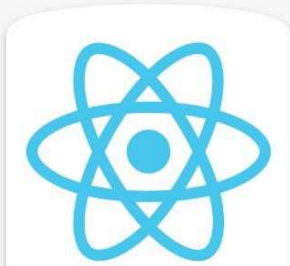


3

React JS



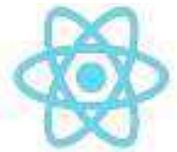
React JS

The Recap Need Before Enter React



What is React?

- React is a JavaScript library for building user interfaces, that is maintained by Facebook as an Open Source project under the MIT license
- “React is a JavaScript library for building User Interfaces”
 - React applications run in the browser.
 - This facilitates speedier responses to user actions.
 - It is maintained by Facebook and community
- React is a library for building composable user interfaces.
- It has to be emphasised that React is **NOT** a framework; it is a library.
- User Interfaces as in web pages can be split into components.
- Many developers tend to use react as the V in MVC.
- React abstracts the DOM providing a simpler programming model and better performance.



Why React?

- React is a JavaScript library used for building reusable UI components.
- UI state is cumbersome to handle using plain JavaScript.
 - DOM elements need to be manually targeted in order to change the structure of the HTML.
 - Complex web applications that need elements to be added/removed tend to become tedious using just JavaScript/jQuery.
- React helps developers focus on business logic
 - Creators have developed react using patterns and best practices thus ensuring optimal code base for us to start with.
- React enjoys a huge ecosystem and vibrant community support.



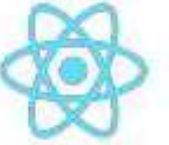
React – ES6

- React applications typically are built with the latest version of JavaScript.
- The newer features allow for the creation of cleaner and robust React applications.
- React itself employs a lot of these newer JavaScript features.
- JavaScript as a language is evolving fairly rapidly.



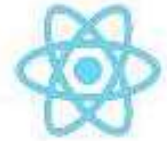
Classes

- Classes are a new feature in ES6, used to describe the blueprint of an object
 - They perceive the transformation of ECMAScript's prototypal inheritance model to a more traditional class-based language.
- ES6 classes offer a much nicer, cleaner and clearer syntax to create objects and deal with inheritance.
- The class syntax is not introducing a new object-oriented inheritance model to JavaScript.



ES 2015

- ES5 has been around since 2009.
 - It is supported by most of the popular browsers.
- In June 2015 a new specification of the JavaScript standard was approved that contains a lot of new features.
- It is called ECMAScript 2015 or also called ES6 as it is the 6th edition of the standard.
 - ECMAScript is the official name of the JavaScript language.
- Existing browsers don't support most of the features of ES6 yet.



ES 2015...

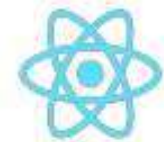
- Feature support across browsers varies widely.
- Are we expected to wait a few years and commence using ES6 after browsers start offering support?
 - <http://kangax.github.io/compat-table/es6/>
- Fortunately not!
- There are tools that can convert ES6 code into ES5 code.
- We write code using the new useful features of ES6 and generate ES5 code that will work in most of the current browsers.



ES6 / ES 2015

ES6 brings a lot of new features, some of which include:

- Classes
- Arrow Functions
- Template Strings
- Inheritance
- Constants and Block Scoped Variables
- Modules



Classes...

```
class Shape {  
  constructor(type) {  
    this.type = type;  
  }  
  getType() {  
    return this.type;  
  }  
}
```

- Use the **class** keyword to declare a class.
- **constructor** is a special method for creating and initializing an object.
 - There can only be one special method with the name **constructor**



Classes...

- The **static** keyword defines a static method for a class.
- Static methods are called without instantiating their class and are not callable when the class is instantiated.
- Static methods are often used to create utility functions for an application.

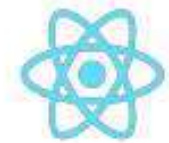
```
class Shape {  
  constructor(type){  
    this.type = type;  
  }  
  static getClassName(){  
    const name = 'Shape';  
    return name;  
  }  
}  
  
console.log(Shape.getClassName());
```



Subclassing

- The **extends** keyword is used in class declarations to create a class as a child of another class.
- The **super** keyword is used to call functions on an object's parent.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  speak() {  
    console.log(this.name + ' makes a noise.');  }  
}  
  
class Dog extends Animal {  
  speak() {  
    super.speak();  
    console.log(this.name + ' barks.');  }  
}  
  
var d = new Dog('Mitzie');  
d.speak();
```

this revisited

- In JavaScript **this** keyword is used to refer to the instance of the class.

