10. React components (class components) and props

10.1 Creating objects in JS using constructor function

- **React component (revision)**: A component is one of the core building blocks of React. They serve the same purpose as JavaScript functions, but work independently and return HTML via a render() function.
- **React class components**: The class components are a little more complex than the functional components in React. We can use JavaScript ES6 classes to create class-based components in React. When creating a React component, the component's name must start with an upper-case letter. The component has to include the “extends” and “React.Component” statement. This is to allow our component to inherit properties from the default class called “Component” that contains the different React Lifecylce Methods.
- **Understanding OOP, class and object before discussing class components in React**: Before explaining class components, let us revise our knowledge of objects and try to understand what is meant by class, inheritance and the 2 different ways of creating objects in JavaScript.
- **Object Oriented Programming (OOP)**: It is a way we write applications/programs based on classes and objects. In OOP everything is considered as an object. OOP is used to structure applications/programs into simple and reusable blueprint code (called classes) which are in turn used to create individual instances of objects. We can mention C++, Java, Python and now JavaScript as OOP languages.
    - **Note**: OOP focuses on the objects that developers want to manipulate than the logic required to manipulate objects.
- **Classes in OOP**: Classes are blueprints used to create more specific, concrete objects that share same attributes inherited from the blueprint class. Classes often represent broad categories, like Person and define attributes like age and gender. Classes can also have functions as their attributes called methods
- **Objects in JavaScript:** In JavaScript everything (including array) is an object except for the primitive data types (Boolean, number and string, and undefined). Thus, if you master objects in JavaScript, think as if you mastered a lot.

- **Ways to initialize/create an object in JS (revision)**: There are different ways to create an object in JavaScript
  - **Initializing an object literal**: The object literal initializes the object with curly brackets. **Example**: const objectLiteral = {};
    - **Note**: Creating an object using object literal method is very limiting because it creates only one object at a time.

```
let myPerson= {
          name: “Abebe”,
          Age: 13
};
```

  - **Initializing object constructor function with new object**: The object constructor initializes the object with the new keyword. Constructor function is the blueprint that creates a new object and set values for any existing object properties.. Unlike object literal, constructor function allows us to create as many objects (of similar type) as we want. **Example**: let objectConstuctor = new Object();
    - **Note**: To separate a regular function from an object constructor function, we always have to use uppercase when giving it a name.
    - **Note**: In a constructor function this does not have a value. It is a substitute for the new object. The value of this will become the new object when a new object is created.
    - **Note**: Below, function Person() is an object constructor function that we can use to create objects of the same type by calling the constructor function with the new keyword.

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

// Creating 2 objects using the same constructor above

const myPerson = new Person (“Hana”, 43);

const herPerson = new Person (“Kibrom”, 43);

- **The “this” keyword in JS**: the keyword has different meanings in JavaScript:
  - **In object constructor function**: “this” refers to the future object that will be created with the constructor. In the above example, if you console.log(this) within the conctructor function, it will print the two objects (myPerson and herPerson)
  - **Alone or when in a function**: “this” refers to the global Object
  - **In events**: “this” refers to the element receiving the event

10.2 Traditional way of property inheritance in JS (using prototype object)

- There are four pillars of OOP: Inheritance, encapsulation, abstraction and polymorphism
    - **Data Encapsulation**: It is the process of containing/ wrapping up data in an object and exposing only selected information. In short, it is a protective shield that prevents the data from being accessed by the code outside this shield.
    - **Data Abstraction**: Data Abstraction is the process of exposing only the required characteristics of an object (such as high-level public methods accessible to all classes in your app) and ignoring the irrelevant details.
    - **Data Polymorphism**: Polymorphism is derived from two words; poly and morphism which means many and variance of form. In programming, polymorphism is defined as the ability of an object, varible or function to take on different forms.
        - Let us imagine that we will need to write a program that will calculate the perimeter of shapes. Let us define shape as class and then define the different shapes as sub-classes of the base class shapes. Therefore, we will have a subclass circle, square and rectangle which will all have their methods and different parameters. What we have done above is polymorphism. Our shape class now has different forms and characteristics like circle, square and rectangle. This means, any child class object can take any form of a class in its parent hierarchy and also take the reference from itself as well.
        - In a programming language exhibiting polymorphism, class objects belonging to the same hierarchical tree (inherited from a common parent class) may have functions with the same name, but with different behaviors.
    - **Data Inheritance**: It is the process by which child classes inherit data and behaviors from parent class. We will discuss this concept in detail below.

