ES6

Advanced JavaScript (ECMAScript 6/2015)

Components of ES6

- Classes
- Arrow Functions
- Variables (let, const, var)
- Array Methods like .map()
- Destructuring
- Modules
- Ternary Operator
- Spread Operator

Object-Oriented Programming Concepts

- **Object** – An object is a real-time representation of any entity. According to Grady Brooch, every object is said to have 3 features –
- **State** – Described by the attributes of an object.
- **Behavior** – Describes how the object will act.
- **Identity** – A unique value that distinguishes an object from a set of similar such objects.
- **Class** – A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
- **Method** – Methods facilitate communication between objects.

Classes

- A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a **constructor()** method.

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}
const mycar = new Car("Ford");
```

Methods

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

const mycar = new Car("Ford");
mycar.present();
```

Setters Method

- A setter function is invoked when there is an attempt to set the value of a property. The set keyword is used to define a setter function

```
{set prop(val) { . . . }}
{set [expression](val) { . . . }}
```

Setters Method

```
class Student {
    constructor(rno,fname,lname){
      this.rno = rno
      this.fname = fname
      this.lname = lname
      console.log('inside constructor')
    }
    set rollno(newRollno){
      console.log("inside setter")
      this.rno = newRollno
    }
  }
  let s1 = new Student(1,'Tarkeshwar','Barua')
  console.log(s1)
  //setter is called
  s1.rollno = 201
  console.log(s1)
```

Getters Method

- A getter function is invoked when there is an attempt to fetch the value of a property. The get keyword is used to define a getter function.

```
{get prop() { ... } }
```

```
{get [expression]() { ... } }
```

- **prop** is the name of the property to bind to the given function.
- Use expressions as a property name to bind to the given function.

Getters Method

```
class Student {
    constructor(rno,fname,lname){
      this.rno = rno
      this.fname = fname
      this.lname = lname
      console.log('inside constructor')
    }
    get fullName(){
      console.log('inside getter')
      return this.fname + " - "+this.lname
    }
  }
  let s1 = new Student(1,'Tarkeshwar','Barua')
  console.log(s1)
  //getter is called
  console.log(s1.fullName)
```

Class Inheritance

```
class Car {
  constructor(name) {
    this.brand = name;
  }
  present() {
    return 'I have a ' + this.brand;
  }
}

class Model extends Car {
  constructor(name, mod) {
    super(name);
    this.model = mod;
  }
  show() {
      return this.present() + ', it is a ' + this.model
  }
}
const mycar = new Model("Ford", "Mustang");
mycar.show();`
```

Arrow Functions

Arrow functions allow us to write shorter function syntax

```
hello = function() {
  return "Hello World!";
}
hello = () => {
  return "Hello World!";
}
hello = () => "Hello World!";
hello = (val) => "Hello " + val;
hello = val => "Hello " + val;
```

Instanceof

- The instanceof operator returns true if the object belongs to the specified type.

```
class Person{ }
var obj = new Person()
var isPerson = obj instanceof Person;
console.log(" obj is an instance of Person " + isPerson);
```

‘this’ Inside Global Scope

```
let a = this;
console.log(a);  // Window {}


this.name = 'Tarkeshwar';
console.log(window.name); // Tarkeshwar
```

this Inside Function

```
function greet() {

    // this inside function
    // this refers to the global object
    console.log(this);
}

greet(); // Window {}
```

‘this’ inside constructor

```
//Class Person(){}
function Person() {

    this.name = 'Jack';
    console.log(this);

}

let person1 = new Person();
console.log(person1.name); // Jack
```

‘this’ inside object method

```
const person = {
    name : 'Jack',
    age: 25,

    // this inside method
    // this refers to the object itself
    greet() {
        console.log(this);
        console.log(this.name);
    }
}

person.greet();
```

‘this’ inside Arrow Function

```
const greet = {
    name: 'Jack',

    // method
    sayHi () {
        let hi = () => console.log(this.name);
        hi();
    }
}

greet.sayHi(); // Jack
```

‘this’ Inside Function with Strict Mode

```
'use strict';
this.name = 'Jack';
function greet() {


    // this refers to undefined
    console.log(this);
}
greet(); // undefined
```

```
'use strict';
this.name = 'Jack';

function greet() {
    console.log(this.name);
}

greet.call(this); // Jack
```

