastName;

console.log(anotherName); // 'Lionel Messi';
```

Another way is to add string directly without creating a variable.

```
let firstName = 'Lionel';

let lastName = 'Messi';

let anotherName = firstName + ' ' + lastName;

console.log(anotherName); // 'Lionel Messi';
```

All three are strings, the only difference is the first and last string are referenced through variable and the middle is string is written as it is.

This is the use case of variables, we dont have to write `'Lionel Messi'` everywhere. We can just use `anotherName` variable.

You can not use multiplication, division and subtraction with strings, only addition which will cause concatenation.

Here we are using single quotes to write strings.

We can use double quotes also and backticks to create strings.

```
'Hello world'
"Hello world"
`Hello world`
```

They differ in a way how they handle special characters.

Backslash has some special powers inside strings. Whenever a `\` is found inside string, it indicates the character after it has some special meaning. This is called **escaping characters.**
For example, if you want to create a line break, between characters, you can use `\n`

```
let firstName = 'Lionel\nMessi';

console.log(firstName);
/*
Lionel
Messi
*/
```

If you want to use single quote inside a string with single quotes.

```
let name = 'Lionel\'s Messi';
console.log(name); // Lionel's Messi
```

Also in JS, string and character is same thing.

```
let something = 'a'; // this is a string
let anything = 'abc'; // this is also a string
```

Backticks (template strings)

Usefulness of template strings, is that we can include variables inside strings. We dont have to use `+` operator multiple times ton include variables.
The syntax is to use `${}`. This will inject variables inside string. You can inject any Javascript expression.

```
let firstName = 'Samarth';
let lastName = 'Vohra';

let state = 'Delhi';
let favLang = 'Javascript';

let greeting = 'My name is ' + firstName + ' ' + lastName + '. I live in ' + state + '. My favorite lang is ' + favLang;

let greeting2 = `My name is ${firstName} ${lastName}. I live in ${state}. My favorite lang is ${favLang}`;

console.log(greeting);
console.log(greeting2);
```

Also with template strings, it is very easy to rearrange strings.

Numbers

You can store numbers in variables, like this.

```
let num = 15;
console.log(num);
```

As you have seen, we have used `let` keyword for both strings and numbers.

```
let num = 10;
num = 20.5;
num = -60;
```

In javascript there is no different data types for float and negative numbers.

Numbers are super flexible in JS.

We can do all kind of mathematical operations on numbers, like addition, subtraction, multiplication, division and modulus (%).

Modulus is used for remainder.

There is also exponentiation operator in Javscript

```
let num = 10**3;
console.log(num); // 10^3 = 1000
```

What will be the output of the below code?

```
let num = 20 + 3 * 4;
console.log(num); // 32
```

Precedence of operators

Parenthesis have highest precedence

Then exponentiation operator.

Then multiplication and division have same precedence.

Then addition and subtraction.

If two operators have same precedence, then value is calculated from left to right.

Some things about variables

You can not redefine variables declared with `let`

```
let something = 'Hello world';

let something = 'Hello world'; // error
```

But what you can do is reassign variables.

```
let something = 'Hello world';

something = 'Other thing';

console.log(something); // Other thing
```

There is one interesting thing about Javascript, is that it is weakly typed, which means we can reassign one variable to different types.

```
let something = 'Hello world';

something = 20;

console.log(something); // 20
```

C++ is a strongly typed language. You can assign a variable of one type to another.

Some things about variables

Booleans

```
let hello = true;

console.log(hello); // true

let other = false;

console.log(other); // false
```

We can assign only two boolean values, true and false.

We can get these booleans using conditional operators also.

```
let score = 100;

let to = score === 100; // true

console.log(to);
```

`===` is an equality operator. Similarly there is `!==` operator.

These operators can be used to compare any values.

There are other comparison operators like, >, <, ≥, ≤.

Decision making

IF

Syntax for `if` statement is given below.

```
if (some condition that should evaluate to true or false) some_code_to_run
```

For example

```
if (true) console.log('It is true');

if (false) console.log('It is false');
```

You dont usually write these true and false directly, you write some condition inside it.

```
let score = 34;

if (score >= 33) console.log('Pass!!');

// you can write the above code as follows
if (score >= 33) // true
  console.log('Pass!!');
```

Similary

```
let score = 32;

if (score >= 33)
  console.log('Congratulations');
  console.log('Pass!!');
```

When you run the above code, `Pass!!` also prints.

This is because `if` statement will run only the single statement. If you want to run multiple statement for the `if` statement, you have to create a block.

Block is used to group multiple statements, where it is expecting only a single statement.

To create a block, we have to use curly braces.

```
{
  console.log('Line one');
  console.log('Line two');
}
```

The above code is a valid JS code.

So, to run multiple statements for `if`, we have to create a block.

```
let score = 34;

if (score >= 33) {
  console.log('Congrats!!');
  console.log('pass!!');
}
```

With the help of `if` statement, we are able to conditionally execute some lines.

ELSE

If you want to run something, if the condition fails, we can use `else`

```
let score = 30;

if (score >= 33) {
  console.log('Congrats!!');
  console.log('pass!!');
} else {
  console.log('Hehe, fail!!');
}
```

IF ELSE

We can also apply multiple conditions using if-else

```
let score = 85;

if (score > 90) {
  console.log('A');
} else if (score > 80) {
  console.log('B');
} else if (score > 70) {
  console.log('C');
} else {
  console.log('D');
}
```

It will run the statement belonging to the first condition which is true.

Logical Operators

&&

Both condition needs to be true.

```
let score = 90;
let attendance = 65;

if (score > 90 && attendance > 75) {
  console.log('A+');
} else {
  console.log('A');
}
```

||

Logical OR operator means, any one can be true

```
let score = 92;
let attendance = 73;

if (score > 90 || attendance > 75) {
  console.log('A+');
} else {
  console.log('A');
}
```

!

Logical not operator. Its a unary operator. It flips the value given to it.

```
let score = 21;
let isPassed = score > 33;

if (!isPassed) {
  console.log('Fail');
}
```

As soon as the final result of the logical operation is known, execution stops.

```
let score = 50;

if(score > 33 || firstName) {
```

```
  console.log('Pass!!');
}
```

In the above code `firstName` is not defined, still code workds fine.

This is because, execution stops as soon as it check score > 33 is true.

```
let score = 50;

if(false && firstName) {
  console.log('Pass!!');
} else {
  console.log('Fail!!');
}
```

Functions

Functions

Functions are the programs within a program. We can run it multiple times within a program.

The syntax for function is as follows

```
function function_name() {
  // code to run
}
```

For example, you can define a function as follows

```
function sum() {
  let num1 = 10;
  let num2 = 30;

  console.log(num1 + num2);
}
```

Function on its own will do nothing, until we call it.

To call function we have to write function name with open and closing brackets

```
function sum() {
  let num1 = 10;
  let num2 = 30;

  console.log(num1 + num2);
}

sum(); // 40
```

We can call this function as many times, as we want.

```
function sum() {
  let num1 = 10;
  let num2 = 30;

  console.log(num1 + num2);
}
```

```
sum();
sum();
sum();
sum();
```

We can pass value to functions

```
function sum(num3) {
  let num1 = 10;
  let num2 = 30;

  console.log(num1 + num2 + num3);
}

sum(10); // 50
```

We are passing `10` to the function. To access 10 inside the function, we name an argument inside the function just like we name a variable.

So, `num3` is now pointing to 10.

We can call this function multiple times with different values.

```
function sum(num3) {
  let num1 = 10;
  let num2 = 30;

  console.log(num1 + num2 + num3);
}

sum(10); // 50
sum(30); // 80
sum(90); // 130
```

The first time the function runs, `num3` will be pointing to 10, the second time, `num3` will be pointing to 30.

To return something from function, we can use the `return` keyword.

```
function sum(num2) {
  let num1 = 20;
  let ans = num1 + num2;
  return ans;
}
```

Returning value from the function

```
sum(20);
```

You can use return statement only once inside a function. You are calculating the answer and returning its value.

As of now, the above code is doing nothing.

You can store the value returned by the function in a variable.

```
function sum(num2) {
  let num1 = 20;
  let ans = num1 + num2;
  return ans;
}

let res1 = sum(20);
console.log(res1); // 40;

let res2 = sum(50);
console.log(res2); // 70
```

Challenge (grade calculator)

Create a function which takes total marks of student as an argument and return a grade for that student.

undefined and null

How do we declare a variable?

```
let someVariable = 'Lionel';

console.log(someVariable); // Lionel
```

But what if we declare a variable but do not assign it.

```
let someVariable;

console.log(someVariable); // ??
```

If we run the above code, `undefined` gets printed on the console.

`undefined` in JS is used to represent absence of a value. We did not assign `undefined` to the variable. JS automatically assign undefined to the variable, if we do not assign it.

This can be useful inside an `if` statement to check if a variable is ever been assigned a value or not.

```
let email;

if (email === undefined) {
  console.log('Email is required!!'); // this will be printed
} else {
  console.log(email);
}
```

To check for `undefined`, we can use JS inbuilt keyword `undefined`.

If we reassign a variable, then it will not be `undefined`

```
let email;

email = 'abc@gmail.com'

if (email === undefined) {
  console.log('Email is required!!');
```

```
} else {
  console.log(email); // this will be printed
}
```

Another example (undefined in argument)

```
function greeting(firstName) {
  console.log(firstName); // Lionel
}

greeting('Lionel');
```

But what if we don't send an argument to the function.

```
function greeting(firstName) {
  console.log(firstName); // undefined
}

greeting();
```

When the argument is not provided, `undefined` will be assigned to `firstName`.

Another example (undefined in return)

```
function sum(num1) {
  console.log(num1);
}

let result = sum(20);
console.log(result); // undefined
```

`result` variable is going to store the value, whatever returned from the function. But what if nothing is returned from the function. Then in that case, `result` variable will be pointing to `undefined` .

From above examples, we can see that `undefined` gets implicitly assigned by the JS, if we ourselves do not assign some value.

Sometimes, in your program you want to clear some value. For example when user clear the form or clear the input field, unselect the dropdown, etc.

In that case, we can explicitly assign `undefined` to a variable.

```
let email = 'abc@gmail.com';

// For example, user clears a form
email = undefined;

console.log(email); // undefined
```

But there is one problem with the above approach. We do not know, the variable is not defined or the variable is explicitly set undefined. And sometimes it is very important to know the difference.

So, for this JS gave us another data type called `null` which also represents `no value`.

```
let email = 'abc@gmail.com';

// For example, user clears a form
email = null;

console.log(email); // null
```

Challenge (grade calculator contd.)

Add feature to the function. When no marks is provided by the user, set grade to `E`

Functions - Multiple arguments and argument defaults

Passing multiple arguments

```
function sum(num1, num2) {
  return num1 + num2;
}

let result = sum(4, 6); // 10
```

We can provide multiple arguments using comma separated values.

`num1` will be pointing to 4 and `num2` will be pointing to 6.

Providing default value to the argument

```
function sum(num1, num2) {
  console.log(num1); // undefined
  console.log(num2); // undefined
}

sum();
```

We can provide default value to arguments using the below syntax.

```
function sum(num1, num2 = 20) {
  console.log(num1); // undefined
  console.log(num2); // 20
}

sum();
```

Now, if `num2` is not provided, its value is going to be 20. If you provide `num2`, it will take that value

```
function sum(num1 = 10, num2 = 20) {
  console.log(num1); // 50
  console.log(num2); // 20
}
```

```
sum(50);
```

`num1` will take the value provided and `num2` will take the default value.

Challenge (grade calculator)

Add default value to marks argument if it is not provided.

Objects

Objects in JS is used to represent a similar group of things. For example, in a to-do application, we have to store title , description and completion. We can store them separately. But it will be better if we store them together as they together belong to a to-do.

Similarly with note-taking app. A note contains a title and description. Instead of storing these two values separately in two strings, it will be better if we can store them together as note.

That's where objects come in. Objects can be used to store similar information in a single place.

Syntax for objects are as follows

```
let todo = {
  title: 'Buy groceries',
  completed: false,
  due: 10
}

console.log(todo)
```

Just like we define other variables with `let` keyword, we do the same with object.

Object starts with curly braces and inside that we have to write key-value pairs. Keys are called properties and values can be anything, number, string, Boolean, function or object.

You write properties just like you write variables.

Dot notation

We can access properties from todo using the dot notation.

For example, if we want to print the title of the todo

```
let todo = {
  title: 'Buy groceries',
  completed: false,
  due: 10
}
```

```
console.log(todo.title); // Buy groceries
console.log(`The title of the book is ${todo.title}`);
```

Changing properties

We can also change the property of an object.