- **Inheritance**: In OOP languages, inheritance refers to a class's ability to access attributes and methods/properties from another class. There are two types of inheritance:
  - **Class inheritance**: Here, properties/methods from base class get copied into derived class, in short, classes as blueprints for objects. PHP, Python, and Java are class-based languages, which instead define classes as blueprints for objects.
  - **Prototypal inheritance**: Every object in JavaScript has an internal property called Prototype. To find the [[Prototype]] of this newly created object, we will use the getPrototypeOf() method.
  - **Reusability**: Reusability: Inheritance supports the concept of "reusability", i.e., when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
- **Prototypal inheritance in JavaScript**: Before ES6, a JavaScript object could "inherit" properties from another object via a prototype linkage. **Prototype**: A built-in property every function/object in JavaScript has by default.
- **Why use prototype in JavaScript**: JavaScript uses prototype to add new properties/methods an object constructor function. The new properties will be shared by all objects that will later be created using this constructor. Look at the example below to understand why we use prototype
  - **Example of adding grade prototype to constructor function**:

```
function Student(name, gender) {
        this.name = name,
}
```

// **Adding "grade" method/prototype for future objects to inherit the method**

```
Student.prototype.grade = 5;
var studentOne = new Student();
console.log(studentOne.grade); // 5
var studentTwo = new Student();
console.log(studentTwo.grade); // returns 5
```

- **Example of adding a method prototype to constructor function**:

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

// **Adding showAge method/prototype to constructor so future objects can inherit the method**

```
Person.prototype.showAge = function () {
        return this.age;
};
const personOne = new Person("Chala", 33);
console.log(personOne.showAge()); // returns 33
```

- **Example of linking 2 constructor functions**: Assume you have a constructor called Person that has 2 properties, name and age. This constructor has a third property in a form of function (called showAge function) that is added to it as a prototype. Assume there is another constructor called Student that inherits all properties of our Person constructor. The Student constructor has also its own property called grade. How can we use the showAge method from Person constructor to get the age of any student object constructed from Student constructor? Look at the following code

```
function Person(name, age) {

        this.name = name;

        this.age = age;

}

// Adding showAge method/prototype to Person

Person.prototype.showAge = function () {

        return this.age;

};

function Student(grade, name, age) {

        this.grade = grade;

        Person.call(this, name, age);

}

//Make Student inherit Person's properties

console.log(Student.prototype);

Student.prototype = Object.create(Person.prototype);

console.log(Student.prototype);// returns 33
```

    - **The Object.create() method**: It creates a new object, using an existing object (using the Person object in our case) as the prototype of the newly created object (the Student object).

- **JavaScript built-in constructors**: JavaScript also has built-in constructors like new Object() and new Array(). This means that we can create JavaScript string and number, array as objects.

```
let a = new Object();    // A new Object object

let a = new Array();    // A new Object object

let b = new String();    // A new String object
```

- We have seen above how property inheritance from one object to another used to happen in JavaScript prior to ES6. This way of writing Object Oriented JavaScript works. But as with other things, ES6 gives us a better way of doing it. Below, let us look at how JavaScript uses class-based inheritance in ES6.

10.3 ES6 way of property inheritance in JS (concept of class)

- Just like other OOP languages, ES6 implements a class-based object-oriented approach. JS developers wanted to use the class syntax to create object because most object-oriented programming languages use class-based approach to handle inheritance.
- **Classes in JavaScript**: In ES6 we can create classes. Classes, just like ES5 constructor functions, are templates used as a blueprint to create individual objects. Classes can contain functions called methods available to instance objects of a class.
  - **Note**: When we created components in our first React lectures, we advised you not to use the keyword "class" in your JSX because we did not want React to confuse the HTML attribute "class" with JavaScript "class"
- **Class declaration/expression**: One way to define a class is using a class declaration. To declare a class, you use the class keyword followed by the name of the class
  - **Syntax for ES6 class declaration**:

    ```
    class Class_name {
            }
    ```

  - **Syntax for ES6 class expression**:

    ```
            var var_name = new Class_name {
            }
    ```

  - **A class definition can include the following**:
    - **Constructors**: Responsible for allocating memory for the objects of the class.
    - **Functions**: They are also at times referred to as methods.
- **Creating objects (instances of a class)**: To create an instance of the class, use the new keyword followed by the class name.
  - **Syntax to create instance of a class**:

    ```
            const object_name= new Class_name()
    ```

- **Example of class declaration and creating an instance (object) of the class:**
    - **Declaring a class**

```
class Person {
        constructor(name, age) {
        this.name = name;
        this.age = age;
        }
        getName() {
                return this.name + " , " + this.age;
                // return `${this.greeting}, ${this.name}!`;
        }
}
```

    - **Creating an instance of the above class**

```
const personOne = new Person("Almaz", 24);
console.log(personOne.getName()); // prints Almaz , 24
```

- **Function declarations vs class declarations and the concept of hoisting**:
  - **Function declarations**: Functions can be called in code that appears before they are defined.
    - **Example**: Let us see below how we can access our constructor function by calling it before it is declared.

```
const personOne = new Person("Selam", 34);

console.log(personOne.age); // prints 33

function Person(name, age) {

        this.name = name;

}
```

  - **Class declarations**: Classes must be defined before they can be used to construct an object/class. In short, you cannot access the class before it is initialized. **Example**: Code like the following will throw a ReferenceError

```
const personOne = new Person("Selam", 34);// ReferenceError

console.log(personOne.age);

class Person {

        constructor(name, age) {

                this.name = name;

        }

}
```

  - **JavaScript hoisting**: process whereby the interpreter appears to move the declaration of functions, variables or classes to the top of their scope, prior to execution of the code. It is because our JavaScript interpret hoisted (moved up) the above function that we were able call it in code before it was declared.

10.4 ES6 way of property inheritance in JS (concept of class inheritance)

- **Concept of inheritance in JavaScript**: We have seen in the previous sections that ES6 supports the concept of inheritance. To revise what discussed, inheritance is the ability of a program to create new entities (example child classes) from an existing entity (example parent classes).
- **Using the "extends" Keyword to create a new class from an existing class**: In JavaScript, to create a class inheritance, use the extends keyword. In other words, we use the "extends" keyword if we have a class already and we want to create a subclass/child class that inherits all properties of the first class
    - **Parent/super class**: The class that is extended to create newer classes is called the parent class/super class.
    - **Child/sub class**: The newly created classes are called the child/sub classes.
        - **Note**: Child classes inherit all properties and methods except constructors from the parent class.
    - **Syntax for class inheritance**:

```
class ChildClassName extends ParentClassName {}
```

- **Single vs multi-level inheritance in JavaScript**: Inheritance in JavaScript can be single (where every class extends from one parent class) or multi-level (when one child class extends the parent /super class and when a new child extends the previous child class and when the latest class inherits properties from the super/parent class indirectly). Let us explain this by examples below.

- **Example of single class inheritance**:

  //parent class

```
class Person {
    constructor(gender) {
        this.gender = gender;
    }
}
```

// child class inheriting properties of Person by extension

```
class Student extends Person {
    showGender() {
        return this.gender;
    }
}
```

//instance of our child class

```
const studentOne = new Student("Female");
```

// Child class accessing the gender property from parent class

```
console.log(studentOne.showGender()); // prints Female
```

- **Example of multi-level class inheritance**: Below, look how RankedStudent class inherits the “gender’ property from the Person class, our super class

```
class Person {
    constructor(gender, rank) {
        this.gender = gender;
        this.rank = rank;
    }
}
```

// child class inheriting properties of Person by extension

```
class Student extends Person {
    showGender() {
        return this.gender;
    }
}
class RankedStudent extends Student {
    showRank() {
        return this.rank;
    }
}
```

//instance of our RankedStudent class

```
const RankedStudentOne = new RankedStudent("Female", "firstRank");
```

// Child class accessing the rank property from super/parent class

```
console.log(RankedStudentOne.showRank()); // prints firstRank
```

- **Class inheritance and method overriding**: When a child class uses the same method name as that of the super class, the child class method will redefine the super class' method. Look at the example below

```
class ParentClass {
        doPrint() {
                console.log("Printing from parent class");
        }
}
class ChildClass extends ParentClass {
        doPrint() {
                console.log("Printing from child class");
        }
}
const ObjectOne = new ChildClass();
ObjectOne.doPrint(); //Printing from child class
```