‘this’ inside Arrow Function

```
const person = {
    name : 'Jack',
    age: 25,

    // this inside method
    // this refers to the object itself
    greet() {
        console.log(this);
        console.log(this.age);

        // inner function
        let innerFunc = () => {

            // this refers to the global object
            console.log(this);
            console.log(this.age);
        }
        innerFunc();
    }
}
person.greet();
```

‘this’ inside inner function

```
const person = {
    name : 'Jack',
    age: 25,

    // this inside method
    // this refers to the object itself
    greet() {
        console.log(this);        // {name: "Jack", age ...}
        console.log(this.age);  // 25

        // inner function
        function innerFunc() {

            // this refers to the global object
            console.log(this);       // Window { ... }
            console.log(this.age);    // undefined

        }

        innerFunc();

    }
}

person.greet();
```

‘static’ Keyword

- The static keyword can be applied to functions in a class. Static members are referenced by the class name.

```
class StaticMem {
  static disp() {
     console.log("Static Function called")
  }
}
StaticMem.disp() //invoke the static method
```

Creating Variable

var x = 5.6; // Older approach

let x = 5.6; // let is the block scoped version of var, and is limited to the block (or expression) where it is defined.

const x = 5.6;   //const is a variable that once it has been created, its value can never change.

Array Methods

```
const myArray = ['apple', 'banana', 'orange'];
// The .map() method allows you to run a function on each item in the array, returning a new array as the result.
const myList = myArray.map((item) => `< p > ${ item } < /p>`);
console.log(myList);
```

Destructuring

```
const vehicles = ['mustang', 'f-150', 'expedition'];


// old way
const car = vehicles[0];
const truck = vehicles[1];
const suv = vehicles[2];


// ES6 Approach
const [car, truck, suv] = vehicles;
const [car, , suv] = vehicles;
```

```
function calculate(a, b) {
  const add = a + b;
  const subtract = a - b;
  const multiply = a * b;
  const divide = a / b;

  return [add, subtract, multiply, divide];
}

const [add, subtract, multiply, divide] = calculate(4, 7);
```

Destructuring Object

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red'
}

myVehicle(vehicleOne);

// old way
function myVehicle(vehicle) {
  const message = 'My ' + vehicle.type + ' is a ' + vehicle.color + ' ' + vehicle.brand + ' ' + vehicle.model + '.';
}
```

Destructuring Object

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red',
  registration: {
    city: 'Houston',
    state: 'Texas',
    country: 'USA'
  }
}

myVehicle(vehicleOne)

function myVehicle({ model, registration: { state } }) {
  const message = 'My ' + model + ' is registered in ' + state + '.';
}
```

Common JavaScript

```
function Welcome (props) {
return <h1> Hi {props.name} </h1>;
}
```

Promises

- To find a user by username from the user list returned by the getUsers() function, you can use the findUser()

```
function getUsers() {
    return [
        { username: 'tbarua1', email: 'tbarua1@gmail.com' },
        { username: 'tarkeshwar', email: 'tarkeshwar.b@regexsoftware.com' },
    ];
}

function findUser(username) {
    const users = getUsers();
    const user = users.find((user) => user.username === username);
    return user;
}
console.log(findUser('tbarua1'));
```

Promises

The **findUser()** function is synchronous and blocking. The **findUser()** function executes the **getUsers()** function to get a user array, calls the **find()** method on the users array to search for a user with a specific username, and returns the matched user.

The **getUsers()** function may access a database or call an API to get the user list. Therefore, the **getUsers()** function will have a delay.

To simulate the delay, you can use the setTimeout() function.

Promises

```
function getUsers1() {
    let users = [];

    // delay 1 second (1000ms)
    setTimeout(() => {
        users = [
            { username: 'tbarua1', email: 'tbarua1@gmail.com' },
            { username: 'tarkeshwar', email: 'tarkeshwar.b@regexsoftware.com' },
        ];
    }, 1000);

    return users;
}

function findUser(username) {
    const users = getUsers1();
    const user = users.find((user) => user.username === username);
    return user;
}
console.log(findUser('tbarua1'));
```

Promises Solution

```
function getUsers(callback) {
    setTimeout(() => {
        callback([
            { username: 'tbarua1', email: 'tbarua1@gmail.com' },
            { username: 'tarkeshwar', email: 'tarkeshwar.b@regexsoftware.com' },
        ]);
    }, 1000);
}

function findUser(username, callback) {
    getUsers((users) => {
        const user = users.find((user) => user.username === username);
        callback(user);
    });
}

findUser('tbarua1', console.log);
```

