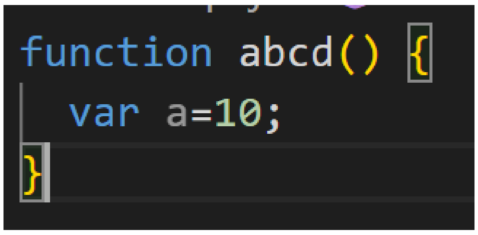
JavaScript Part-3

1. Scope
   - Function scope:
     i. Inside function only it can be accessed

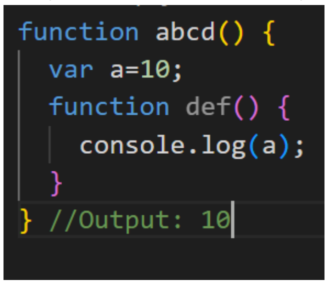


```
function abcd() {
  var a=10;
}
```

   - Global scope:
     i. Entire code anywhere it can be accessed
   - Block scope:
     i. {} only inside this can be accessed

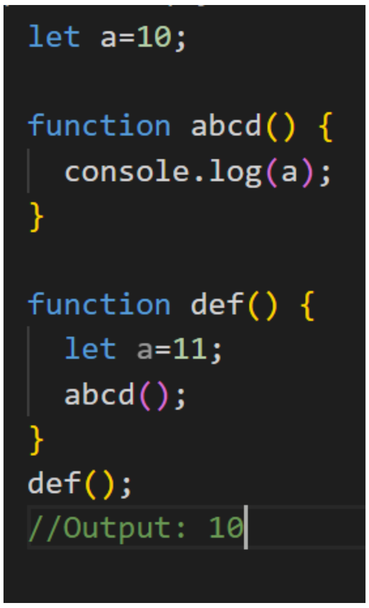
2. Execution Context
   - Memory phase: variables are stored
   - Execution phase: code is executed

3. JS is a Lexical Scope
   - Only Can be accessed anywhere within the function



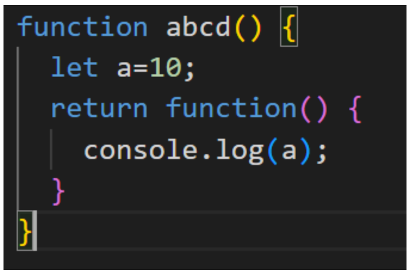
```
function abcd() {
  var a=10;
  function def() {
    console.log(a);
  }
} //Output: 10
```

4. Dynamic scope
   - Not used in JS



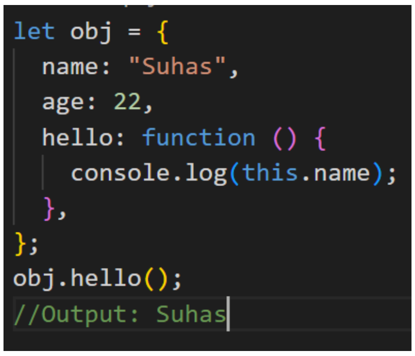
```
let a=10;

function abcd() {
  console.log(a);
}

function def() {
  let a=11;
  abcd();
}
def();
//Output: 10
```

5. Closures
    - Is a function which is inside parent function and the inside function is returning any parent function variable.



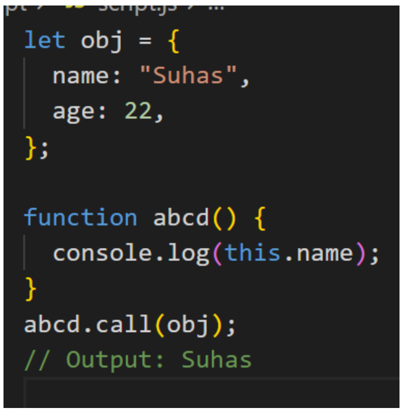
```
function abcd() {
  let a=10;
  return function() {
    console.log(a);
  }
}
```

6. This keyword
    - In global scope value of this: window
    - In function scope value of this: window
    - In method with function: object
    - In method with arrow function: window
    - Function inside function method: window
    - Arrow function inside method: object (always takes from parent)
    - Event handler: element
    - Class: blank object



