2. JavaScript Advanced

1. Nested Functions

- Lexical Scoping - Starts with innermost and moves outwards towards global scope.
- Have access to variables defined in their own scope and outer scope.

```
let a = 10;
function outer() {
  let b = 20;
  function inner() {
    let c = 30;
    console.log(a, b, c);
  }
  inner();
}

outer();

//Output: 10 20 30
```

---

2. Closure

- MDN - Combination of function and reference to the surrounding state.
- Created every time a function is created.

```
function outer() {
  let counter = 0;
  function inner() {
```

```
    counter++;
    console.log(counter);
  }
  inner();
}
outer();
outer();

// 1
// 1
```

- With every new invocation of a function - a new temporary memory is established - i.e. the counter variable runs from initial value.

```
function outer() {
  let counter = 0;
  function inner() {
    counter++;
    console.log(counter);
  }
  return inner;
}

const fn = outer();
fn();
fn();

// 1
// 2
```

- Combination of function and scope chain
  - When we return a function from another function - we return a combination of function - definition, and scope.
  - This allows function definition to have a persistent memory to hold live data between executions.
  - Therefore, a function is always packed with a lexical environment.

---

3. Function Currying

- Process in functional programming
- Transforms a function with multiple arguments - into a sequence of nested functions - taking one argument at a time

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

```
console.log(sum(2, 3, 5));

function curry(fn) {
  return function (a) {
    return function (b) {
      return function (c) {
        return fn(a, b, c);
      };
    };
  };
}

const curriedSum = curry(sum);
console.log(curriedSum(2)(3)(5));

const add2 = curriedSum(2);
const add3 = add2(3);
const add5 = add3(5);

console.log(add5);
```

---

4. *this* keyword

- Used in a function - refers to the object it belongs to.
- Makes the function reusable by deciding the object value.
- However, object value is based on the procedure by which the function is called - runtime binding.
- Implicit Binding

```
const person = {
  name: "Nikhil",
  sayMyName: function () {
    console.log(`Name : ${this.name}`);
  },
};

person.sayMyName();

// Name: Nikhil
```

- Explicit Binding

```
const person = {
  name: "Nikhil",
};

function sayMyName() {
  console.log(`Name : ${this.name}`);
}
```

```
sayMyName.call(person);

// Name: Nikhil
```

- New Binding

```
function Person(name) {
  this.name = name;
}

const person1 = new Person("Nikhil");
console.log(person1.name);

// Nikhil
```

- Default
  - In this case, *this* keyword relies on Global Scope.

```
globalThis.name = "NikTan";
function sayMyName() {
  console.log(`My name is ${this.name}`);
}

sayMyName();
```

- Order of Precedence
  - New → Explicit → Implicit → Default

---

5. Prototype

- Typical approach to objects and adding a new property.

```
//Adding a new function to the existing object

function Person(fname, lname) {
  this.firstName = fname;
  this.lastName = lname;
}

const person1 = new Person("Nik", "Tan");
person1.getFullName = function () {
  return `${this.firstName} and ${this.lastName}`;
};

console.log(person1.getFullName());

//Output: Nik and Tan
```

- However, if we have to leverage the newly added property for Person 2, it would not be available.
- An error would be thrown.
- For this purpose, we can use Prototypes - this would allow adding a new property or method to all available instances.
- We can use the below-given syntax to add a prototype.

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

5.1 Prototypal Inheritance

```
// Typical Person Method

function Person(fName, lName) {
  this.firstName = fName;
  this.lastName = lName;
}

// Adding a new property to the Person object using the prototype

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

// Creating a new Superhero Method

function Superhero(fName, lName) {
  Person.call(this, fName, lName); // Explicit Binding to Person method
  this.isSuperHero = true; // A generic property of method
}

// Cleanup - This allows JavaScript to understand that Superhero is not inherited from Person
Superhero.prototype.constructor = Superhero;

// Creating a fallback for Superhero using Object. Create
Superhero.prototype = Object.create(Person.prototype);

// Adding a new property to the Superhero object using this prototype

Superhero.prototype.fightCrime = function () {
  console.log("Fights Crime!");
};

const batman = new Superhero("Bruce", "Wayne");
console.log(batman.getFullName());
```

6. Class

- Syntactical sugar over prototype method.

```
class Person { // Simply define the method as Class
  constructor(fName, lName) { //Within const function - write the parameters
    this.firstName = fName;
    this.lastName = lName;
  }

//Within the class - write any associated functions
  sayMyName() {
    return `${this.firstName} with ${this.lastName}`;
  }
}

//Create new instances of the class using new keyword
const personP1 = new Person("A", "B");
console.log(personP1.sayMyName());

// Inheritance

// Simply use the extends keyword - followed by the class to inherit from

class Superhero extends Person {
  constructor(fName, lName) {       // define your constructor function
    super(fName, lName);            // In super, define the properties to inherit
    this.isSuperHero = true;
  }

  fightsCrime() {
    console.log("Superhero Fights Crime!");
  }
}

const batman = new Superhero("Bruce", "Wayne");
console.log(batman.sayMyName());
```

---

7. Iterables and Iterators

- Regular methods of accessing values in array or string - we tend to use for loop and increment a specific iterator.
- However, it is difficult to keep track of the iteration and for different data structures, we have to use varying types of methods.
- Therefore, there was a need for having a specific concept that could abstract the complexity and be uniform in data structures.
- We need to be focused on what to do with the data rather than just how to access the data.

```
//String
const myStr = "Nikhil";
for (const character of myStr) {
  console.log(character);
}

//Array
const myArr = [1, 2, 3, 4, 5];
for (const value of myArr) {
  console.log(value);
}
```

```
// Typical Iterator Object
const obj = {
  [Symbol.iterator]: function () {
    let step = 0;
    const iterator = {
      next: function () {
        step++;
        if (step == 1) {
          return { value: "Hey", done: false };
        } else if (step == 2) {
          return { value: "There", done: false };
        }
        return { value: "There", done: true };
      },
    };
    return iterator;
  },
};

for (const word of obj) {
  console.log(word);
}
```

---

8. Generators

- Simple functions - allows to simplify the task of creating iterators.

```
function* generatorFunction() { //define a function with asterisk
  yield "Hey";                  //use yield keyword
  yield "There";
}

const generatorObject = generatorFunction(); //define a object for generator function

for (const word of generatorObject) { //access using for .. of loop
  console.log(word);
}
```