```
let todo = {
  title: 'Buy groceries',
  completed: false,
  due: 10
}

console.log(todo.completed); // false

todo.completed = true;

console.log(todo.completed); // true
```

Challenge

Create a note object with title, description and pages properties.

Methods

Methods are nothing but object properties that are function.

As we have seen, object properties can be number, string and Boolean, but object properties can also be functions.

Syntax:

```
let marks = {
  pa: 90,
  fnd: 100,
  nalr: 0,
  totalMarks: function() {
    return 90 + 100 + 0;
  }
}

console.log(marks.totalMarks()); // 190
```

You can also pass arguments to methods

```
let marks = {
  pa: 90,
  fnd: 100,
  nalr: 0,
  totalMarks: function(fine) {
    return 90 + 100 + 0 - fine;
  }
}

console.log(marks.totalMarks(50)); // 140
```

But it will be better if we don't have to hardcode marks and use the marks on the object itself.

JS provide a special keyword called `this`. The value of `this` is the object itself on which the method is called.
You can print the value of `this` on the console.

```
let marks = {
  pa: 90,
  fnd: 100,
  nalr: 0,
  totalMarks: function(fine) {
```

```
    console.log(this);
    return 90 + 100 + 0 - fine;
  }
}

console.log(marks.totalMarks(50)); // 140
```

`this` points to the same object on which the method is present.

Now, you can access properties of the object using dot notation on `this`.

```
let marks = {
  pa: 90,
  fnd: 100,
  nalr: 0,
  totalMarks: function(fine) {
    console.log(this);
    return this.pa + this.fnd + this.nalr - fine;
  }
}

console.log(marks.totalMarks(50)); // 140
```

Arrays

Arrays are used to store collection of multiple items under a single variable name.

Syntax to declare an array

```
let arr = [2, 'Hello', false];
console.log(notes);
```

Arrays items can be of any type. They do not have to be same.

Grabbing individual items

To grab individual items, we will use the bracket notation, to grab items using its index in the array. Indexing starts from 0 just like other programming languages.

```
let arr = ['Messi', 'Ronaldo', 'Neymar'];

console.log(arr[0]); // Messi
console.log(arr[2]); // Neymar
```

If we try to acces the index that does not exist, we will not get an error, we will get `undefined`.

```
let arr = ['Messi', 'Ronaldo', 'Neymar'];

console.log(arr[10]); // undefined
```

Setting items

You can also set item at a particular index

```
let arr = ['Messi', 'Ronaldo', 'Neymar'];
arr[1] = 'Zlatan';
console.log(arr); // ['Messi', 'Zlatan', 'Neymar'];
```

Arrays

Array properties and methods

`length`

To get the number of items in an array, there is a property on array which you can access

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo.length); // 3
```

`push()`

Now, lets say, you have to add items to the array, you can use `push()` method on an array. You have to pass the item you want to add to the `push()` method. `push()` method will add item to the end of the array.

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo); // ['Buy groceries', 'Complete assignment', 'ST-1'];

todo.push('ST-2');
console.log(todo); // ['Buy groceries', 'Complete assignment', 'ST-1', 'ST-2'];
```

`pop()`

We can also remove item from the end of an array using `pop()` . `pop()` method also returns the remove item which you can store in a variable.

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo); // ['Buy groceries', 'Complete assignment', 'ST-1'];

let removedItem = todo.pop();
console.log(todo); // ['Buy groceries', 'Complete assignment']
console.log(removedItem); // 'ST-1'
```

`unshift()`

This method add items to the beginning of an array.

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo); // ['Buy groceries', 'Complete assignment', 'ST-1'];
```

```
todo.unshift('ST-2');
console.log(todo); // ['ST-2', 'Buy groceries', 'Complete assignment', 'ST-1'];
```

`shift()`

This method is used to remove item from the beginning of an array.

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo); // ['Buy groceries', 'Complete assignment', 'ST-1'];

let removedItem = todo.shift();
console.log(todo); // ['Complete assignment', 'ST-1'];
console.log(removedItem); // 'Buy groceries'
```

The method discussed above changes the original array.

`join()`

This method returns the string. by concatenating all the elements of any array.

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo.join());
// "Buy groceries,Complete assignment,ST-1"
```

By default it will use comma as a separator. You can pass different separators

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo.join('-'));
// "Buy groceries-Complete assignment-ST-1"
```

`includes()`

Return true or false depending upon whether an array includes a certain element or not.

```
let todo = ['Buy groceries', 'Complete assignment', 'ST-1'];
console.log(todo.includes('ST-1')); // true

console.log(todo.includes('ST-2')); // false
```

Loops

If you want to print from 1 to 10, you can write `console.log` for all the items. But that would be inefficient. Instead we can use loops to do so.

for loop

```
for (let i = 0; i <= 5; i = i + 1) {
  console.log(i); // 0 1 2 3 4 5
}

for (let i = 0; i <= 5; i++) {
  console.log(i); // 0 1 2 3 4 5
}
```

Looping through array.

```
let todo = ['Assignment', 'ST-1', 'Buy groceries'];

for (let i = 0; i < todo.length; i++) {
  console.log(todo[i]);
}
```

---

for…of loop

There is one more syntax which you can use to loop through array.

```
let todo = ['Assignment', 'ST-1', 'Buy groceries'];

for (let myTodo of todo) {
  console.log(myTodo);
}
```

The value of `myTodo` variable will change at every iteration.

---

for…in loop

To loop through object, we can use `for-in` loop.

```
let obj = {
  english: 80,
  maths: 90,
```

```
  hindi: 70
}

for (let key in obj) {
  console.log(key); // english | maths | hindi
  console.log(obj[key]); // 80 | 90 | 70
}
```

---

break

If you want to break out of the loop at a certain point, we can use `break` keyword

```
let arr = [90, 80, 30, 60, 10];

for (let i = 0; i < arr.length; i++) {

  if (arr[i] < 33) {
    break;
  }

  console.log(arr[i]);
}
```

As soon as `arr[i] = 33`, it breaks out of loop.

---

continue

If you want to continue the loop, without executing the remaining code, we can use `continue` keyword.

```
let arr = [90, 80, 30, 60, 10];

for (let i = 0; i < arr.length; i++) {

  if (arr[i] < 33) {
    continue;
  }

  console.log(arr[i]);
}
```

3 ways to declare a variable

We have seen one keyword `let` using which we are declaring variables.

There are two more keywords using which we can declare variables.

---

const

Just like `let` , we can declare variables using `const` keyword.

```
const email = 'abc@xyz.com';

console.log(email);
```

Difference between `let` and `const` is that, with `const` you can not reassign variables.

```
const email = 'abc@xyz.com';
email = 'cnfdhsjk'; // error
```

Although, you can reassign properties of an object.

```
const person = {
  username: 'messi',
};

person.username = 'ronaldo'; // valid

person = {
  username: 'messi'
}; // Error
```

Here, `person` variable is still pointing to the same object. You are not reassigning `person` variable.

Another difference between `let` and `const` is that with `const` you cannot just declare variables and not assign it.

In case of `let` , it is okay to declare a variable but not assign it. JS will automatically assign `undefined` to it. In case of `const` , it is compulsary to assign it a value.

```
const username; // Error

const email = 'abc@xyz.com'; // Valid
```

---

var

`var` is similar to `let`. Previously there is only one way to declare variables and that is using `var` , but JS introduced two more keywords `let` and `const` to declare variables.

```
var username = 'hello';

username = 'Sam';

console.log(username); // Sam
```

The difference between `var` and `let` is that with `var` you can redeclare variables.

```
var username = 'hello';

var username = 'Sam'; // Valid

console.log(username); // Sam
```

---

var	let	const
• ✅ redeclare variables • ✅ reassign	• ❌ cannot redeclare variables • ✅ can reassign	• ❌ cannot redeclare variables • ❌ cannot reassign variables variables variables

Execution context and Call stack

We will deep dive into JS Engine to take a look how JS Engine actually execute JS code.

Whatever code is executed in JS is executed inside execution context.

Execution context consists of two things

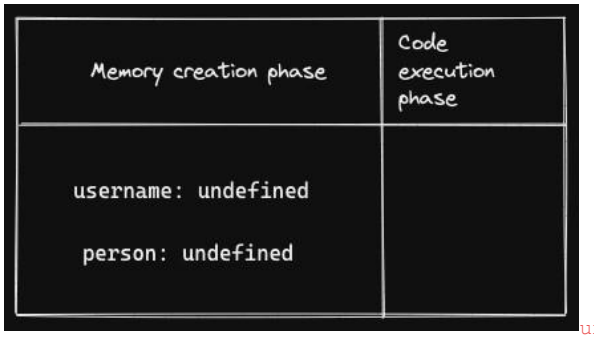
- Memory creation phase
- Code execution phase

Before executing any line of code, JS Engine will create an execution context. The first EC that is created is called Global Execution Context.

What JS engine will do is skim out all the variables and functions which are in global scope.

```
var username = 'Zeeshan';

var person = {
  email: 'abc@xyz.com',
  password: 'something-secure'
};

console.log(username);
```

For example

First, memory creation phase will run. In memory creation phase, all variables get skimmed out and are assinged 

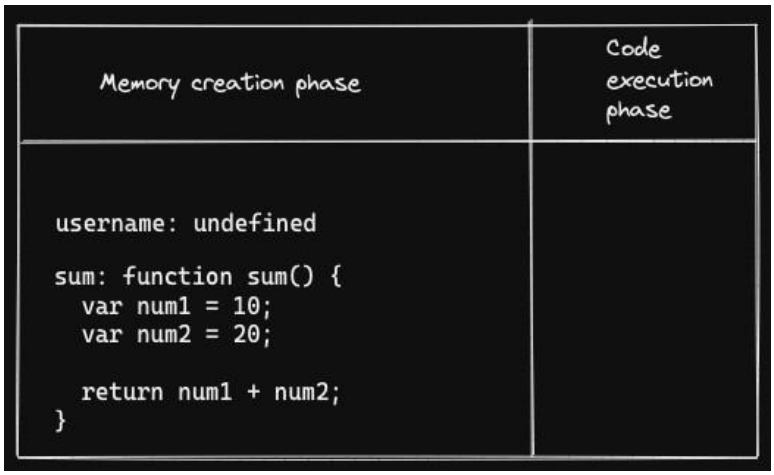
`undefined` .

After memory creation phase gets completed, code execution phase gets started. In this phase JS code will be executed line by line and `username` and `person` will be assigned respective values.

Its not the variables that are skimmed out during memory creation phase. Functions are also skimmed out. But in case of functions, `undefined` is not assigned, but the whole function gets stored there.

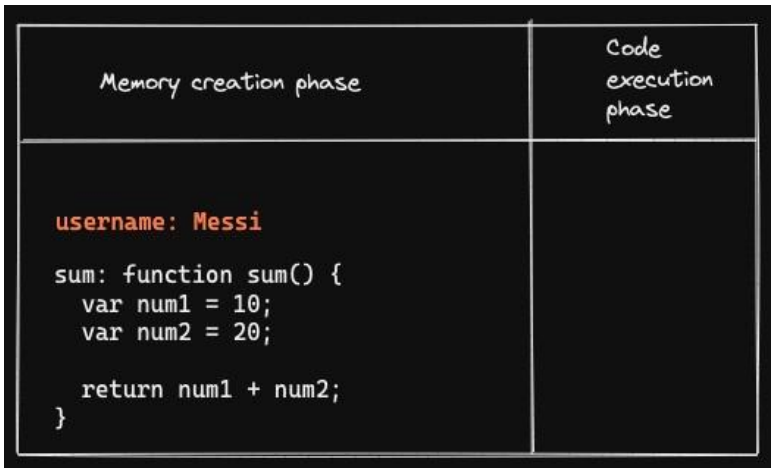
```
var username = 'Messi';

function sum() {
  var num1 = 10;
  var num2 = 20;

  return num1 + num2;
}

sum();

console.log(username);
```

As we have discussed, during memory creation phase, variables and functions are skimmed out.