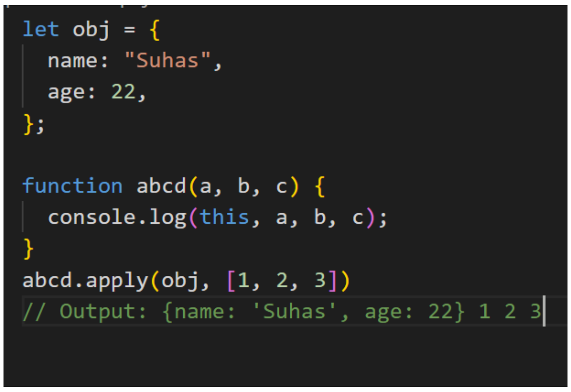
```
let obj = {
  name: "Suhas",
  age: 22,
  hello: function () {
    console.log(this.name);
  },
};
obj.hello();
//Output: Suhas
```

7. Manual Binding
    - Call, bind, apply
    - Call



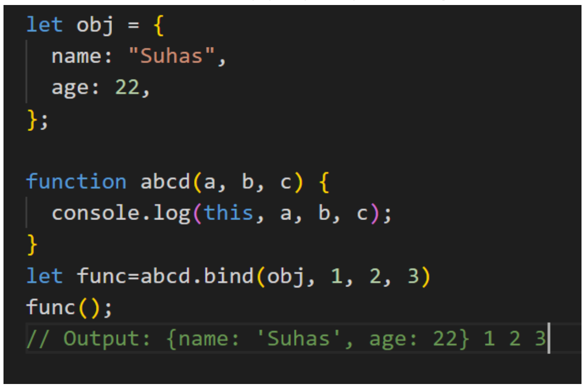
```
let obj = {
  name: "Suhas",
  age: 22,
};

function abcd() {
  console.log(this.name);
}
abcd.call(obj);
// Output: Suhas
```

- Apply: passes 2 values (object, values)



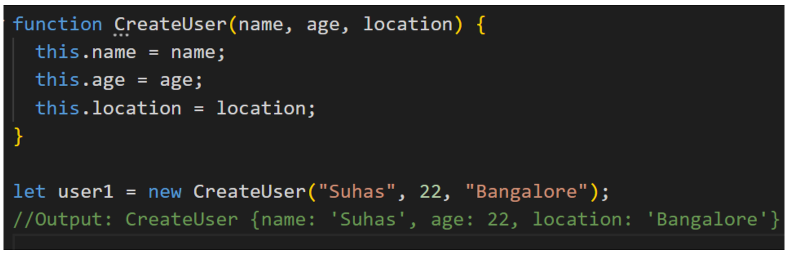
```
let obj = {
  name: "Suhas",
  age: 22,
};

function abcd(a, b, c) {
  console.log(this, a, b, c);
}
abcd.apply(obj, [1, 2, 3])
// Output: {name: 'Suhas', age: 22} 1 2 3
```

- Bind: creates new copy by duplicating



```
let obj = {
  name: "Suhas",
  age: 22,
};

function abcd(a, b, c) {
  console.log(this, a, b, c);
}
let func=abcd.bind(obj, 1, 2, 3)
func();
// Output: {name: 'Suhas', age: 22} 1 2 3
```

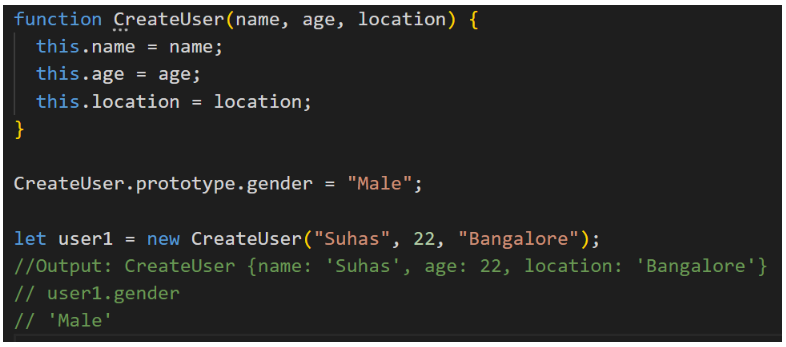
8. Constructor:



```
function CreateUser(name, age, location) {
  this.name = name;
  this.age = age;
  this.location = location;
}

let user1 = new CreateUser("Suhas", 22, "Bangalore");
//Output: CreateUser {name: 'Suhas', age: 22, location: 'Bangalore'}
```