```
var shape = {
  name: 'square',
  say: function(){
    console.log('This is say(): ' + this.name);

    setTimeout(function(){
      console.log('Inside setTimeout(): '
        + this.name);
    }, 2000);
  }
};

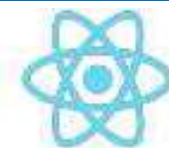
shape.say();
```

- The **this.name** is empty when accessed within **setTimeout**.



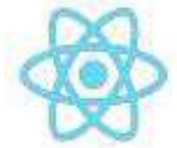
Arrow functions

- ES6 offers new feature for dealing with **this**, “arrow functions” =>
 - Also known as *fat arrow*
- Some of the motivators for a *fat arrow* are:
 - One does not need to specify **function**
 - It lexically captures the meaning of **this**
- The fat arrow notation can be used to define anonymous functions in a simpler way.
- Helps provide the context to **this**



Arrow functions

```
var shape = {  
  name: 'square',  
  say: function(){  
    console.log('This is say(): ' + this.name);  
  
    setTimeout(() => {  
      console.log('Inside setTimeout(): ' + this.name);  
    }, 2000);  
  }  
};  
  
shape.say();
```



Arrow functions...

- Arrow functions do not set a local copy of **this**, **arguments** etc.
- When **this** is used in an arrow function, JavaScript uses the **this** from the outer scope.
- If **this** *should* be the calling context, do not use the arrow function.



let

- **var** variables in JavaScript are *function scoped*.
- This is different from many other languages(Java, C#) where variables are *block scoped*.
- In ES5 JavaScript and earlier, **var** variables are scoped to the function and they can “see” outside their functions into the outer context.

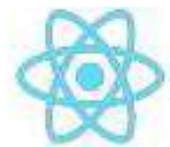
```
var foo = 123;  
if(true){  
    var foo = 456;  
}  
console.log(foo); //456
```



let...

- ES6 introduces the **let** keyword to allow defining variables with true *block scope*.
- Use of the **let** instead of **var** gives a true unique element disconnected from what is defined outside the scope.

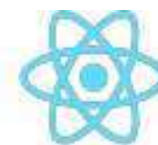
```
let foo = 123;  
if(true){  
  let foo = 456;  
}  
console.log(foo); //123
```



let...

- Functions create a new variable scope in JavaScript as expected.

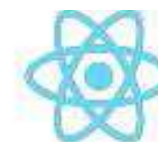
```
var num = 123;  
function numbers() {  
    var num = 456;  
}  
  
numbers();  
console.log(num);    //123
```



let...

- Usage of **let** helps reduce errors in loops.
- **let** is extremely useful to have for the vast majority of the code.
- It helps decrease the chance of a programming oversight.

```
var index = 0;
var myArray = [1,2,3];
for(let index = 0; index < myArray.length;
    index++){
    console.log(myArray[index]);
}
console.log(index);    //0
```

const

- **const** is a welcome addition in ES6.
- It allows immutable variables.
- To use **const**, replace **var** with **const**

```
const num = 123;
```

- **const** is a good practice for both readability and maintainability.
- **const** declarations must be initialized

```
const foo; //ERROR
```



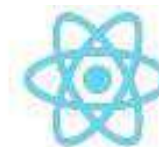
const

- A **const** is block scoped like the **let**
- A **const** works with object literals as well.

