# 2024-03-26

## Object :

- collection of properties and methods

## Constructor function :

A constructor function in JavaScript is a special type of function that is used to create and initialize objects. It serves as a blueprint for creating multiple instances of similar objects with predefined properties and methods. When you create a new instance of an object using a constructor function, you are essentially invoking that function as a constructor.

Here's a basic example of a constructor function:

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

// Creating instances of Person using the constructor function
const person1 = new Person("John", 30);
const person2 = new Person("Alice", 25);

console.log(person1); // Output: Person { name: 'John', age: 30 }
console.log(person2); // Output: Person { name: 'Alice', age: 25 }
```

In the above example:

- We define a constructor function `Person` which takes `name` and `age` parameters.
- Inside the constructor function, `this` refers to the newly created instance of the object.
- We assign the values of `name` and `age` passed to the constructor to properties of the object ( `this.name` and `this.age` ).
- We create new instances of `Person` using the `new` keyword, which invokes the constructor function with the specified arguments.

Constructor functions are commonly used in JavaScript for object creation, especially when you need to create multiple instances of objects with similar properties and behaviors. They provide a convenient way to create objects with predefined initial values and methods.

## Prototypes

## Classes

## Instances

In JavaScript, instances refer to individual objects created from a constructor function or a class. When you use the `new` keyword with a constructor function or a class, it creates a new instance of that function or class.

Here's an example of creating instances using a constructor function:

```
// Constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Creating instances of Person
const person1 = new Person("John", 30);
const person2 = new Person("Alice", 25);
```

In this example, `person1` and `person2` are instances of the `Person` constructor function. Each instance has its own set of properties (`name` and `age`), which are assigned when the instance is created.

Instances in JavaScript allow you to create multiple objects with similar properties and behaviors using the same constructor function or class blueprint. Each instance is independent of other instances and can have its own unique data.

Instances are fundamental in object-oriented programming as they enable code reuse, encapsulation, and the creation of multiple objects with similar characteristics. You can manipulate and interact with each instance independently, allowing for modular and scalable code.

# The 4 pillars :

## Abstraction

In object-oriented programming (OOP), abstraction is the concept of hiding the implementation details and showing only the essential features of an object to the outside world. This allows for simpler and more modular code, as well as easier maintenance and updates. JavaScript, although not traditionally seen as a classical OOP language like Java or C++, still supports abstraction through various mechanisms.

## Encapsulation

Encapsulation in Object-Oriented Programming (OOP) refers to the bundling of data and methods that operate on that data into a single unit, known as a class. This concept helps in hiding the internal state and behaviors of an object from the outside world and only exposing

the necessary functionality through well-defined interfaces. JavaScript, though not a classical OOP language like Java or C++, still supports encapsulation through various techniques. Let's delve into them with examples:

1. **Using Function Constructors:** Function constructors in JavaScript can be used to create objects with private properties and methods using closures.

```
function Car(make, model) {
    // Private variables
    let _make = make;
    let _model = model;

    // Public methods
    this.getMake = function() {
        return _make;
    };

    this.getModel = function() {
        return _model;
    };

    this.displayInfo = function() {
        return `${_make} ${_model}`;
    };
}

let myCar = new Car('Toyota', 'Corolla');
console.log(myCar.getMake()); // Output: Toyota
console.log(myCar.getModel()); // Output: Corolla
console.log(myCar._make); // Output: undefined (private)
console.log(myCar.displayInfo()); // Output: Toyota Corolla
```

2. **Using ES6 Classes:** With the introduction of ES6, JavaScript has a more structured way of implementing classes, which also facilitates encapsulation.