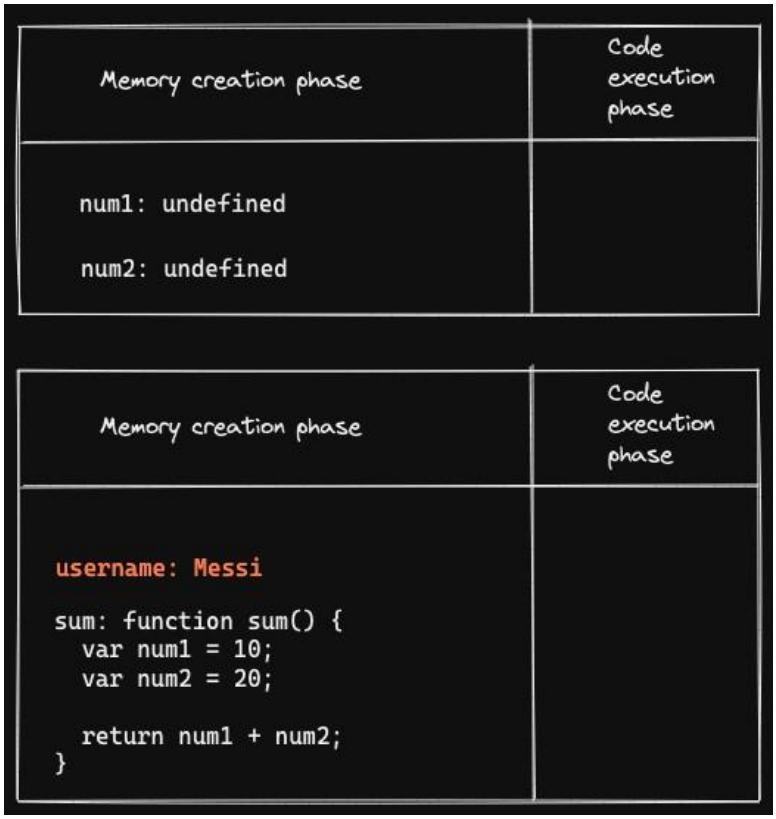
Now, code execution phase will run.

When first line is executed, `Messi` will be assigned to username.



Then there is function definition, so nothing will be executed.

Then there is a function call. Whenever a function is called, a new execution context is created for that function. And it will go through its own memory creation and code execution phase.



Then the code execution phase for this execution context will run, which will assign 10 and 20 to respective variables.

Then the `return` statement is executed. Whenever a function return something or its execution ends, its execution context gets destroyed.

And when all of the code gets executed, Global execution context also gets destroyed.

Call stack

These execution contexts are managed inside a stack called **Call Stack.**

Execution contexts are pushed and poped from this call stack.

Hoisting

What will be the output of the below code?

```
var a = 20;

function myFunc() {
  console.log('Inside my function');
}

console.log(a); // 20
myFunc(); // Inside my function
```

Now, what will be the output of below code?

```
console.log(a); // undefined
myFunc(); // Inside my function

var a = 20;

function myFunc() {
  console.log('Inside my function');
}
```

The above code will not produce error.

As we have discussed in Execution context, in memory creation phase, all global variables are assigned undefined and all functions are assigned whole function code before any code is executed.

When the first line of code runs, it will find `a` in global scope whose value is undefined.

`myFunc()` variable is assigned whole function, so it will be executed

---

Question

```
function a() {
  console.log(b); // undefined
  var b = 20;
}

a();
```

Scopes

What will be the output of below code?

```
var b = 20;

function a() {
  console.log(b); // 20
}

a();
```

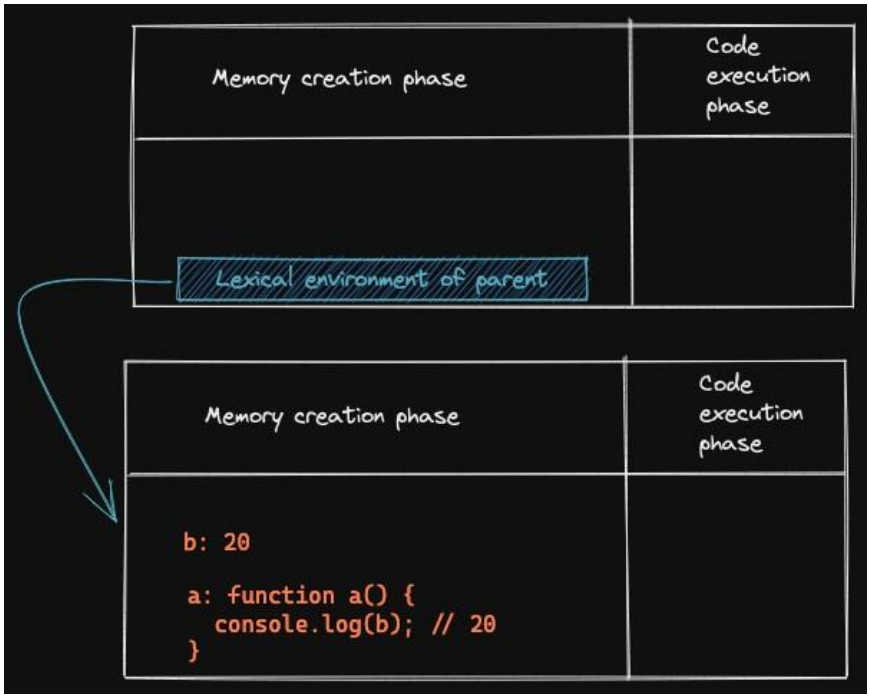
How can a function access a variable `b`

There is a concept called `scopes.` This concept is present in almost all programming languages.

Scope for a variable roughly can be defined, where that variable can be accessed.

For example, in the above case, variable `b` has global scope. It can be accessed any where in the code.

When the function execution context is created, it will try to find `b` in its local scope (or memory). If it does not find it there, it will try to find it in the lexical scope (or memory or environment) of its parent.

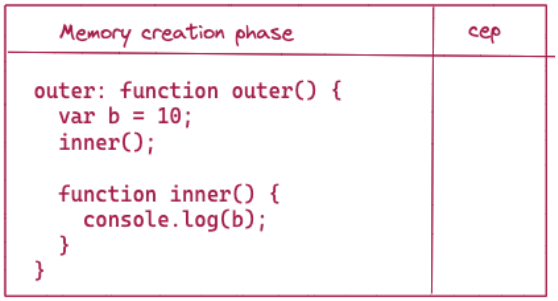


Another example

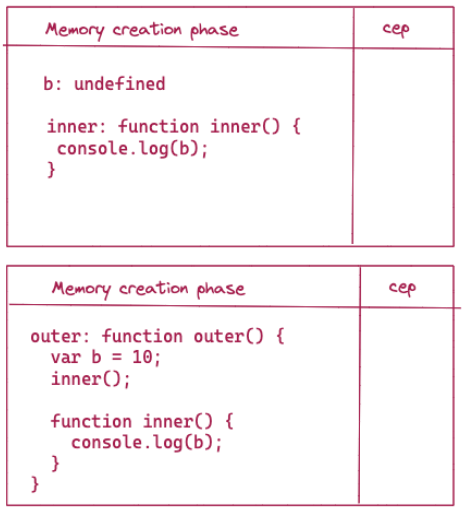
```
function outer() {
  var b = 10;
  inner();

  function inner() {
    console.log(b); // 10
  }
}

outer();
```

Lets see how this function runs

First the memory creation phase will run for global execution phase



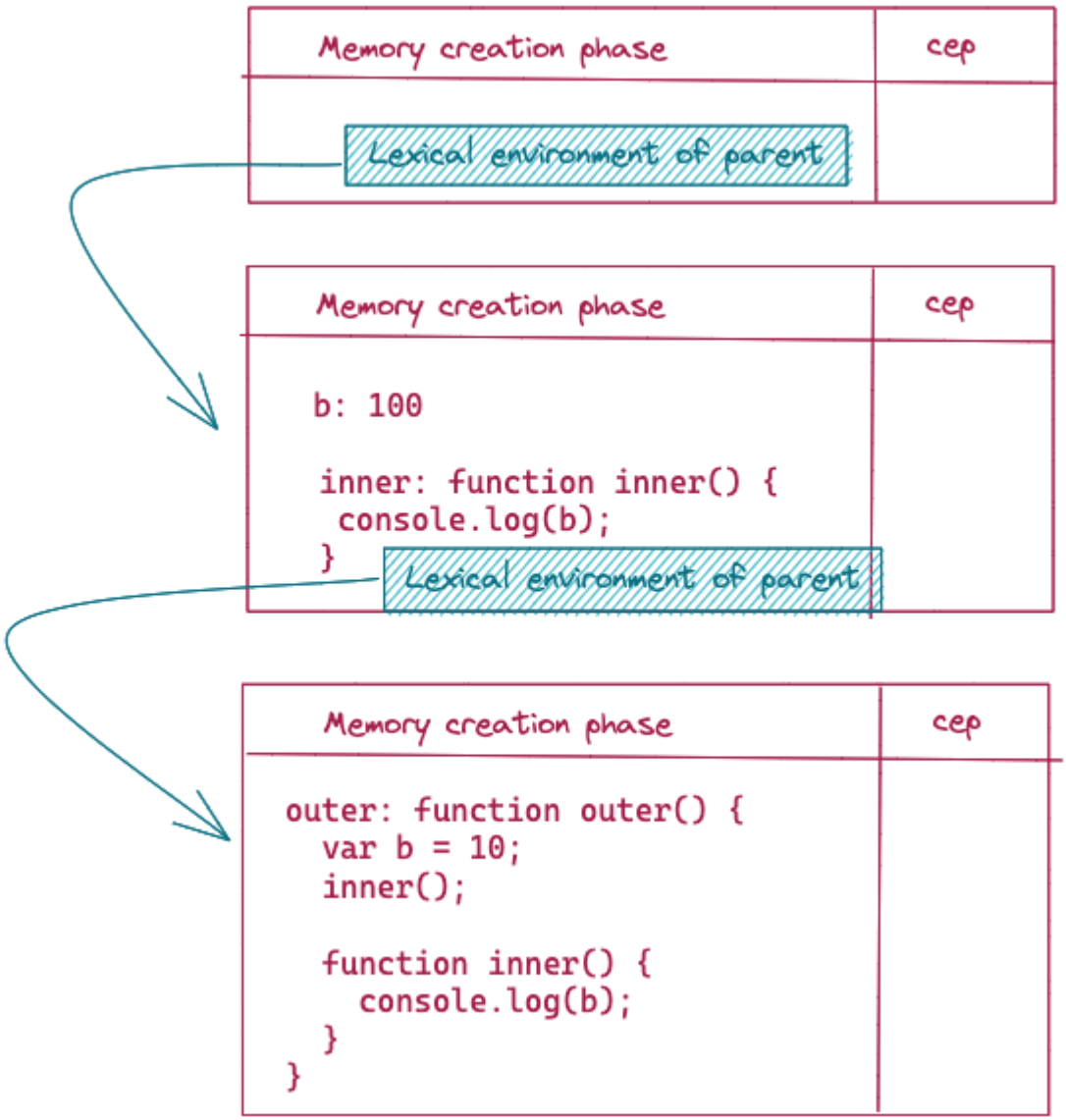
Then the `outer()` function will be called. To run that function, an execution context is created and its memory creation phase is run.



Then the outer function will start exeuting one by one.

First `b` will be assigned 100

Then `inner` function will be called. The execution context for `inner` function will be created.



When the `inner` function is executed line by line, it will look for variable `b` in its local scope. When it does not find it there, it will look it in the lexical environment of its parent. This chain of lexical environment is called **Scope Chaining**

What will happen if we call `inner` before assigning b

```
function outer() {
  inner();
  var b = 10;
```

```
  function inner() {
    console.log(b); // undefined
  }
}

outer();
```

More differences between var, let and const

Blocks

To understand this concept, we must know what is a `block` in javascript. You can create a block using curly braces.