JavaScript Promises

- a promise is an object that encapsulates the result of an asynchronous operation.
- The promise constructor accepts two callback function **‘resolve’**, **‘reject’** that typically performs an asynchronous operation. This function is often referred to as an executor.
- Once a promise reaches either fulfilled or rejected state, it stays in that state and can’t go to another state.
- A promise object has three state:
  - Pending
  - Fulfilled with a value
  - Rejected for a reason

Creating a Promises

```
const promise = new Promise((resolve, reject) => {
  // contain an operation
  // ...

  // return the state
  if (success) {
    resolve(value);
  } else {
    reject(error);
  }
});
```

Promises Example

```
function getUsers() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve([
                { username: 'tbarua1', email: 'tbarua1@gmail.com' },
                { username: 'tarkeshwar', email: 'tarkeshwar.b@regexsoftware.com' },
            ]);
        }, 2000);
    });
}

function onFulfilled(users) {
    console.log(users);
}

const promise = getUsers();
promise.then(onFulfilled);
```

Promises with Arrow Function

```
function getUsers() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve([
                { username: 'tbarua1', email: 'tbarua1@gmail.com' },
                { username: 'tarkeshwar', email: 'tarkeshwar.b@regexsoftware.com' },
            ]);
        }, 1000);
    });
}

const promise = getUsers();

promise.then((users) => {
    console.log(users);
});
```

Promises Implementation

```
function getUsers() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve([
                { username: 'tbarua1', email: 'tbarua1@gmail.com' },
                { username: 'tarkeshwar', email: 'tarkeshwar.b@regexsoftware.com' },
            ]);
            reject('Failed to the user list');
        }, 2000);
    });
}

function onFulfilled(users) {
    console.log(users);
}

function onRejected(error) {
    console.log(error);
}
const promise = getUsers();
promise.then(onFulfilled, onRejected);
```

A practical JavaScript Promise example

https://www.javascripttutorial.net/sample/promise/api.json

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>JavaScript Promise Demo</title>
    <link href="css/style.css" rel="stylesheet">
</head>
<body>
    <div id="container">
        <div id="message"></div>
        <button id="btnGet">Get Message</button>
    </div>
    <script src="js/promise-demo.js">
    </script>
</body>
</html>
```

promise-demo.js

```
function load(url) {
  return new Promise(function (resolve, reject) {
    const request = new XMLHttpRequest();
    request.onreadystatechange = function () {
      if (this.readyState === 4 && this.status == 200) {
        resolve(this.response);
      } else {
        reject(this.status);
      }
    };
    request.open('GET', url, true);
    request.send();
  });
}

const url = 'https://www.javascripttutorial.net/sample/promise/api.json';
const btn = document.querySelector('#btnGet');
const msg = document.querySelector('#message');

btn.addEventListener('click', () => {
  load(URL)
    .then((response) => {
      const result = JSON.parse(response);
      msg.innerHTML = result.message;
    })
    .catch((error) => {
      msg.innerHTML = `Error getting the message, HTTP status: ${error}`;
    });
});
```

```
function load(url) {
  return new Promise(function (resolve, reject) {
    const request = new XMLHttpRequest();
    request.onreadystatechange = function () {
      if (this.readyState === 4 && this.status == 200) {
        resolve(this.response);
      } else {
        reject(this.status);
      }
    };
    request.open('GET', url, true);
    request.send();
  });
}
```

A practical JavaScript Promise example

```
const url = 'https://www.javascripttutorial.net/sample/promise/api.json';
const btn = document.querySelector('#btnGet');
const msg = document.querySelector('#message');

btn.addEventListener('click', () => {
  load(URL)
    .then((response) => {
      const result = JSON.parse(response);
      msg.innerHTML = result.message;
    })
    .catch((error) => {
      msg.innerHTML = `Error getting the message, HTTP status: ${error}`;
    });
});
```

Maps and Sets

Maps and Sets are new Data structure introduced

A map is an ordered collection of key-value pairs. Maps are similar to objects.