```
class Employee {
    constructor(name, age) {
        // Private properties
        let _name = name;
        let _age = age;

        // Public methods
```

```
        this.getName = function() {
            return _name;
        };

        this.getAge = function() {
            return _age;
        };

        this.printInfo = function() {
            return `${_name} is ${_age} years old.`;
        };
    }
}

let emp1 = new Employee('John', 30);
console.log(emp1.getName()); // Output: John
console.log(emp1.getAge()); // Output: 30
console.log(emp1._name); // Output: undefined (private)
console.log(emp1.printInfo()); // Output: John is 30 years old.
```

# Inheritance

Inheritance in Object-Oriented Programming (OOP) is a mechanism where a new class inherits properties and behaviors from an existing class. JavaScript, although prototype-based rather than class-based, supports inheritance through prototype chaining.

```
// Parent Constructor
function Animal(name) {
  this.name = name;
}

// Adding a method to Parent's prototype
Animal.prototype.sayName = function() {
  console.log("My name is " + this.name);
};

// Child Constructor
function Dog(name, breed) {
  Animal.call(this, name); // Call the parent constructor
  this.breed = breed;
}
```

```
// Inheriting from Parent
Dog.prototype = Object.create(Animal.prototype);

// Adding a method to Child's prototype
Dog.prototype.bark = function() {
  console.log("Woof!");
};

// Creating instances
let myDog = new Dog("Max", "Labrador");
myDog.sayName(); // Output: My name is Max
myDog.bark();    // Output: Woof!
```

# Polymorphism

Polymorphism in object-oriented programming (OOP) refers to the ability of different objects to respond to the same message or method call in different ways. In JavaScript, polymorphism can be achieved through various mechanisms such as function overriding, method overloading, or using interfaces. Let's explore these concepts with examples:

1. **Function Overriding**: Function overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. In JavaScript, you can achieve this by defining a method with the same name in the subclass.

```
class Animal {
  sound() {
    console.log("Some generic sound");
  }
}

class Dog extends Animal {
  sound() {
    console.log("Bark");
  }
}

class Cat extends Animal {
  sound() {
    console.log("Meow");
  }
}
```

```
const dog = new Dog();
const cat = new Cat();