The above code is a valid JS code. We have used blocks in `if` statement when we have to use multiple statements inside `if`.

```
{
  console.log('Hello world');
}

console.log('Hello India');
```

Scope

`let` and `const` are block scoped while `var` is a function scoped. That means the variables declared using

`var` is not block scoped.

```
let score = 50;

if (score > 33) {
  var x = 20;
}
```

```
let score = 50;

if (score > 33) {
  let x = 20;
}

console.log(x); // Error x is not defined
```

Variable `x` is only accessible inside the `if` statement. As soon as the block ends, the variable x gets vanished. Outside the `if` block, there is no variable `x`, so you will get an error.
`let` is only accessible inside the block it is declared.

```
console.log(x); // 20
```

Question

What will be the output of below code?

```
let x = 10;

{
  let x = 20;
  console.log(x); // 20
}

console.log(x); // 10
```

As `let` is block scoped, it will not conflict with that in global scope.

Question

What will be the output of below code?

```
console.log(a); // undefined
console.log(b); // Cannot access b before initialization
console.log(c); // Cannot access c before initialization

var a = 10;
let b = 20;
const c = 30;
```

```
var x = 10;

{
  var x = 20;
  console.log(x); // 20
}

console.log(x); // 20
```

Hoisting Question

`let` and `const` are hoisted differently as compared to `var` .

`let` and `const` are hoisted but they remain in **temporal dead zone.** You can not access those variables declared with `let` and `const` until they are in temporal dead zone. They remain in temporal dead zone till they are initialized.

Take a look at the error. Error is not that it is not defined. The error is you can not access them before initialization.

Higher order functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called **higher order functions**.

Passing function to another function

```
function a(fn) {
  console.log('Inside a');
  fn();
}

function b() {
  console.log('Inside b');
}

a(b);
// Inside a
// Inside b
```

You don’t have to declare a new function to pass to another function. You can declare inside an argument directly like below.

```
function a(fn) {
  console.log('Inside a');
  fn();
}

// function b() {
//   console.log('Inside b');
// }

a(function() {
  console.log('Inside b');
});
// Inside a
// Inside b
```

```
function a() {
  function b() {
    console.log('inside b')
  }
```

Returning function from another function

```
  console.log('inside a');
  return b;
}

let temp = a(); // inside a
temp(); // inside b
```

Some real-world HOF

Lets say you have an array with numbers, strings and booleans. You have to write functions that will return all the strings present in the array, then all the numbers and then all the booleans.

```
function getString(arr) {
  let result = [];

  for (let item of arr) {
    if (typeof item === 'string') {
      result.push(item);
    }
  }

  return result;
}

function getNumber(arr) {
  let result = [];

  for (let item of arr) {
    if (typeof item === 'number') {
      result.push(item);
    }
  }

  return result;
}



function getBoolean(arr) {
  let result = [];

  for (let item of arr) {
    if (typeof item === 'boolean') {
      result.push(item);
    }
  }

  return result;
}

let arr = [120, 'Hello', 90, false, 'World', true, 20, 80, 'Messi'];
```

```
console.log(getString(arr));
console.log(getNumber(arr));
console.log(getBoolean(arr));
```

As you can see, you are repeating a lot of code. You can extract out the logic of checking item and pushing it into the array as it is same for all the 3 functions. The changing part is the condition. We can pass a function to check for 3 conditions.

```
function getString(item) {
  return typeof item === 'string';
}
function getNumber(item) {
  return typeof item === 'number';
}
function getBool(item) {
  return typeof item === 'boolean';
}

function get(arr, fn) {
  let result = [];

  for (let item of arr) {
    if (fn(item)) {
      result.push(item);
    }
  }

  return result;
}

let arr = [120, 'Hello', 90, false, 'World', true, 20, 80, 'Messi'];

console.log(get(arr, getString));
console.log(get(arr, getNumber));
console.log(get(arr, getBool));
```

We are passing function whose job is check for different type into a function whose job is to check and push items into the array. Separation of concern.

Some more array methods

forEach()

`forEach` method takes callback function as an argument which is called for each item of an array. You are not calling this function yourself. `forEach` is calling your function for each item. You are just passing it as an argument.

You can create variable for a function and then pass or you can directly declare a function inside an argument.

The callback function you pass, receive array item as an argument.

```
let players = ['Messi', 'Ronaldo', 'Neymar', 'Zlatan'];

players.forEach(function(item) {
  console.log(item);
})

/*
Messi
Ronaldo
Neymar
Zlatan
*/
```

The callback function also receive index as a second argument.

```
let players = ['Messi', 'Ronaldo', 'Neymar', 'Zlatan'];

players.forEach(function(item, index) {
  console.log(`${item} at index ${index}`);
})

/*
Messi at index 0
Ronaldo at index 1
Neymar at index 2
Zlatan at index 3
*/
```

---

map()

This method also takes callback function as an argument, but return a new array populated by the result of calling that callback function. The function you pass in as a

callback will be called for each item in an array and whatever you return for that item will become the new item in a new array.

```
let marksYouDeserve = [9, 8, 7, 8];

let marksYouGet = marksYouDeserve.map(function(item) {
  return item + 30;
})

console.log(marksYouGet); // [39, 38, 37, 38]
console.log(marksYouDeserve); // [9, 8, 7, 8]
```

---

filter()

This method also takes callback as a function and return a new array. The callback function will be called for each item.

If you return true, then that item will be included in the new array otherwise it will not be included.

```
let marks = [1, 2, 3, 4, 5, 6, 7, 8];

let evenMarks = marks.filter(function(item) {
  if (item % 2 === 0) return true;
  else return false;
})

console.log(evenMarks); // [2, 4, 6, 8]
console.log(marks); // [1, 2, 3, 4, 5, 6, 7, 8]
```

There are many other methods, you can read them on MDN docs.

Closures

Consider the below function

```
function someFunc() {
  let username = 'Samarth';

  function printName() {
    console.log(username);
  }

  printName();
}

someFunc(); // Samarth
```

What will be the output of below function? As already discussed, inner function will have access to the lexical environment of its parent.

So function `printName` will have access to `username` variable which out of its scope but is in the lexical scope of its parent.

Now, what if, instead of calling `printName` function inside `someFunc` , we return `printName` function and call it outside the scope of `someFunc` , like below. What will be the output?

```
function someFunc() {
  let username = 'Samarth';

  function printName() {
    console.log(username);
  }

  return printName;
}

let fn = someFunc();
fn(); // Samarth
```

It will print `Lionel` , even though the `username` variable is not in its scope.

How, is it working? How `printName` function has access to `username` variable even though it is not in its scope?

This is a `closure` → A function bundled together with references to its surrounding state or we can say lexical environment is called **closure**.

When `printName` function was defined, it has access to `username` variable. So, it will always have access to `username` variable even though the variable is not in its scope or its parent scope.

Use of closure

There are many uses of closures, one of them is

Emulating private method

Before classes, JS does not have any way to declare private properties or methods, but with closures, you can emulate private methods.

Lets say you want to build a counter. But you dont want user to update count directly but give them some methods like `increment` , `decrement` to change the value of count.

You can do is easily in other languages with classes and private property `count` . In JS, you can use closure.

```
function counter() {

  let count = 0;

  return {
    getCount: function() {
      return count;
    }
  };
}

let counter1 = counter();
console.log(counter1.getCount());
```

We are creating a function `counter` which is going to return an object though which we can change the value of `count` variable using closures. As you can see, end user will only have access to the object with `getCount` method. User have no access to `count` variable. So they can't manipulate it directly.

You can provide different methods to manipulate count for end user.

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

  return {
    getCount: function() {
      return count;
    },
```

```
    increment: function() {
      count += 1;
    },

    decrement: function() {
      count -= 1;
    },

    reset: function() {
      count = 0;
    }
  };
}

let counter1 = counter();
console.log(counter1.getCount()); // 0
counter1.increment();
counter1.increment();
counter1.increment();
counter1.decrement();
console.log(counter1.getCount()); // 2
counter1.reset();
console.log(counter1.getCount()); // 0
```

Prototypes

Consider the below example

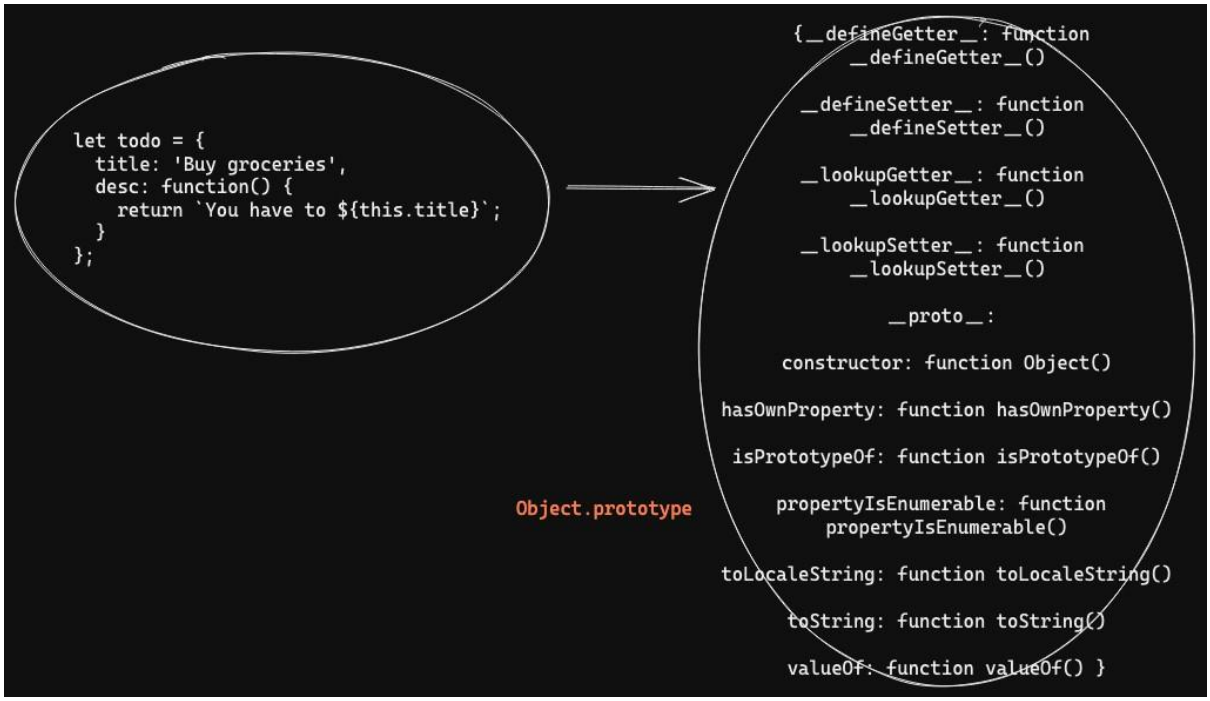
We accessed `title` property and `desc` method on `todo` object.