```
let map = new Map([iterable])
let map = new Map()

let andy = {ename:"Andrel"},
     varun = {ename:"Varun"},
     prijin = {ename:"Prijin"}
  let empJobs = new Map([
  [andy,'Software Architect'],
  [varun,'Developer']]
  );
  console.log(empJobs)
```

Maps and Sets

```
let daysMap = new Map();
daysMap.set('1', 'Monday');
daysMap.set('2', 'Tuesday');
daysMap.set('3', 'Wednesday');
console.log("Total Stored Objects are " + daysMap.size);
daysMap.delete("1");
let obj2 = daysMap.get("2")
console.log("Object Retrived " + obj2);
console.log("Total Stored Objects are " + daysMap.size);
```

Reduce Function

```
const ages = [21,18,42,40,64,63,34];
const maxAge = ages.reduce((max, age) => {
 console.log(`${age} > ${max} = ${age > max}`);
 if (age > max) {
 return age
 } else {
 return max
 }
}, 0)
console.log('maxAge', maxAge);
const max = ages.reduce(
 (max, value) => (value > max) ? value : max,
 0
)
```

WeakMap

- WeakMap is a small subset of map. Keys are weakly referenced, so it can be non-primitive only. If there are no reference to the object keys, it will be subject to garbage collection.
- not iterable
- every key is object type
- The WeakMap will allow garbage collection if the key has no reference.

```
new WeakMap([iterable])
```

WeakMap

```
let empMap = new WeakMap();
   // emp.set(10,'Sachin');
// Error as keys should be object
   let e1= {ename:'Kiran'},
      e2 = {ename:'Kannan'},
      e3 = {ename:'Mohtashim'}

   empMap.set(e1,1001);
   empMap.set(e2,1002);
   empMap.set(e3,1003);

   console.log(empMap)
   console.log(empMap.get(e2))
   console.log(empMap.has(e2))
   empMap.delete(e1)
```

Set

- A set is an unordered collection of unique values. This data structure can contain values of primitive and object types.

```
new Set([iterable])

new Set()


let names = new Set(['A','B','C','D']);

console.log(names)
```

WeakSet

- A Weakset holds objects weakly, that means object stored in a WeakSet are subject to garbage collection, if they are not referenced. WeakSets are not iterable and do not have the get method.

```
 let e1 = {ename:'A'}
   let e2 ={ename:'B'}
   let e3 ={ename:'C'}

   let emps = new WeakSet();
   emps.add(e1);
   emps.add(e2)
   .add(e3);

   console.log(emps)
   console.log(emps.has(e1))
   emps.delete(e1);
   console.log(emps)
```

Modules

- JavaScript modules allow you to break up your code into separate files.
- This makes it easier to maintain the code-base.
- ES Modules rely on the import and export statements.
- You can export a function or variable from any file.
- There are two types of exports: Named and Default.

Modules Export

```
// person.js  In-line individually, or all at once at the bottom
export const name = "Tarkeshwar"
export const age = 35
export { name, age }
```

```
// Default export only have one default export in a file.
const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + ' is ' + age + 'years old.';
};

export default message;
```

Modules Import

- Import can be achieve based on if they are named exports or default exports.
- Named exports must be destructured using curly braces. Default exports do not.

```
import { name, age } from "./person.js"; // named import
import message from "./message.js"; // default import
```

Ternary Operator

```
<h1 id="demo"></h1>
<script>
function renderApp() {
  document.getElementById("demo").innerHTML = "Welcome!";
}
function renderLogin() {
  document.getElementById("demo").innerHTML = "Please log in";
}
let authenticated = true;
authenticated ? renderApp() : renderLogin();
</script>
<p>Try changing the "authenticated" variable to false, and run the code to see what happens.</p>
```

Spread Operator

- The spread operator is three dots (...) that perform several different tasks. First, the spread operator allows us to combine the contents of arrays. For example, if we had two arrays, we could make a third array that combines the two arrays into one.