dog.sound(); // Output: "Bark"
cat.sound(); // Output: "Meow"
```

---

# Prototype based programming :

A prototype-based language is a type of programming language where objects are the primary mechanism for organizing and storing data, rather than classes. In a prototype-based language, objects serve as prototypes for creating new objects, and inheritance is achieved by cloning existing objects and extending or modifying them as needed.

In a prototype-based language:

1. **Objects are Dynamic**: Objects can be created and modified at runtime without the need for predefined class definitions. This flexibility allows for rapid prototyping and adaptation to changing requirements.
2. **Inheritance via Prototypes**: Inheritance is achieved by cloning existing objects (prototypes) and then extending or modifying them to create new objects. The new objects inherit properties and behaviors from their prototypes.
3. **No Classes**: Unlike class-based languages such as Java or C++, where classes define the structure and behavior of objects, prototype-based languages do not have classes. Instead, objects directly serve as prototypes for creating new objects.

JavaScript is a prominent example of a prototype-based language. In JavaScript, objects inherit properties and behaviors from their prototypes through a mechanism called the prototype chain. Each object has a prototype, which is another object. If a property or method is not found on the object itself, JavaScript looks for it in the object's prototype, and if not found there, it continues up the prototype chain until it reaches the end.

Prototype-based languages offer a different paradigm compared to class-based languages, and they can be particularly useful for certain types of applications, such as those requiring flexible and dynamic object creation. However, they may also require careful design to ensure code maintainability and readability.

Prototype-based programming and object-oriented programming (OOP) are both paradigms for organizing and structuring code around objects, but they have some key differences:

1. **Class vs. Prototype**:

- OOP: In traditional OOP languages like Java or C++, objects are instances of classes. Classes serve as blueprints or templates for creating objects. Objects are instances of classes, and they inherit properties and behaviors defined in their class.
- Prototype-based: In prototype-based languages like JavaScript, objects are created directly from other objects, called prototypes. There are no classes in the traditional sense. Objects serve as prototypes for creating new objects, and inheritance is achieved through cloning and extending existing objects.

2. **Inheritance Mechanism**:
   - OOP: In class-based OOP, inheritance is typically achieved through class hierarchies. Subclasses inherit properties and methods from their superclass. This inheritance hierarchy is determined at compile-time.
   - Prototype-based: In prototype-based programming, inheritance is achieved by cloning existing objects (prototypes) and extending or modifying them to create new objects. Objects inherit properties and behaviors directly from their prototypes, and the prototype chain determines the inheritance relationships. This inheritance mechanism is more dynamic and flexible than class-based inheritance.
3. **Flexibility and Dynamic Nature**:
   - OOP: Class-based OOP languages often require class definitions and declarations before objects can be created. Changes to the class hierarchy may require modifying class definitions and potentially recompiling the code.
   - Prototype-based: Prototype-based languages offer more flexibility and dynamism because objects can be created and modified at runtime without predefined class definitions. This allows for rapid prototyping and adaptation to changing requirements.
4. **Usage**:
   - OOP: Class-based OOP is commonly used in languages like Java, C++, C#, and Python.
   - Prototype-based: Prototype-based programming is commonly used in languages like JavaScript.
5. **Code Organization**:
   - OOP: In class-based OOP, code is organized around classes, which encapsulate both data and behavior.
   - Prototype-based: In prototype-based programming, code is organized around objects, and objects serve as both data containers and behavior holders.

While both paradigms aim to achieve similar goals of code organization, encapsulation, and reuse, they offer different approaches and trade-offs. Prototype-based programming tends to be more lightweight and flexible, while class-based OOP provides a more structured and statically-typed approach.

---

# Event driven programming :

Event-driven programming is a programming paradigm in which the flow of the program is determined by events that occur either asynchronously or synchronously. In event-driven programming, the program responds to events triggered by the user, the system, or other programs.

Key concepts in event-driven programming include:

1. **Events**: Events are occurrences that happen during the execution of a program, such as user actions (clicking a button, typing into a text field), system notifications (timer expiration, file loading), or messages from other parts of the program. Events can be generated by hardware, software, or the user.
2. **Event Handlers or Listeners**: Event handlers (also called event listeners) are functions or code blocks that are executed in response to specific events. When an event occurs, the corresponding event handler is invoked to handle the event.
3. **Event Loop**: The event loop is a central component of event-driven programming environments. It continuously checks for events and dispatches them to the appropriate event handlers. It ensures that the program remains responsive to user interactions and other events while executing code.
4. **Asynchronous Programming**: Event-driven programming often involves asynchronous operations, where code execution does not block while waiting for certain operations to complete. Asynchronous operations are typically used for I/O operations (such as fetching data from a server or reading files) and allow the program to continue executing other tasks while waiting for the operation to finish.

Event-driven programming is commonly used in various software development domains, including:

- **User Interfaces (UI)**: Event-driven programming is widely used in building interactive user interfaces for web applications, desktop applications, and mobile apps. User interactions such as clicking buttons, typing text, or resizing windows trigger events that are handled by event handlers to update the UI accordingly.
- **Networking**: Event-driven programming is prevalent in network programming, especially in handling asynchronous communication tasks such as sending and receiving data over the network. Events such as incoming data packets or connection status changes trigger event handlers to process the data or respond to the changes.
- **Graphical User Interfaces (GUI)**: Many GUI frameworks and toolkits, such as Qt, GTK, and Windows Forms, are based on event-driven programming models. GUI events such as mouse movements, keyboard inputs, or window resizing trigger event handlers to update the GUI elements and respond to user interactions.

Overall, event-driven programming facilitates the development of responsive, interactive, and scalable software systems by decoupling event generation from event handling and

enabling non-blocking, asynchronous execution of code.

---

A callback function in JavaScript is a function that is passed as an argument to another function and is executed after the completion of a particular task or operation. Callback functions are commonly used in asynchronous programming to handle the result of asynchronous operations such as fetching data from a server, reading files, or handling user interactions.

Here's a detailed explanation of callback functions along with some examples:

1. **Basic Concept**:
   - A callback function is simply a function that is passed as an argument to another function.
   - The function receiving the callback function will execute it at some point in the future, typically after completing some asynchronous task.
   - Callback functions allow us to perform actions after certain events or tasks have been completed, enabling non-blocking behavior in JavaScript.
2. **Example 1: Asynchronous File Reading**:
   In this example, we use the `readFile` function from the Node.js `fs` module to read the contents of a file asynchronously. We pass a callback function that will be executed once the file reading operation is complete.