```
let todo = {
  title: 'Buy groceries',
  desc: function() {
    return `You have to ${this.title}`;
  }
};

console.log(todo.title); // Buy groceries
console.log(todo.desc()); // You have to Buy groceries

console.log(todo.toString()); // [object Object]
```

We also accessed `toString` method on `todo` object. But this method does not exist on todo object. We should be getting `undefined` , but instead we are getting some value, which indicates that this method do exist on the `todo` object. How is it possible?

There is a thing in Javascript called `Prototypes` .

If you try to access a property of an object, what JS will do is, it will try to find that property in the object. If it fails to find that property, then it will search its prototype for the property. Prototype is another object which is used as a fallback source of properties.



When you try to access `toString` property on an object, it will first search the object for that property. It didn't find that property. Then it will search the prototype of that property which is `Object.prototype`. It will find that property and displays the result.

So, prototype is just another object which is used as a fallback source for properties.

Every object in Javascript has a prototype.

You can check the prototype of any object using `__proto__` property.

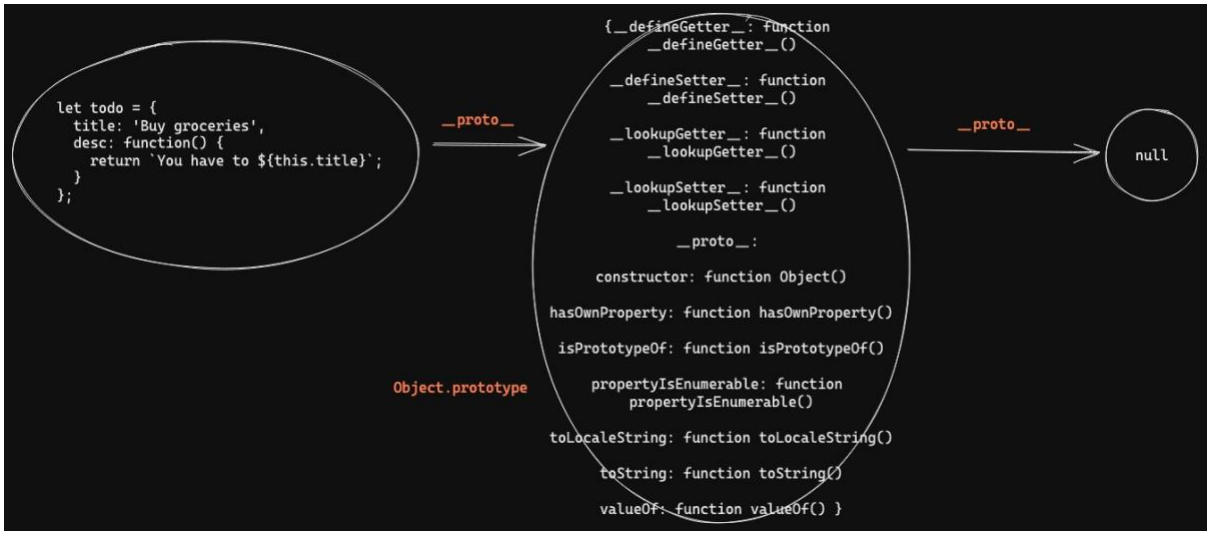
So,

```
todo.__proto__ === Object.prototype
```

As, we have seen `Object.prototype` is an object, so it must have its own prototype. What is the prototype of `Object.prototype`. It is `null`.

```
todo.__proto__.__proto__ === null
```

This chain of prototypes is called `Prototypal chain`. As there must be an end to this chain, the prototype of `Object.prototype` is `null`.



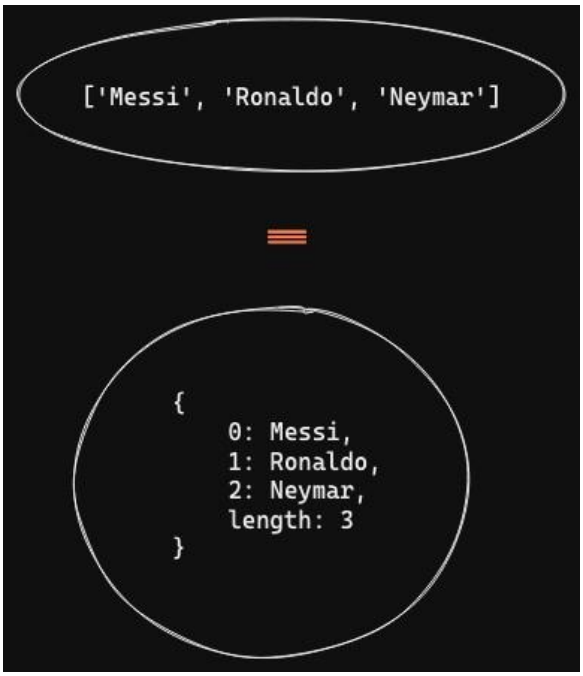
Every object by default has a prototype which is `Object.prototype`.

If you create another object, that too has `Object.prototype` as its prototype.

Accessing properties on Arrays

We have discussed few properties and methods on arrays, like `push` and `pop` . But how are we able to access these properties and methods on an array. We have seen that only objects have properties and methods not arrays.

What JS will do behind the scenes, is convert this array to an object.

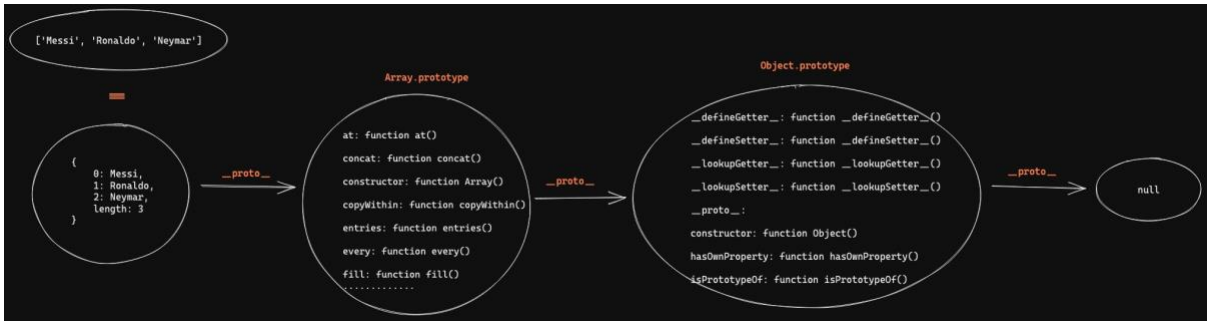


So, arrays are basically an object, on which we can access properties and methods.

For example, we can access `length` property.

We have seen the object to which an array is converted to. But that object do not have `push` and `pop` properties. Where do those properties come from.

These properties come from its prototype. As we have discussed that when we try to access the property on an array, it is converted to an object. So that object has a prototype which has all these properties and methods, `push` , `pop` , `shift` , `unshift` , etc.



This is a prototypal chain.

When we try to access property on an array, it will first convert array to an object. If it does not find property there, it will search `Array.prototype` , then it will search `Object.prototype` .

Accessing properties on strings

When we were discussing strings, we used some properties and methods like `toUpperCase` , `toLowerCase` , `trim` , etc. How are we accessing these properties and methods on strings?

Just like arrays, strings are converted to objects when you try to access properties on that string.

Inheritance

Prototypes can be considered as inheritance, as it can be looked as objects are inheriting the properties of `Object.prototype` .

Constructor functions

What will be the output of below code?

```
function user() {

}

const user1 = user();

console.log(user1); // undefined
```

We have already discussed that, if you don't return anything from the function then `undefined` is returned.

Now, what will be the output of below code.

```
function user() {

}

const user1 = new user();

console.log(user1); // {}
```

An empty object is returned, if we use `new` keyword before function call.

We can generate objects using functions using the `new` keyword infront of function call.

These functions are called **Constructor functions.**

We can add properties to the object that is being generated using the `this` keyword.

```
function user() {
  this.username = 'Samarth';
  this.email = 'sam@xyz.com';
}
const user1 = new user();
console.log(user1); // { username: 'Samarth', email: 'sam@xyz.com' }
```

We can pass custom values for username and email.

```
function user(username, email) {
  this.username = username;
  this.email = email;
}
let user1 = new user('messi', 'messi@xyz.com');
console.log(user1);
let user2 = new user('ronaldo', 'ronaldo@xyz.com');
console.log(user2);
```

Constructor functions are basically used as a blueprint to generate objects of same type with same properties.

If there are no constructor functions, we have to hardcode objects with the same properties on every object.

Lets add methods to the generated object.

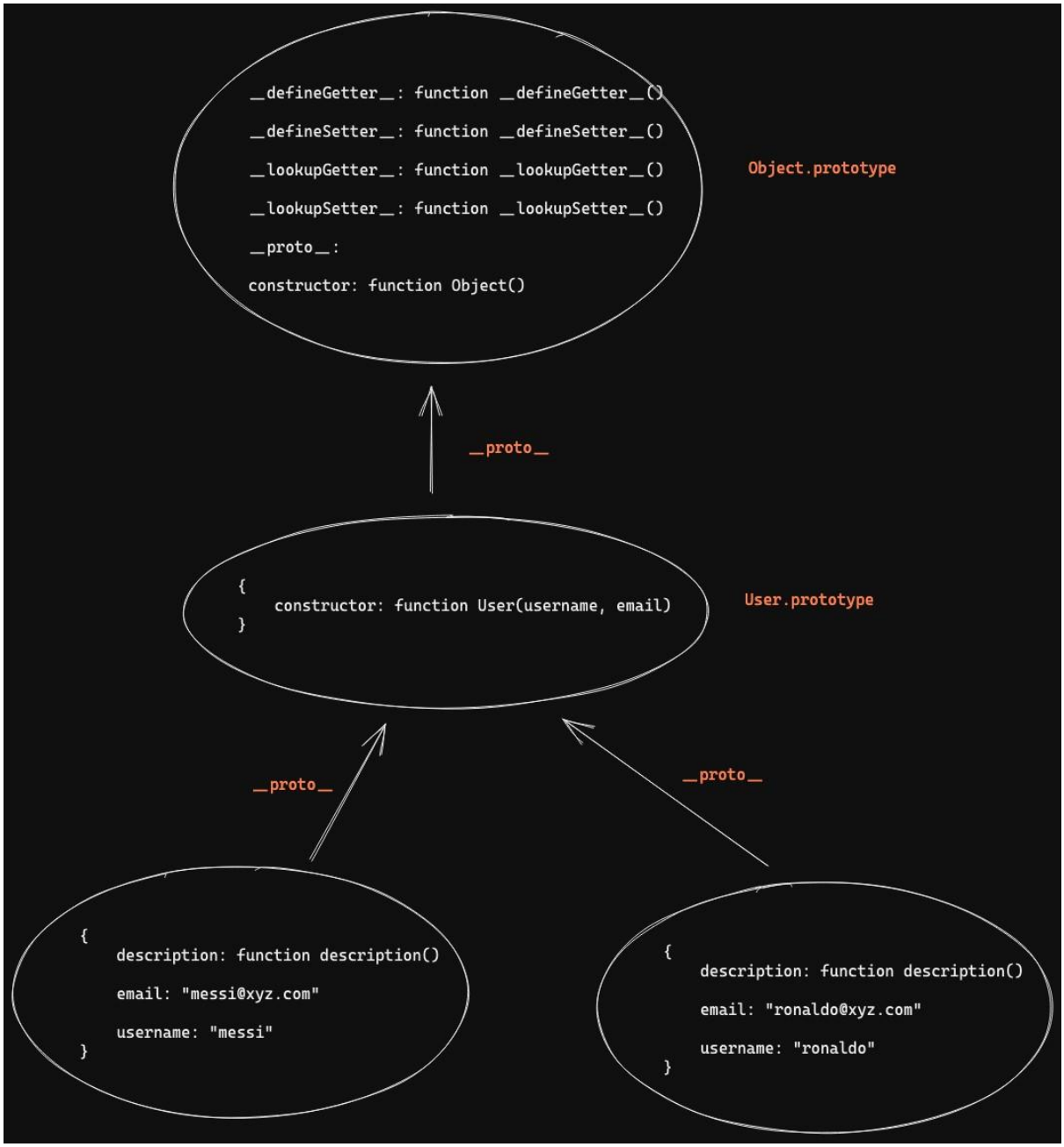
```
function user(username, email) {
  this.username = username;
  this.email = email;

  this.description = function() {
    return `My name is ${this.username}`;
  }
}
let user1 = new user('messi', 'messi@xyz.com');
console.log(user1.description()); // My name is messi
let user2 = new user('ronaldo', 'ronaldo@xyz.com');
console.log(user2.description()); // My name is ronaldo
```

You can add as many properties you want and all the generated objects will have the same properties.

There is a convention in JS to capitalize the name of constructor functions. It is just a convention. It will still work if you don’t capitalize.