```
let first = ["One", "Two", "Three ", "Four"]
let second = ["Five", "Six", "Seven"]
let combine = [...first, ...second];
let combineReverse = [...first, ...second].reverse();
console.log(first)
console.log(second)
console.log(combine)
console.log(combine.join(', '))
let reverseValues = first.reverse();
console.log(reverseValues)
console.log(combineReverse)
```

Common JavaScript v/s ES6

```
function Welcome (props) {
return <h1> Hi {props.name} </h1>;
}
```

```
class Welcome extends React.Component {
render () {
return <h1> Hi {this.props.name} </h1>;
}
}
```

Common JavaScript

```
var photos = [ 'images / cat.jpg' , 'images / dog.jpg' , 'images / owl.jpg' ]
ReactDOM.render (
<App>
<Photos photos = photos />
<LastNews />
<Comments />
</App>,
document.getElementById ('root')
);
```

Computed Property

```
let propName = 'c';


const rank = {
  a: 1,
  b: 2,
  [propName]: 3,
};


console.log(rank.c); // 3
```

```
let namefull = 'fullName';
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    get[namefull]() {
        return `${this.firstName} ${this.lastName}`;
    }
    set firstName1(value) {
        this.firstName = value;
    }
    set lastName1(value) {
        this.lastName = value;
    }
}
let person = new Person('Tarkeshwar', 'Barua');
console.log(person.fullName);
person.firstName1 = "Dr";
person.lastName1 = "Barua";
console.log(person.fullName);
```

new.target Metaproperty

- detects whether a function or constructor was called using the new operator.
- The **new.target** consists of the new keyword, a dot, and target property. The **new.target** is available in all functions.
- In arrow functions, the **new.target** is the one that belongs to the surrounding function.
- Useful to inspect at runtime whether a function is being executed as a function or as a constructor.
- To determine a specific derived class that was called by using the new operator from within a parent class.

new.target Metaproperty

```
function Person(name) {
    if (!new.target) {
        throw "must use new operator with Person";
    }
    this.name = name;
}
let tarkesh = new Person('Tarkeshwar'); // new object from the Person function by using the new
operator
console.log(tarkesh.name); // Tarkeshwar
let tarkeshUsingFunction = Person('Dr. Tarkeshwar Barua');
console.log(tarkeshUsingFunction.name)
```

Append Child object

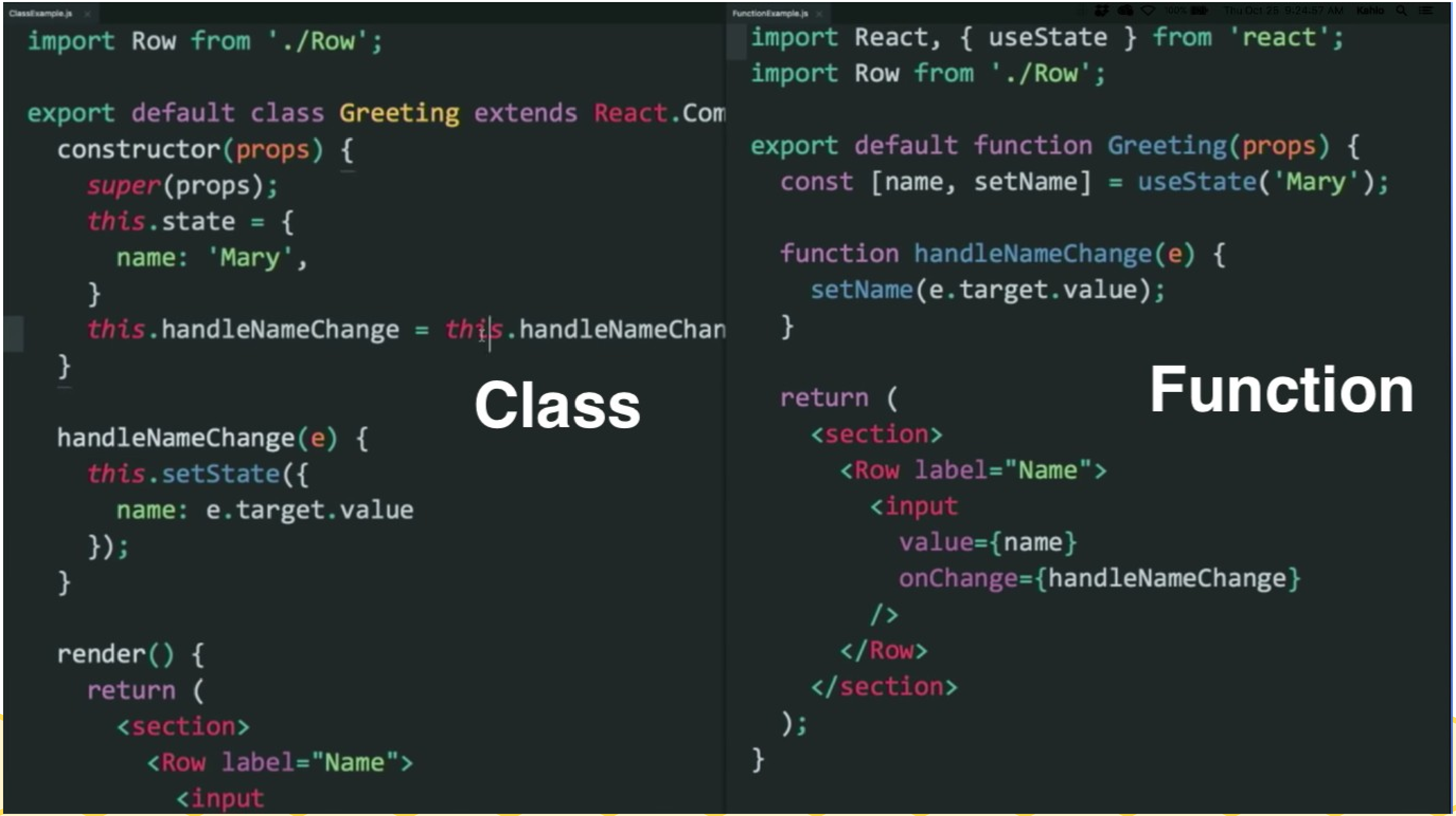
```
var div,
 container = document.getElementById('container')
for (var i=0; i<5; i++) {
 div = document.createElement('div')
 div.onclick = function() {
 alert('This is box #' + i)
 }
 container.appendChild(div)
}
```

Transpiling

- Most web browsers don’t support latest ES, and even those that do, don’t support every-thing.
- The only way to be sure that your latest ES code will work is to convert it to lowest ES code before running it in the browser.
- This process is called transpiling.
- One of the most popular tools for transpiling is Babel

Hooks

- Hooks were added to React in version 16.8.
- Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.
- Hooks generally replace class components, there are no plans to remove classes from React.
- Hooks allow us to "hook" into React features such as state and lifecycle methods.
- npm install react@next react-dom@next