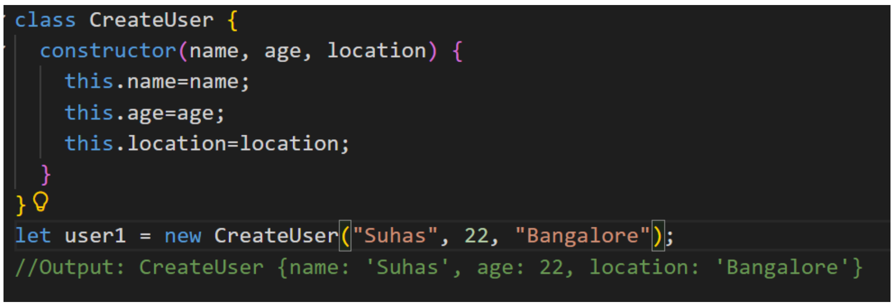
9. Prototype

- Used to add field in constructor



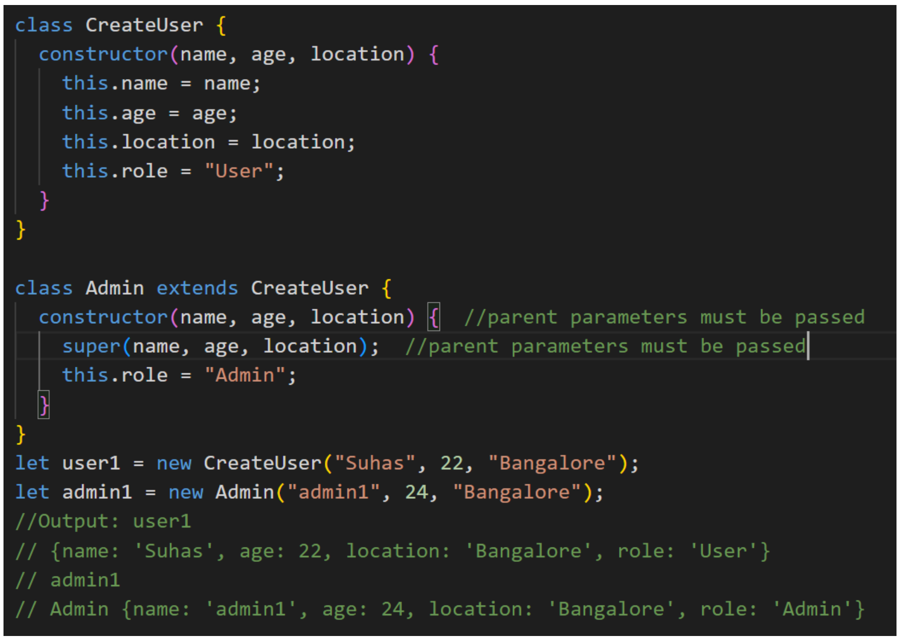
```
function CreateUser(name, age, location) {
  this.name = name;
  this.age = age;
  this.location = location;
}

CreateUser.prototype.gender = "Male";

let user1 = new CreateUser("Suhas", 22, "Bangalore");
//Output: CreateUser {name: 'Suhas', age: 22, location: 'Bangalore'}
// user1.gender
// 'Male'
```

10. Class



```
class CreateUser {
  constructor(name, age, location) {
    this.name=name;
    this.age=age;
    this.location=location;
  }
}
let user1 = new CreateUser("Suhas", 22, "Bangalore");
//Output: CreateUser {name: 'Suhas', age: 22, location: 'Bangalore'}
```