```
function User(username, email) {
  this.username = username;
  this.email = email;

  this.description = function() {
    return `My name is ${username}`;
  }
}
let user1 = new User('messi', 'messi@xyz.com');
console.log(user1.description()); // My name is messi
```

As we have seen, if we generate objects using curly brace notation, its prototype will be `Object.prototype`. What will be the prototype of the objects generated from constructor functions. Its prototype will be an object with one property called `constructor` whose value is the constructor function itself.

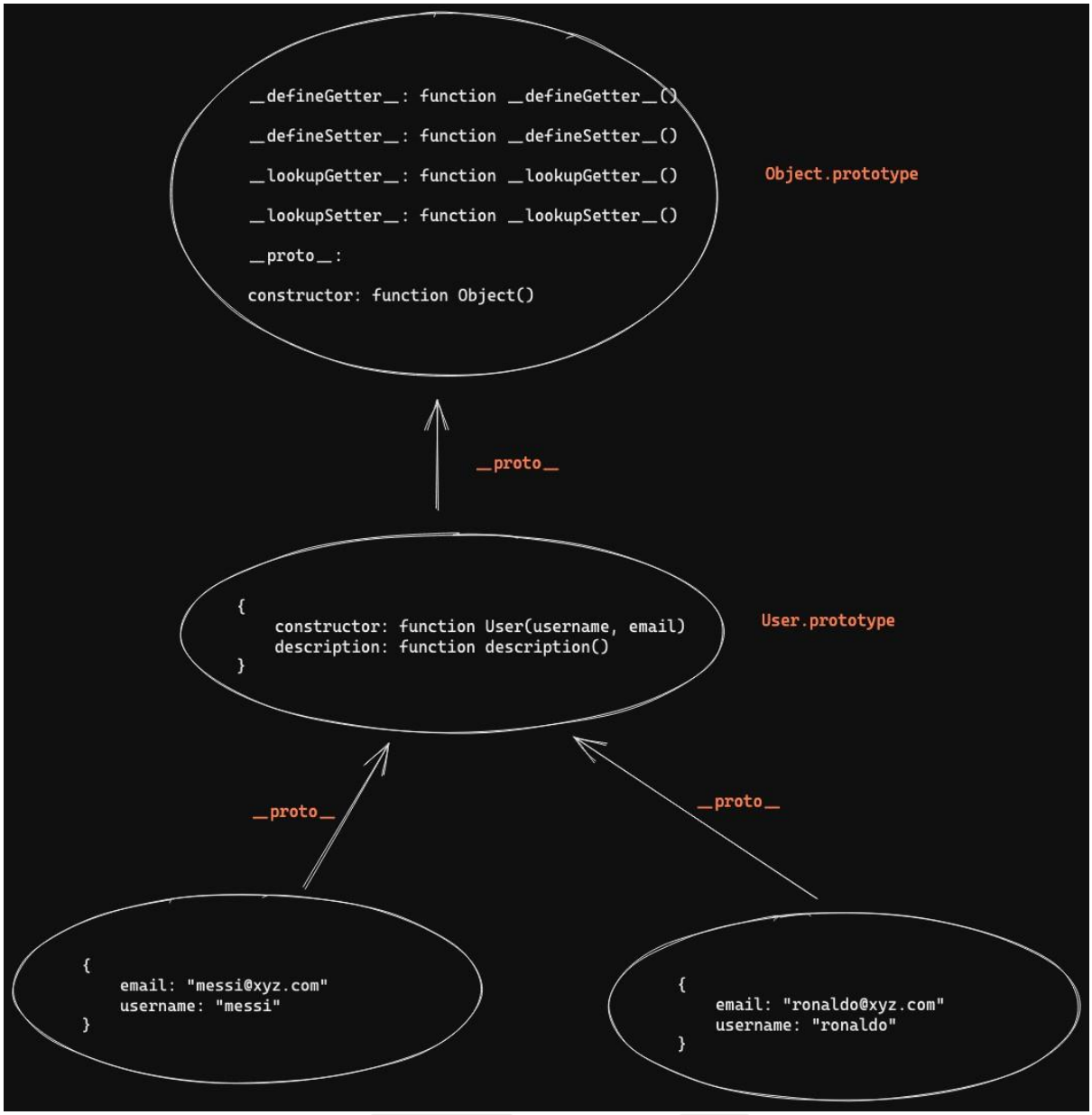


You have created your own custom data type. Just like there are Array datatype, String datatype., now there is User datatype. You can create new objects of that datatype using `new` keyword.

Now, as you have seen the `description` method is same for all objects. It will be better that we move that description method to `User.prototype.`

```
function User(username, email) {
  this.username = username;
  this.email = email;
}

User.prototype.description = function() {
  return `My name is ${this.username}`;
}

let user1 = new User('messi', 'messi@xyz.com');
console.log(user1.description());
let user2 = new User('ronaldo', 'ronaldo@xyz.com');
console.log(user2.description());
```

Now, the prototype chain will look like below



Now, when we try to access `description` property on `user1` object, it didn't exist. So it will check its prototype for that method.

Class syntax

Just like `let` and `const` , classes are very new to Javascript. It is just an alternative way to write constructor functions. It is just the Syntactic sugar. It is doing the exact same thing as constructor function, its just the sytax is different.

Consider the below constructor function. We have to convert this into Class syntax.

```
function Person(firstName, lastName, email) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.email = email;
}

Person.prototype.printBio = function() {
  console.log(`My username is ${this.username}`);
}

Person.prototype.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
}

const person1 = new Person('Lionel', 'Messi', 'abc@gmail.com');
console.log(person1);
```

The syntax for class, is we have to start witl `class` keyword and then name of that class.

You can literally copy paste the constructor function.

```
class Person {
  constructor(firstName, lastName, email) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }
}

const person1 = new Person('Lionel', 'Messi', 'abc@gmail.com');
console.log(person1);
```

You can add methods too. In this case too, you just have to copy paste. Its just the syntax is different, functionality is same.

```
class Person {
  constructor(firstName, lastName, email) {
```

```
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}

const person1 = new Person('Lionel', 'Messi', 'abc@gmail.com');
console.log(person1.getFullName());
```

We are getting the exact same behavior as we got with constructor function.

Inheritance

Lets say we want to create a `Student` class which has all the properties and methods of `Person` class. One way is to define a new class and add all properties and methods again on that class.

Other way is inheritance.

Lets say we want to create a class called `Student` with all the properties of `Person` class. One way is to copy all the properties and methods from `Peron` class.

Other method is Inheritance.

We can create a class that inherits properties from other class. Just like what we did with prototype inheritance.

```
class Person {
  constructor(firstName, lastName, email) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}

class Student extends Person {
}

const student1 = new Student('Lionel', 'Messsi', 'abc@gmail.com');
console.log(student1.getFullName());
```

We can inherit all properties and methods of one class using `extends` keyword.

Now `Student` class has all the properties (`firstName` , `lastName` and `email`) and all the methods (`constructor` and `getFullName`) of parent class `Person` .

You can override some of the methods.

For example, if you want to add some additional fields including already existed fields to Student class like `groupNo` . You can override the constructor function because that's where you are initializing properties

```
class Person {
    constructor(firstName, lastName, email) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.email = email;
    }

    getFullName() {
      return `${this.firstName} ${this.lastName}`;
    }
  }

  class Student extends Person {
    constructor(firstName, lastName, email, groupNo) {
        super(firstName, lastName, email);
        this.groupNo = groupNo;
    }
  }

  const student1 = new Student('Lionel', 'Messi', 'abc@gmail.com', 13);
  console.log(student1);
```

Now, the objects created using `Student` class will have `groupNo` property.

Similarly, you have access to `getFullName` methods from parent class.

You can use the same method or you can override it.

```
console.log(student1.getFullName()); // Lionel Messi
```

As there is no method named `getFullName` , it will check its parent.

You can also override this property just like constructor.

```
class Person {
    constructor(firstName, lastName, email) {
      this.firstName = firstName;
      this.lastName = lastName;
```

```
    this.email = email;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}

class Student extends Person {
  constructor(firstName, lastName, email, groupNo) {
      super(firstName, lastName, email);
      this.groupNo = groupNo;
  }

  getFullName() {
      return `My name is ${this.firstName} ${this.lastName}`;
  }
}

const student1 = new Student('Samarth', 'Vohra', 'abc@gmail.com', 13);
console.log(student1.getFullName()); // My name is Samarth Vohra
```

Just like you have defined new properties on `Student` class, you can also define new methods on it.

Async programming

Javascript is a synchronous single threaded language.

Single threaded means that Javascript engine can execute only one statement at a time. It can not run multiple statements. It has a single call stack to execute the statement. It does not have multiple call stacks to run statements in parallel.

Synchronous means that it will execute statements in order.
Basically, JS Engine will wait for nothing, it will keep on executing single statements line by line.

But what if we have to wait for some time and then run some code. How you are going to achieve that in JS.

Lets say you want to run the seconds log statement after 4 seconds.

```
console.log('start');

console.log('run after 4 seconds');

console.log('end');
```

There is something called `setTimeout` which is being provided by the browser which will help you to run some code after some time.

```
console.log('start');

setTimeout(function() {
  console.log('run after 4 seconds');
}, 4000);