```
import Row from './Row';

export default class Greeting extends React.Com
  constructor(props) {
    super(props);
    this.state = {
      name: 'Mary',
    }
    this.handleNameChange = this.handleNameChan
  }

  handleNameChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <section>
        <Row label="Name">
          <input
```

Class

```
import React, { useState } from 'react';
import Row from './Row';

export default function Greeting(props) {
  const [name, setName] = useState('Mary');

  function handleNameChange(e) {
    setName(e.target.value);
  }

  return (
    <section>
      <Row label="Name">
        <input
          value={name}
          onChange={handleNameChange}
        />
      </Row>
    </section>
  );
}
```

Function

Hooks

- You must import Hooks from react.
- Here we are using the useState Hook to keep track of the application state.
- State generally refers to application data or properties that need to be tracked.
- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional
- If you have stateful logic that needs to be reused in several components, you can build your own custom Hooks.

React useState Hook

- The React useState Hook allows us to track state in a function component.
- State generally refers to data or properties that need to be tracking in an application.
- To use the useState Hook, we first need to import it into our component.
- we are destructuring useState from react as it is a named export.
- We initialize our state by calling useState in our function component.
- useState accepts an initial state and returns two values:
  - The current state.
  - A function that updates the state.

React useState Hook

```
import React, { useState } from 'react';

function Demo1() {
  const [count, setCount] = useState(0);
  return (
    <div>
      Count: {count}
      <button onClick={() => setCount(0)}>Reset</button>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
}
export default Demo1;
```

React useState Hook

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}!</h1>
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

Update State in React

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

Updating Objects and Arrays in State

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
    color: "red"
  });

  const updateColor = () => {
    setCar(previousState => {
      return { ...previousState, color: "blue" }
    });
  }

  return (
    <>
      <h1>My {car.brand}</h1>
      <p>
        It is a {car.color} {car.model} from {car.year}.
      </p>
      <button
        type="button"
        onClick={updateColor}
      >Blue</button>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
    color: "red"
  });

  return (
    <>
      <h1>My {car.brand}</h1>
      <p>
        It is a {car.color} {car.model} from {car.year}.
      </p>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

React useEffect Hooks

- The useEffect Hook allows you to perform side effects in your components.
- Some examples of side effects are: fetching data, directly updating the DOM, and timers.
- useEffect accepts two arguments. The second argument is optional.
- useEffect(<function>, <dependency>)
- useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.
- This is not what we want. There are several ways to control when side effects run.
- We should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

React useEffect Hooks