11. Extends, super

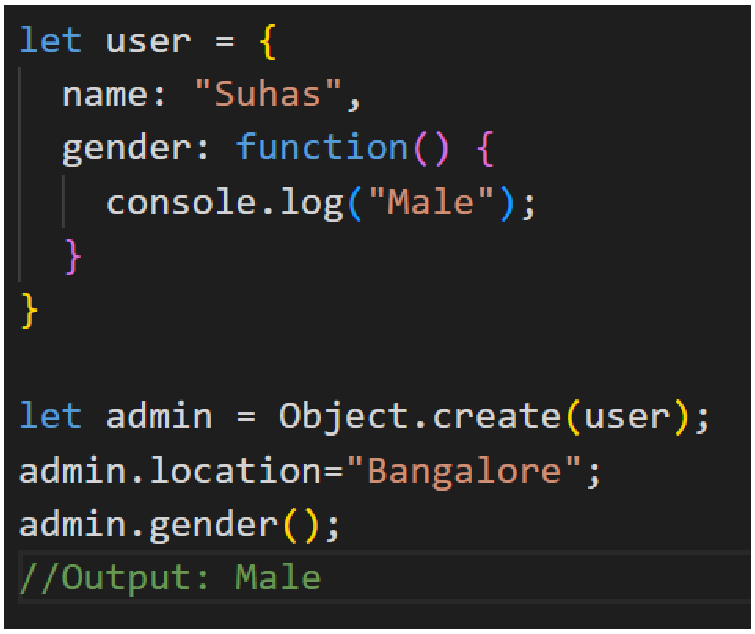


```
class CreateUser {
  constructor(name, age, location) {
    this.name = name;
    this.age = age;
    this.location = location;
    this.role = "User";
  }
}

class Admin extends CreateUser {
  constructor(name, age, location) {  //parent parameters must be passed
    super(name, age, location);  //parent parameters must be passed
    this.role = "Admin";
  }
}
let user1 = new CreateUser("Suhas", 22, "Bangalore");
let admin1 = new Admin("admin1", 24, "Bangalore");
//Output: user1
// {name: 'Suhas', age: 22, location: 'Bangalore', role: 'User'}
// admin1
// Admin {name: 'admin1', age: 24, location: 'Bangalore', role: 'Admin'}
```

-

12. Prototype Inheritance

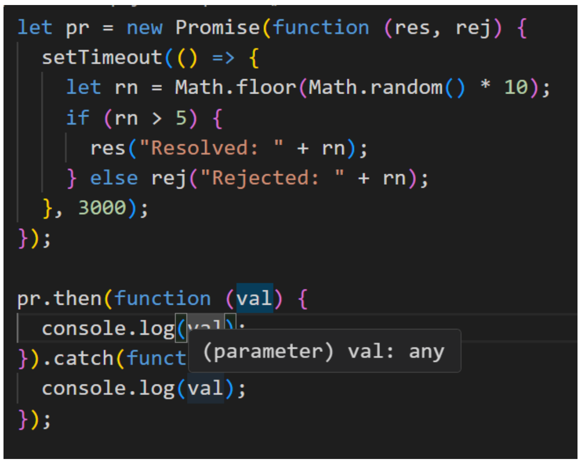
- Similar to inheritance but extra Object.create();



```
let user = {
  name: "Suhas",
  gender: function() {
    console.log("Male");
  }
}

let admin = Object.create(user);
admin.location="Bangalore";
admin.gender();
//Output: Male
```