console.log('end');
```

In `setTimeout`, you have to pass a callback function as first argument and then time in milliseconds as a second argument. The callback function will run after 4 seconds.

What will happen, first `start` will print, then `end` will print instantly then after 4 seconds `run after 4 seconds` will print.

But how is it possible? We just discussed that JS will wait for none. And it will execute statements in single order. So, how JS is running `console.log` after 4

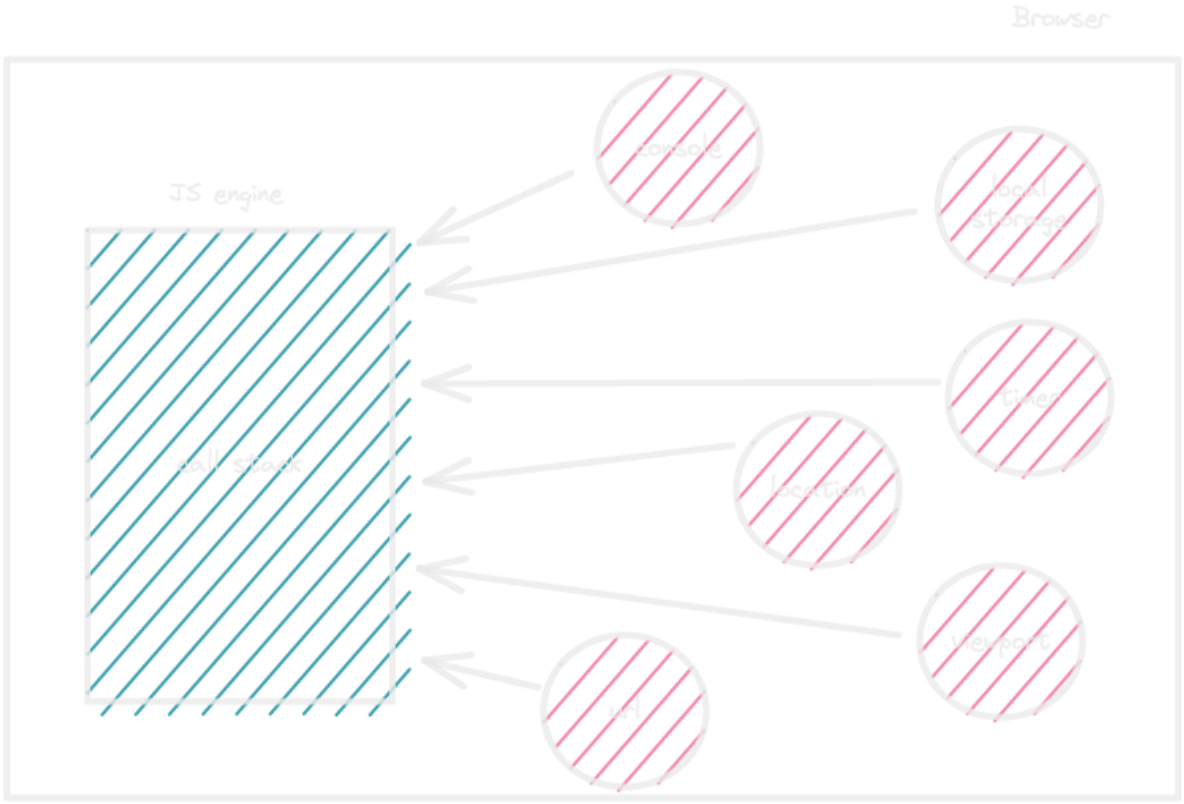
seconds.

`setTimeout` is not a part of JS. Javascript Engine do not have a timer to time for 4seconds and then run the code. It is the browser that has timer and provide us this `setTimeout` function.

Browser has many other useful things like console, local storage, session storage, fetch, viewport, location and many other things.

Browser provide these things to the javascript engine. Javscript as a language has no local storage, timer, location and other things.

Browser provide these things to the JS engine.



But how browser is providing these things to Javascript?

Browser provide these things using Web APIs. Basically, browser provide some objects, some functions to JS Engine which we can use to access these functionalities.

`setTimeout` is one such function which helps us to access timer in the browser through Javascript.

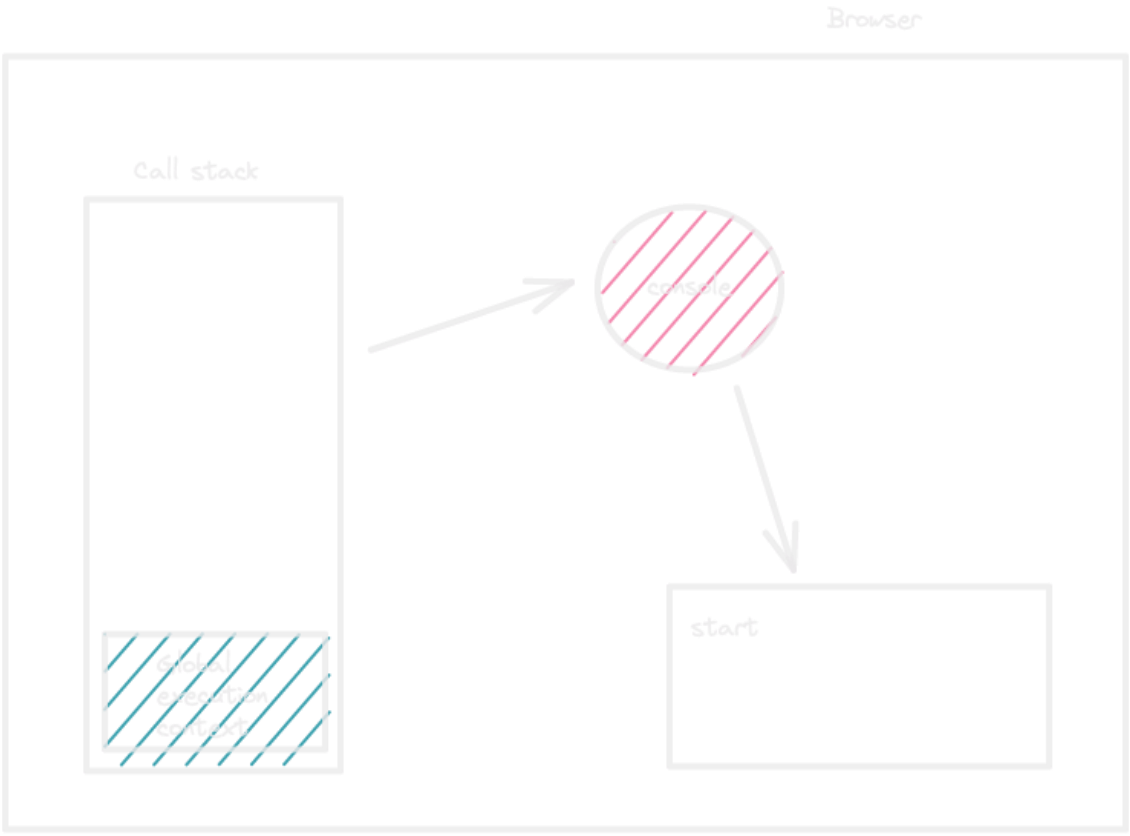
Lets see how does these things work behind the scenes.

```
console.log('start');

setTimeout(function() {
  console.log('run after 4 seconds');
}, 4000);

console.log('end');
```

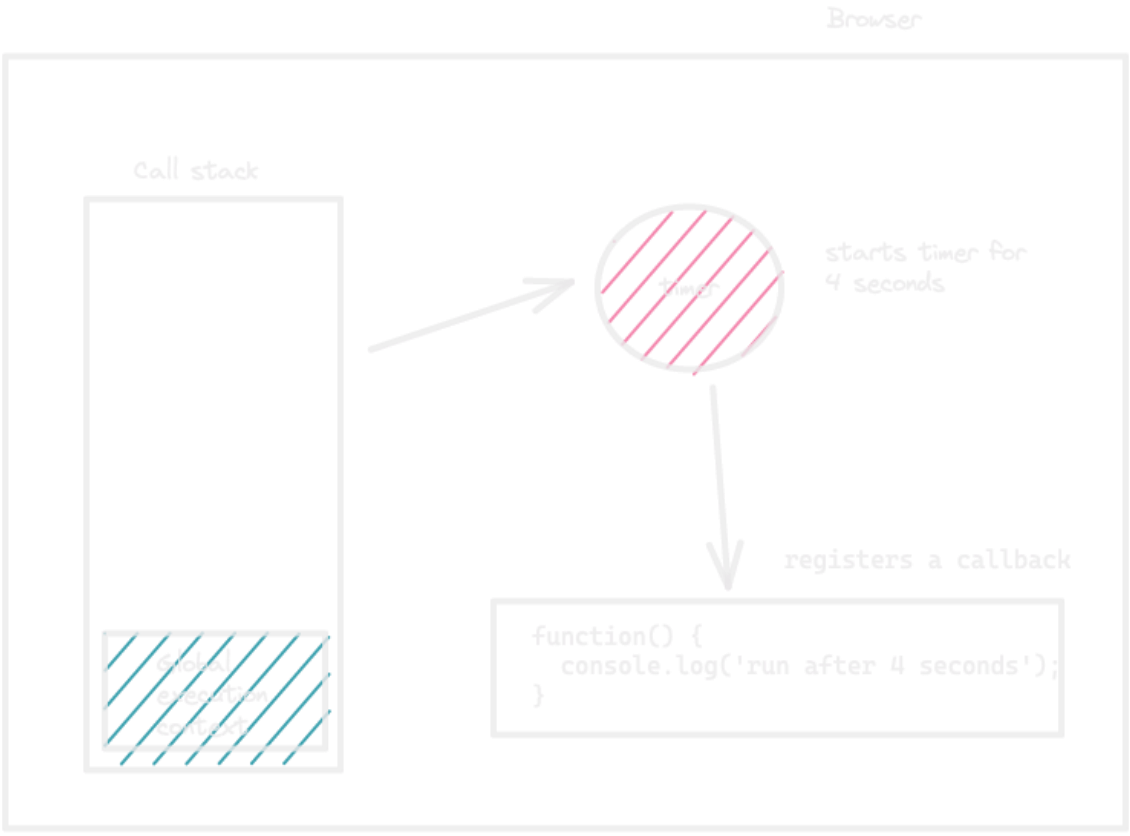
As we have already discussed, before running any code, an execution context is created and pushed inside a call stack.

As there are no variables, so there is nothing to do in memory creation phase. Then code execution phase will start. Each line will be executed line by line.

When first line is executed, JS engine will access the console from the browser and print `start` on the console.

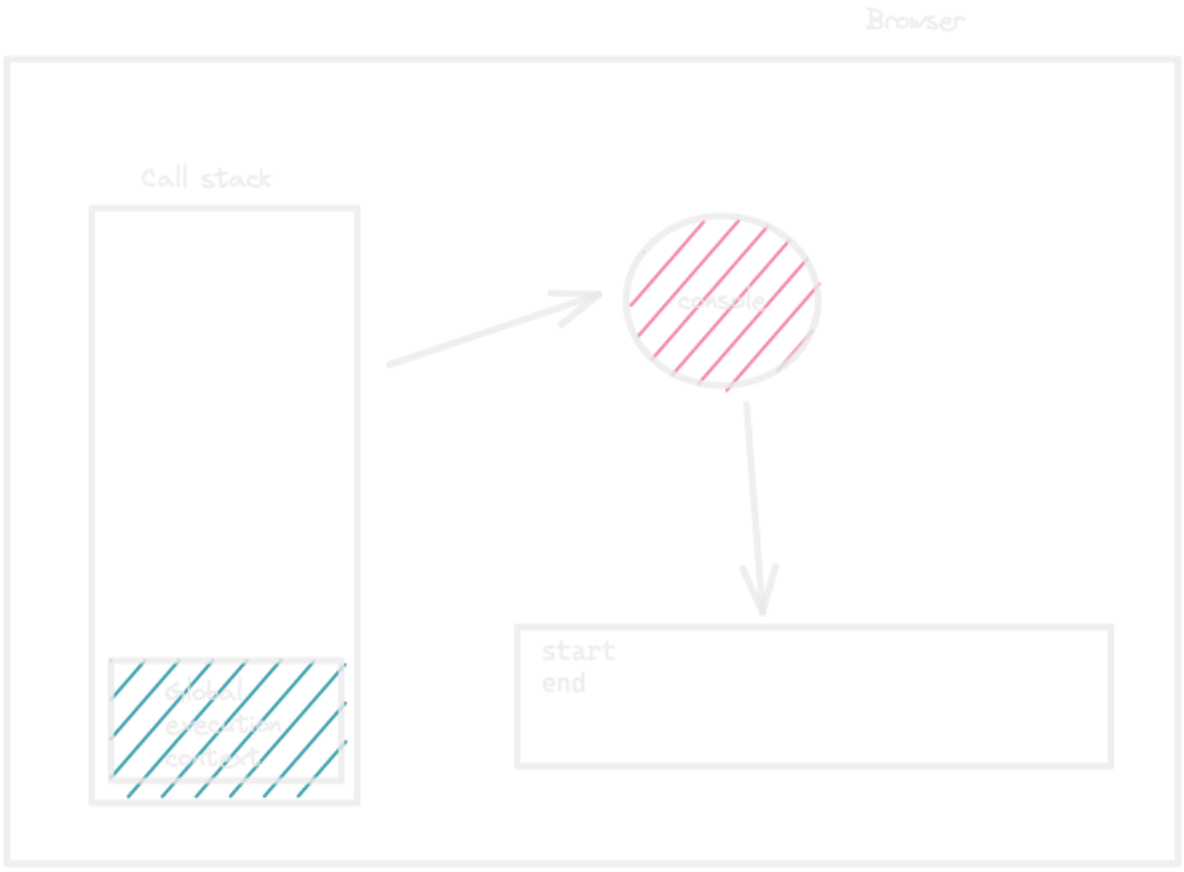


Then the second line will be executed, what it wil do is, it will contact the browser. Browser will start a timer for 4 seconds and register a callback function.



After a callback is registered, javascript engine will move to the next line for execution. It will not wait for 4 seconds to run the callback function because JS is synchronous.

It will execute the last line and print `end` in the console.

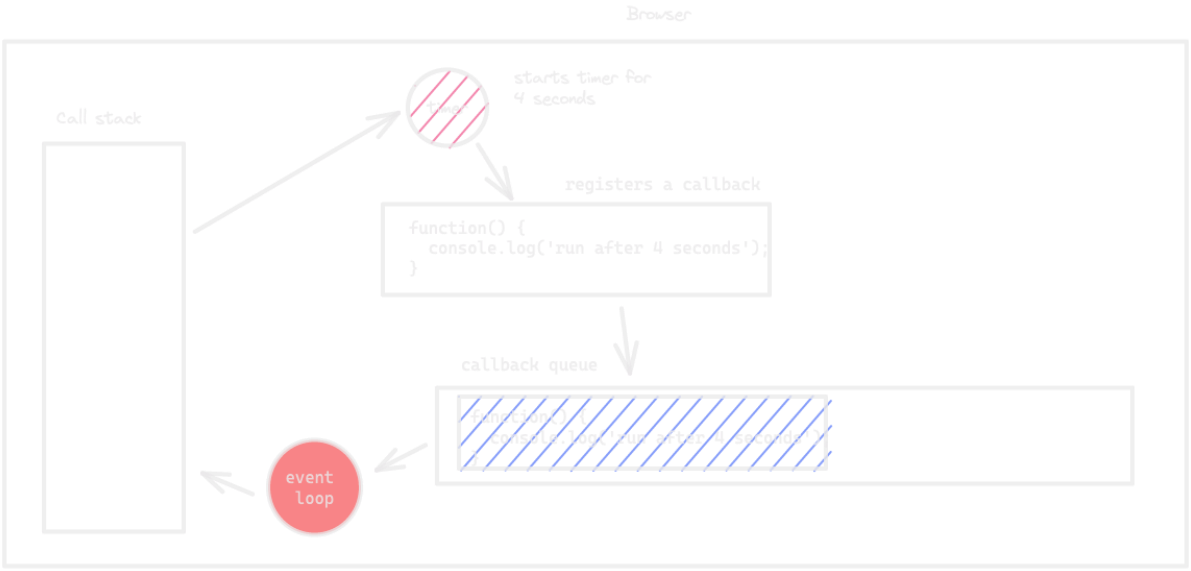


As the last line is executed, global execution context is popped out of the call stack.

How will that callback which is registered to run after 4 seconds be executed?

As we know, in Javascript everything is executed inside the execution context which is inside the call stack. So, it is the duty of browser now to send that callback function to the call stack.

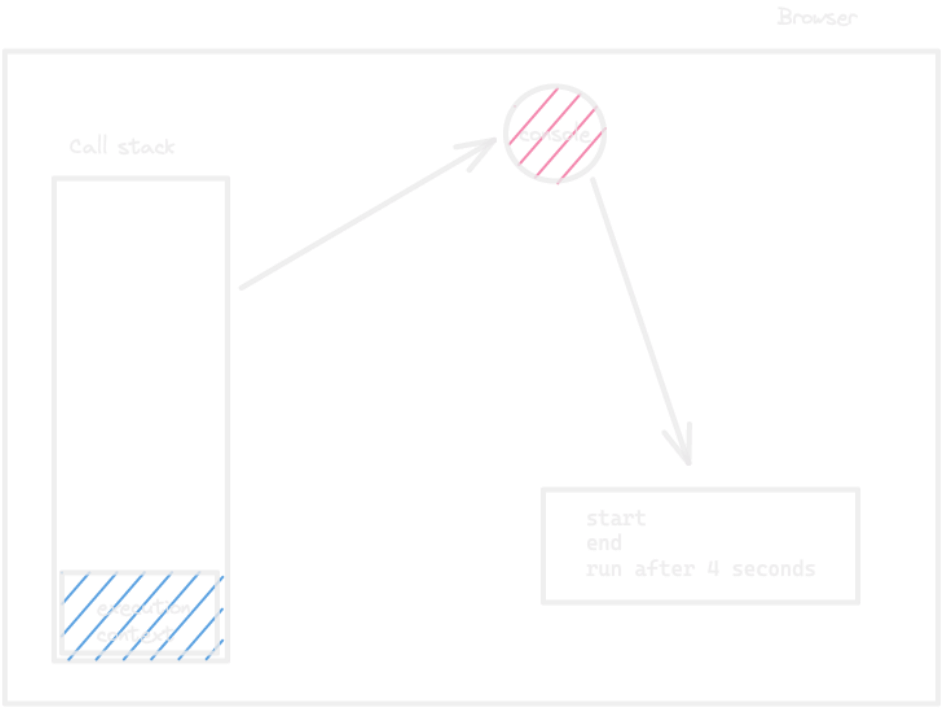
Lets see how, browser send that callback function to the call stack.



When 4 seconds elapsed, the registered callback is pushed to the callback queue and will wait for its turn.

What even loop will do is, it will keep and eye on call stack. As soon as the call stack gets empty, event loop will push the callback from callback queue in the call stack to get executed.

As callback is a function, it will create its own execution context and start executing the code inside it. It will print `run after 4 seconds` to the console.



So, the job of callback queue is to hold all the callbacks that are registered for their turn.

The job of event loop, is to keep an eye on the call stack. Once call stack get empty, event loop will start pushing callback functions to the call stack.

Question