```
function Demo2() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

React useEffect Hooks

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    let timer = setTimeout(() => {
    setCount((count) => count + 1);
  }, 1000);

  return () => clearTimeout(timer)
  }, []);

  return <h1>I've rendered {count} times!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

React useContext Hook

- React Context is a way to manage state globally.
- It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.
- State should be held by the highest parent component in the stack that requires access to the state.
- To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.
- To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

React useContext Hook

```
const TestContext = React.createContext();

function Display() {
  const value = useContext(TestContext);
  return <div>{value}, I am learning react hooks.</div>;
}

function App() {
  return (
    <TestContext.Provider value={"Tarkeshwar"}>
      <Display />
    </TestContext.Provider>
  );
}
```

React useContext Hook

```
import { useState, createContext, useContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
  );
}
```

```
function Component4() {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 />
    </>
  );
}

function Component5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

React useRef Hook

- The useRef Hook allows you to persist values between renders.
- It can be used to store a mutable value that does not cause a re-render when updated.
- It can be used to access a DOM element directly.
- If we tried to count how many times our application renders using the useState Hook, we would be caught in an infinite loop since this Hook itself causes a re-render. To avoid this, we can use the useRef Hook.

React useRef Hook

```
function App() {
  let [name, setName] = useState("Nate");

  let nameRef = useRef();

  const submitButton = () => {
    setName(nameRef.current.value);
  };

  return (
    <div className="App">
      <p>{name}</p>

      <div>
        <input ref={nameRef} type="text" />
        <button type="button" onClick={submitButton}>Submit</button>
      </div>
    </div>
  );
}
```

React useRef Hook

```
import { useState, useEffect, useRef } from "react";
import ReactDOM from "react-dom/client";

function App() {
  const [inputValue, setInputValue] = useState("");
  const previousInputValue = useRef("");

  useEffect(() => {
    previousInputValue.current = inputValue;
  }, [inputValue]);

  return (
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <h2>Current Value: {inputValue}</h2>
      <h2>Previous Value: {previousInputValue.current}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

React useReducer Hook

- The useReducer Hook is similar to the useState Hook. It allows for custom state logic.
- If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.
- The useReducer Hook accepts two arguments. **useReducer(<reducer>, <initialState>)**
- The reducer function contains your custom state logic and the initialStatecan be a simple value but generally will contain an object.
- The useReducer Hook returns the current stateand a dispatchmethod.

React useReducer Hook

```
import { useReducer } from "react";
import ReactDOM from "react-dom/client";

const initialTodos = [
  {
    id: 1,
    title: "Todo 1",
    complete: false,
  },
  {
    id: 2,
    title: "Todo 2",
    complete: false,
  },
];

const reducer = (state, action) => {
  switch (action.type) {
    case "COMPLETE":
      return state.map((todo) => {
        if (todo.id === action.id) {
          return { ...todo, complete: !todo.complete };
        } else {
          return todo;
        }
      });
    default:
      return state;
  }
};
```

```
function Todos() {
  const [todos, dispatch] = useReducer(reducer, initialTodos);

  const handleComplete = (todo) => {
    dispatch({ type: "COMPLETE", id: todo.id });
  };

  return (
    <>
      {todos.map((todo) => (
        <div key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleComplete(todo)}
            />
            {todo.title}
          </label>
        </div>
      ))}
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Todos />);
```

React useCallback Hook

- The React useCallback Hook returns a memoized callback function.
- Think of memoization as caching a value so that it does not need to be recalculated.
- This allows us to isolate resource intensive functions so that they will not automatically run on every render.
- The useCallback Hook only runs when one of its dependencies update.
- This can improve performance.
- The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function.

React useMemo Hook

- The React useMemo Hook returns a memoized value.
- Think of memoization as caching a value so that it does not need to be recalculated.
- The useMemo Hook only runs when one of its dependencies update.
- This can improve performance.
- The useMemo and useCallback Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function. You can learn more about useCallback in the useCallback chapter.
- The useMemo Hook can be used to keep expensive, resource intensive functions from needlessly running.

React Custom Hooks

- Hooks are reusable functions.
- When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.
- Custom Hooks start with "use". Example: useFetch.