```
const fs = require('fs');

// Asynchronous file reading with a callback function
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File contents:', data);
});
```

3. **Example 2: Event Handling in a Web Browser**:
   In this example, we use callback functions to handle user interactions in a web browser. We add an event listener to a button element, and when the button is clicked, the callback function is executed.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>Callback Example</title>
</head>
<body>
  <button id="myButton">Click Me</button>
  <script>
    // Event handling with a callback function
    document.getElementById('myButton').addEventListener('click', () => {
      console.log('Button clicked');
    });
  </script>
</body>
</html>
```

4. **Example 3: Custom Callback Function**:
   In this example, we define a function `calculate` that takes two numbers and a callback function as arguments. The callback function is invoked with the result of the calculation.

```
// Custom function with a callback
function calculate(x, y, callback) {
  const result = x + y;
  callback(result);
}

// Using the custom function with a callback
calculate(5, 3, (sum) => {
  console.log('Sum:', sum); // Output: 8
});
```

These examples demonstrate the versatility and usefulness of callback functions in JavaScript, particularly in scenarios involving asynchronous operations or event handling. Callback functions allow for modular and flexible code by enabling the separation of concerns and promoting code reuse.

---

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of a document as a tree-like structure where each node represents a part of the document, such as elements, attributes, and text. The DOM provides a way to access, manipulate, and update the content, structure, and style of HTML and XML documents dynamically using JavaScript.

Here's a brief overview of the DOM and some examples:

1. **Document Structure**:

- The root node of the DOM tree is the `document` object, which represents the entire HTML document.
- Elements in the HTML document are represented as nodes in the DOM tree, with parent-child relationships reflecting the structure of the HTML document.

2. **Accessing Elements**:
   - You can access elements in the DOM using various methods such as `getElementById`, `getElementsByClassName`, `getElementsByTagName`, `querySelector`, and `querySelectorAll`.
   - Once you have a reference to an element, you can manipulate its properties, attributes, and content using JavaScript.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM Example</title>
</head>
<body>
  <div id="container">
    <h1>Hello, DOM!</h1>
    <p>This is an example of the DOM.</p>
    <button id="myButton">Click Me</button>
  </div>
  <script>
    // Accessing elements in the DOM
    const container = document.getElementById('container');
    const heading = container.querySelector('h1');
    const button = document.getElementById('myButton');

    // Manipulating elements
    heading.textContent = 'Hello, Updated DOM!';
    button.addEventListener('click', () => {
      alert('Button clicked!');
    });
  </script>
</body>
</html>
```

3. **Manipulating Content and Styles**:
   - You can manipulate the content of elements using properties like `textContent`, `innerHTML`, `innerText`, etc.
   - You can modify CSS styles using the `style` property or by adding/removing CSS classes.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM Manipulation Example</title>
  <style>
    .highlight {
      background-color: yellow;
    }
  </style>
</head>
<body>
  <p id="demo">This is a paragraph.</p>
  <button onclick="changeText()">Change Text</button>
  <button onclick="highlight()">Highlight Text</button>
  <script>
    function changeText() {
      const paragraph = document.getElementById('demo');
      paragraph.textContent = 'Text changed!';
    }

    function highlight() {
      const paragraph = document.getElementById('demo');
      paragraph.classList.toggle('highlight');
    }
  </script>
</body>
</html>
```

The examples above illustrate how JavaScript can interact with the DOM to dynamically modify the content, structure, and style of a web page based on user actions or other events. The DOM is a fundamental part of web development and enables the creation of interactive and dynamic web applications.

---