6. Programming Paradigms

6.1 Types

- **Functional Programming (FP)**
  - Building programs by applying and composing functions
  - Data is passed into function as and when required
  - New values are returned → which are used at some place in the program
- **Object-Oriented Programming (OOP)**
  - Data and functions are stored as objects and methods
  - Methods tend to update properties in OOP and do not return new values
  - Emphasizes on modelling of real-life objects → grouping of properties

```
var virtualPet = {
  sleepy : false,
  nap : function(){
    console.log('Pet is Sleepy');
}
}
```

- **First-Class Functions**
  - Can be passed to another functions

    - Saved in a variable
    - Returned from another functions
    - In simple words, here - we treat First-Class functions similar to string or numbers
- **Higher-Order Functions (HOF)**
    - Two potential properties - Either one of them needs to be true at once
        - Accept functions as argument
        - Returns functions when invoked
- Pure Functions
    - Tends to return same values → When given same input values
    - Should not have side effects
        - Function makes a change outside of itself
            - Change variables - Browser API - Math Random

6.2 Scope

- **Var**
    - We can use this keyword even before providing the value.

```
console.log(user)
var user = 'Nik'

//undefined
```

    - The code will execute without any issue.
    - We can also declare and re-declare the same variables.

```
var user = 'Nik'
var user = 'Man'

console.log(user)

//Man
```

- **Let**

- We can declare the variable and not provide any value → This will work without error.

```
let user;
console.log(user)

// undefined
```

- However, we can not re-declare this variable. This will produce error.

```
let user;
let user = 'Nik'

// Error
```

- We can re-assign it.

```
user = 'Man'
console.log(user)

// Man
```

- **Const**
  - Strictest of all → Needs to be declared and initialized together.

```
const user;
user = 'Nik'

// Error
// Need to be initialized when declared.
```

  - Can not be re-declared.

5.3 Object Oriented Programming (OOP)

- Classification or a general way of writing code
- In OOP
  - Data is grouped
- In FP

    - Data is kept separate
- In OOP → The same object needs to be repeated → Waste of resources
    - Solution : Create method templates

```
class Car {
    constructor(color, speed){
        this.color = color;
        this.speed = speed;
    }

    stats(){
        console.log(`The car has ${this.color} color and ${this.speed} speed`);
    }

    turboOn(){
        console.log('Turbo is On!');
    }
}

const car1 = new Car('Blue', 120);
car1.stats();
```

- Advantages of using OOP
    - Modular ↔ Flexible ↔ Reusable : Code

5.4 Principles of OOP

- **Inheritance**
    - Core foundation of OOP
    - Base class inherits properties from Super Class
    - This base class can also be a Super Class for further Classes
    - We use `*extends*` keyword here.

```
class Animal {//...//}
class Bird extends Animal {//...//}
class Eagle extends Bird {//...//}
```

- **Abstraction**
    - Focus on the essential features of an object or system while ignoring the irrelevant details.

    - It allows you to create a simplified model of a complex system that can be easily understood and used.
    - For example, a car dashboard is an abstraction of the car's internal systems, providing the driver with a simplified view of important information like speed, fuel level, and engine temperature.
- **Encapsulation**
    - In simple words → Keeping the implementation of code : hidden
    - We don’t need to know how the code works in order to consume it
- **Polymorphism**
    - One that can take multiple forms
    - Something with many shapes
    - Here, the bell method behaves differently on the type of object passed to it.

```
const bicycle = {
  bell: function () {
    return "Bicycle! Watch out, please!";
  },
};
const door = {
  bell: function () {
    return "Door! Come here, please!";
  },
};

function ringTheBell(thing) {
  console.log(thing.bell());
}

ringTheBell(bicycle);
ringTheBell(door);
```

- Similarly
    - Concatenation methods works differently based on what data type is specified

```
console.log("abc".concat("def"));
console.log(["abc"].concat(["def"]));
console.log(["abc"] + ["def"]);

// abcdef
```

```
// ['abc','def']
// abcdef
```

- Another example

```
class Bird {
    useWings() {
        console.log("Flying!")
    }
}

class Eagle extends Bird {
    useWings() {
        super.useWings()
        console.log("Barely flapping!")
    }
}

class Penguin extends Bird {
    useWings() {
        console.log("Diving!")
    }
}

var baldEagle = new Eagle();
var kingPenguin = new Penguin();
baldEagle.useWings(); // "Flying! Barely flapping!"
kingPenguin.useWings(); // "Diving!"

// use Wings method has different manipulations based on the input data
```

- Another Example
    - Focus on **get Self** and **get Prototype** functions
    - get Self → Logs the object of which it is a part of
    - get Prototype → Logs all the properties that are available on the prototype

```
> class Train {
    constructor(color, lightsOn) {
      this.color = color;
      this.lightsOn = lightsOn;
    }
    toggleLights() {
      this.lightsOn = !this.lightsOn;
    }
    lightsStatus() {
      console.log("Lights on?", this.lightsOn);
    }
    getSelf() {
      console.log(this);
    }
    getPrototype() {
      var proto = Object.getPrototypeOf(this);
      console.log(proto);
    }
  }

  const train1 = new Train("Blue", false);
< undefined
> train1.getSelf()
  ▶ Train {color: 'Blue', lightsOn: false}
< undefined
> train1.getPrototype()
  ▼{constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}
    ▶ constructor: class Train
    ▶ getPrototype: ƒ getPrototype()
    ▶ getSelf: ƒ getSelf()
    ▶ lightsStatus: ƒ lightsStatus()
    ▶ toggleLights: ƒ toggleLights()
    ▶ [[Prototype]]: Object
< undefined
>
```

- **Inheritance in Class**

```
class HighSpeedTrain extends Train {
  constructor(passengers, highSpeedOn, color, lightsOn) {
    super(color, lightsOn);
    this.passengers = passengers;
    this.highSpeedOn = highSpeedOn;
  }
}
```

  - Here, we use the super keyword → This allows us to define what properties will get inherited from the parent class.
  - In this case - color and lights On props are being inherited.

5.5 Default Parameters

```
function noDefaultParams(number) {
  let result = number * number;
  console.log(`Square of ${number} is ${result}`);
}

noDefaultParams();

// Square of undefined is NaN
```

- Doesn’t throw an error → But returns a non-sensical output
- For this purpose, we can specify → Default Parameters

```
function noDefaultParams(number = 10) {
  let result = number * number;
  console.log(`Square of ${number} is ${result}`);
}

noDefaultParams();

// Square of 10 is 100
```