```
console.log('start sam');

setTimeout(function() {
  console.log('run after 2 seconds');
}, 2000);

setTimeout(function() {
  console.log('run after 4 seconds');
}, 4000);

console.log('end sam');

/*
Start sam
End sam
run after 2 seconds
run after 4 seconds
*/
```

Question

```
console.log('start');

setTimeout(function() {
  console.log('run after 0 seconds');
}, 0);

console.log('end');

/*
start
end
run after 0 seconds
*/
```

Callback Hell

Lets say you are buiding an application like instagram or instagram clone. And you want to allow users to upload their pictures. Now, what functions will you write to allow user to upload their pictures.

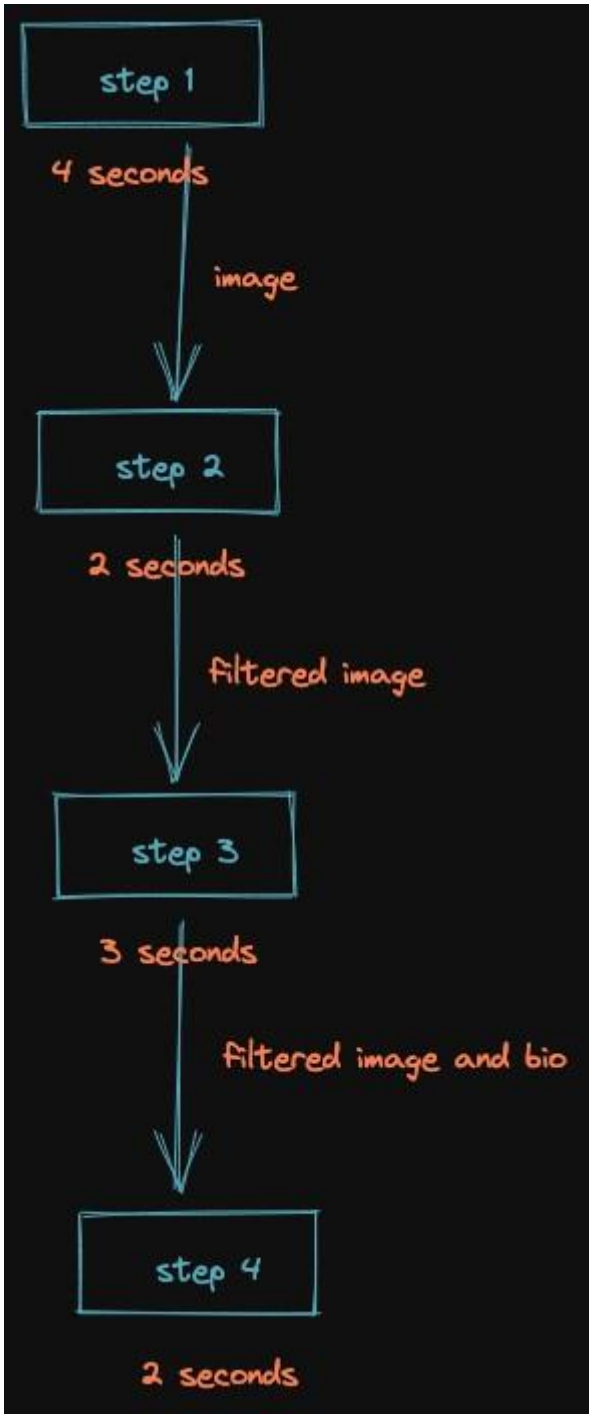
When user click on upload button, you will run a function called `step1` that will open a file explorer or camera to chose image.

Once user selects image, you will run another function `step2` and pass that selected image to that function. This function `step2` will let user to add filters to their image.

Then you will allow user to add caption. For that you will write another function, lets say `step3` for that which will take that filtered photo as argument and let user add caption to the image.

Then, it the last step, you will write a function `step4` to finally upload the image. That function will take final image and caption as argument.

We are going to mock this behaviour. Lets say each function take some time to complete and one function is dependant on another.



How are you going to achieve this? One function should run after completion of the previous function and also have the output of the previous function.

Let consider the 4 functions are as below

```
function step1() {
  setTimeout(function() {
    console.log('Selecting image');
    return 'image';
  }, 4000);
```

```
}

function step2 (image) {
  setTimeout(function() {
    console.log(`Applying filters to ${image}`);
    return 'filtered image'
  }, 2000);
}

function step3(filteredImage) {
  setTimeout(function() {
    console.log(`Adding caption to ${filteredImage}`);
    return 'filtered image with caption';
  }, 3000);
}

function step4(final) {
  setTimeout(function() {
    console.log(`${final} uploaded`);
  }, 2000);
}
```

You can try calling these function like below. Will it work?

As

```
let image = step1();
let filteredImage = step2(image);
let finalImage = step3(filteredImage);
step4(finalImage);

/*
  Applying filters to undefined
  undefined uploaded
  Adding caption to undefined
  Selecting image
*/
```

But this will not work as Javascript will not wait for `step1` function to complete and it instantly calls `step2` function, then `step3` and then `step4` instantly.

it prints `undefined`.
`step2` is called `image` variable is `undefined` as nothing is returned from `step1` . So

Also another problem is `step2` takes least time to finish, so it gets printed first and then `step4` , then `step3` and at last `step1` . This is not what we want. We want it to run sequentially, in order.

How are we going to achieve this?

We can solve this problem using **callbacks**.

What we will do is, we will pass callback functions to each step and will run that callback function when that step ends.

Lets just focus on first two steps.

We will pass a callback function to `step1` and the job of that callback function is to call `step2` with tht required image.
Also instead of returning `image` from `step1`, we pass that `image` to callback function and then that callback function will pass that to `step2` after 4 seconds.

```
 function step1(fn) {
  setTimeout(function() {
    console.log('Selecting image');
    // return 'image';
    fn('image');
  }, 4000);
}

function step2 (image) {
  setTimeout(function() {
    console.log(`Applying filters to ${image}`);
    return 'filtered image'
  }, 2000);
}

step1(function(image) {
  step2(image);
});
```

Now, we want `step3` to run after `step2` gets completed. So same like above, we will pass a callback function to `step2` whose job will be to call `step3` with required arguments.

`step2` will pass `filteredImage` to callback function and then that callback function will pass that to `step3`.

```
function step1(cb) {
  setTimeout(function() {
    console.log('Selecting image');
    // return 'image';
    cb('image');
  }, 4000);
}

function step2 (image, cb) {
  setTimeout(function() {
    console.log(`Applying filters to ${image}`);
    // return 'filtered image'
    cb('filtered image');
```

```
  }, 2000);
}

function step3(filteredImage) {
  setTimeout(function() {
    console.log(`Adding caption to ${filteredImage}`);
    return 'filtered image with caption';
  }, 3000);
}
```

Lets wire up the last `step4`.

```
function step1(cb) {
  setTimeout(function() {
    console.log('Selecting image');
    // return 'image';
    cb('image');
  }, 4000);
}

function step2 (image, cb) {
  setTimeout(function() {
    console.log(`Applying filters to ${image}`);
    // return 'filtered image'
    cb('filtered image');
  }, 2000);
}

function step3(filteredImage, cb) {
  setTimeout(function() {
    console.log(`Adding caption to ${filteredImage}`);
    // return 'filtered image with caption';
    cb('filtered image with caption');
  }, 3000);
}

function step4(final) {
  setTimeout(function() {
    console.log(`${final} uploaded`);
  }, 2000);
}

step1(function(image) {
  step2(image, function(filteredImage) {
    step3(filteredImage, function (finalImage) {
      step4(finalImage);
    });
  });
});
```

Now, everything will work as expected. `step2` will be called after 4 seconds with required argument. Then after 2 seconds `step3` will be called and after 3 seconds

`step4` will be called with required arguments.

Pros

Pros are that you can call one function after the completion of other function in sequential order. Basically you can do async stuff using callbacks

---

Cons

There are 2 problems with this approach.

One is quite evident is that our code is growing horizontly instead of vertically. As we add more callbacks, it will get difficult to maintain the codebase.

Another problem with this code is that we are giving the power to call `step2` to `step1`.

What if `step2` never get called. In this case we ourselves are writing `step1`. But it might not be the case everytime.

Solution

You can solve these problems using Promises.


CALLBACKS
PROMISES
ASYNC/AWAIT