```
const foo = { bar : 123 };  
foo = { bar : 456 }; //ERROR
```

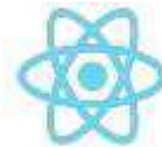
- **const** allows sub properties of objects to be mutated

```
const foo = { bar : 123 };  
foo.bar = 456; //allowed  
console.log(foo); // { bar : 456 }
```



Template Strings

- In traditional JavaScript, text that is enclosed within matching “ or ‘ marks is considered a string.
- Text within double or single quotes can only be on one line.
- There was no way to insert data into these strings.
- If there was a need it would have required concatenation that looked complex and not so elegant.
- ES6 introduces a new type of string literal that is marked with back ticks (`)



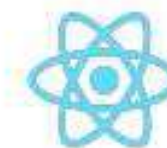
Template Strings

- The motivators for Template strings include
 - Multiline Strings
 - String Interpolation
- Multiline Strings

```
var desc = 'Do not give up \  
\n Do not bow down';
```

- with Template Strings

```
var desc = `Do not give up  
Do not bow down`;
```

Template Strings...

- String Interpolation

```
var lines = 'Do not give up';  
var html = '<div>' + lines + '</div>';
```

- with Template Strings

```
var lines = 'Do not give up';  
var html = `<div>${lines}</div>`;
```

- Any placeholder inside the interpolation `${ }` is treated as a JavaScript expression and evaluated.

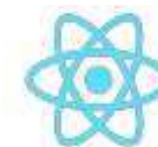


Spread and Rest Operators

- These new operators are represented by ...
- *Spread* operator is used to split up array elements OR object properties

```
let myData = ['A', 'B', 'C'];
let myNewData = [...myData, 'D']
console.log(myNewData); // ['A', 'B', 'C', 'D']

let user = { name : 'Lakshman' }
let userData = {
  ...user,
  email : 'laks@acme.org'
}
console.log(userData); // [object Object] {email:"laks@acme.org",
                        name: "Lakshman"}
```



Spread and Rest Operators...

- **Rest** operator is used to merge a list of function arguments into an array.
 - Array methods can be applied to the arguments list.

```
function getData(...args) {  
  return(args.filter(el => el > 3).sort())  
}  
console.log(getData(3,1,4,8,6))    //    [4, 6, 8]
```



Destructuring

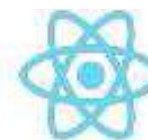
- *Destructuring* allows the extraction of array elements or object properties and store them in variables.
 - *Spread* operator takes out all array elements or all properties and distributes them in a new array or object.
- *Destructuring* allows pulling out single array elements or properties and store them in variables.

```
let myChars = ['A', 'B', 'C'];  
[char1, char2] = myChars;  
console.log(char1, char2);    // 'A' 'B'  
[char1, , char3] = myChars;  
console.log(char1, char3);    // 'A' 'C'
```




Modules

- JavaScript has always had problem with namespaces.
 - Variables and functions can end up in the global namespace if not cautious.
- JavaScript lacks built-in features specific to code organization.
- Modules help resolve these issues.



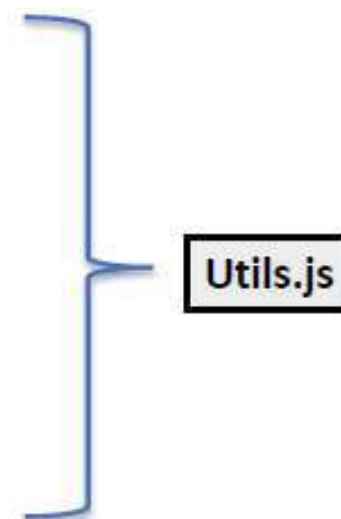
Modules in ES6

- In ES6 each module is defined in its own file.
- The functions and variables defined in a module are not visible outside unless explicitly exported.
- A module can be coded in a way such that only those values that need to be accessed by other parts of the application need be exported.
- Modules in ES6 are declarative in nature.
- To export variables from a module use **export**.
- To consume the exported variables in a different module use **import**.



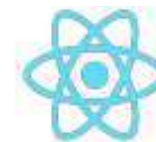
Working with Modules

```
function getRandom(){  
    return Math.random();  
}  
  
function add(n1, n2){  
    return(n1+n2);  
}  
  
export {getRandom, add}
```



- The **export** keyword is used to export the two functions.
- The exported functions can be renamed while exporting

```
export {getRandom as random, add as sum}
```



Working with Modules...

```
import { getRandom, add } from 'Utils';
```

```
console.log(getRandom());
```

```
console.log(add(1,2));
```

useUtils.js



- This imports the exported values from the module 'Utils'.
- A single value can be imported as well

```
import { add } from 'utils';
```