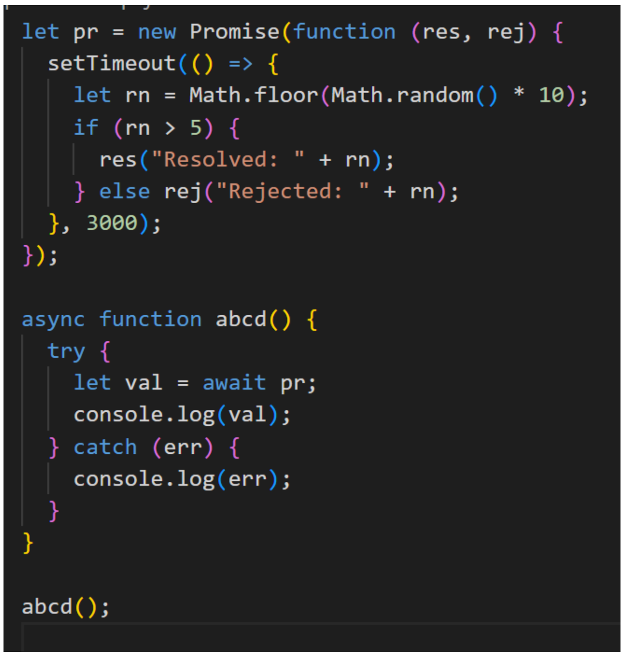
-

13. Asynchronous
    - Sync: line by line execute
    - Async: which is ready to execute
14. Callback hell
    - Callback inside callback
15. Promises
    - Resolve, reject, then, catch



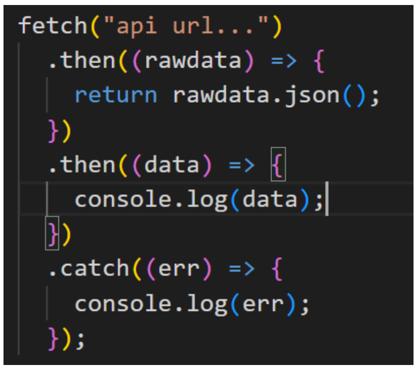
```
let pr = new Promise(function (res, rej) {
  setTimeout(() => {
    let rn = Math.floor(Math.random() * 10);
    if (rn > 5) {
      res("Resolved: " + rn);
    } else rej("Rejected: " + rn);
  }, 3000);
});

pr.then(function (val) {
  console.log(val);
}).catch(funct
  console.log(val);
});
```

16. Async & await



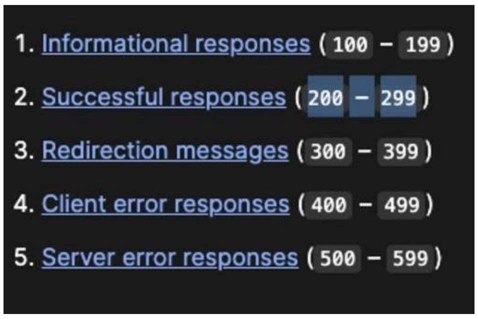
```
let pr = new Promise(function (res, rej) {
  setTimeout(() => {
    let rn = Math.floor(Math.random() * 10);
    if (rn > 5) {
      res("Resolved: " + rn);
    } else rej("Rejected: " + rn);
  }, 3000);
});

async function abcd() {
  try {
    let val = await pr;
    console.log(val);
  } catch (err) {
    console.log(err);
  }
}

abcd();
```

17. Fetch API + HTTP Basics



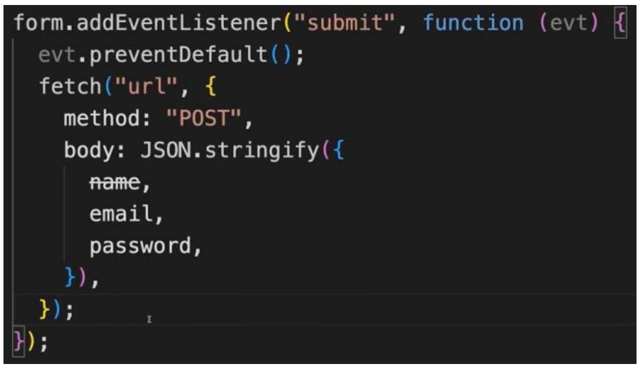
```
fetch("api url...")
  .then((rawdata) => {
    return rawdata.json();
  })
  .then((data) => {
    console.log(data);
  })
  .catch((err) => {
    console.log(err);
  });
```

18. Response Codes



1. Informational responses (100 – 199)
2. Successful responses (200 – 299)
3. Redirection messages (300 – 399)
4. Client error responses (400 – 499)
5. Server error responses (500 – 599)

19. Form Submission using fetch



```
form.addEventListener("submit", function (evt) {
  evt.preventDefault();
  fetch("url", {
    method: "POST",
    body: JSON.stringify({
      ~~name~~,
      email,
      password,
    }),
  });
});
```

20.