  - **Question**: What do we do if we want to avoid a child class from overriding a super class's method? In other way, how can a child class invoke/execute functions found in the parent class in ES6? We use the "super" keyword
- **The Super Keyword**: ES6 enables a child class to access and call functions of a parent class using the "super" keyword. The super keyword is used to refer to the immediate parent of a class.
  - **Note**: "super" is valid in both classes and object literal methods/functions
  - **Syntax**:
    - super(arguments) // calls the parent constructor from inside the child class's constructor
    - super.parentMethod(); // calls a parent method from the child's method

- **Example (calling parent class's constructor by using "super()" within the child class constructor to access the parent's brand property)**

```
class CarBrand {
constructor(brand) {
this.brand = brand;
}
sayCarBrand() {
return this.brand;
}
        }


class CarYear extends CarBrand {
        constructor(brand, year) {
        super(brand);
        this.year = year;
        }
}
const carOne = new CarYear("Honda", 2013);
console.log(carOne.sayCarBrand());// prints Honda
```

- **Example (calling parent class’s method by using “super()” within the child class method to access the parent’s method)**

```
class PrinterClassParent {
        doPrint() {
                console.log("Printing from parent ");
        }
}
class PrinterClassChild extends PrinterClassParent {
        doPrint() {
                super.doPrint();
                return "Printing from child class";
        }
}
var printerOne = new PrinterClassChild();
printerOne.doPrint();  // prints both “Printing from parent "
```

10.5 Array and object destructuring

- **Destructuring**: We will need to understand the concept of array and object destructuring as we will use destructuring tehnique when we pass data using props and states.
- **Destructuring in JavaScript**: It is a JavaScript feature that allows us to extract multiple pieces of data from an array or object and assign them to their own variables. Destructuring was introduced in ES6.
    - **Destructuring explained with real life example**: To illustrate destructuring, it is easier to understand the analogy of how we make a sandwich from items in our fridge. If you want to make a sandwich, there is no need to take out everything you have in the fridge. You only take out the items you would like to use on your sandwich, such as cheese, vegetables and choice of your meat. Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these. Destructuring makes it easy to extract only what is needed.
- **Array Destructuring**:
    - **Assigning array items before ES6**: Please look how repetitive it is to assign each item of the array.
        - **Example**:

```
const students = ["Sisay", "Hunde", "Sara"];
                const firstStudent = students[0];
                const secondStudent = students[1];
                const thirdStudent = students[2];
                console.log(firstStudent); // prints Sisay
                console.log(secondStudent); // prints Hunde
                console.log(thirdStudent); // prints Sara
```

- **Assigning array items with destructuring (after ES6)**: You use an array literal on the left-hand side of the assignment. Destructuring will take each variable on the array literal on the left-hand side and maps it to the same element at the same index in the array. **Note**: When destructuring arrays, the order that variables are declared is important. Look how the above array assigning is longer and the assigning with destructuring below has shorter and clearer code
  - **Example**:

```
const students = ["Sisay", "Hunde", "Sara"];
        const [firstStudent, secondStudent, thirdStudent] = students;
        console.log(firstStudent); // prints Sisay
        console.log(secondStudent); // prints Hunde
        console.log(thirdStudent); // prints Sara
```

  - **Note 1**: If the number of variables passed to the destructuring array literals are more than the elements in the array, then the variables which aren't mapped to any element in the array return undefined. **Example**:

```
const students = ["Sisay", "Hunde"];
        const [firstStudent, secondStudent, thirdStudent] = students;
        console.log(firstStudent); // prints Sisay
        console.log(secondStudent); // prints Hunde
        console.log(thirdStudent); // prints undefined
```

- **Note 2**: If the number of variables passed to the destructuring array literals are lesser than the elements in the array, the elements without variables to be mapped to are just left and no errors here. **Example**:

  ```
  const students = ["Sisay", "Hunde", "Sara"];
          const [firstStudent, secondStudent] = students;
          console.log(firstStudent); // prints Sisay
          console.log(secondStudent); // prints Hunde
  ```

- **Note 3**: When destructuring arrays, if you want only few of the array elements, you cans simply leave out the one you want to leave but keep a comma as a placeholder for that element. **Example**:

```
const students = ["Sisay", "Hunde", "Sara"];
                const [firstStudent, , thirdStudent] = students;
                console.log(firstStudent); // prints Sisay
                console.log(thirdStudent); // prints Sara
```

- **Destructuring an array retrurned from a function**: Destructuring comes in handy when a function returns an array. Prior to ES6, there was no direct way to assign the elements of the returned array to multiple variables such as firstStudent, secondStudent and thirdStudent. Fortunately, starting from ES6, you can use the destructing assignment as follows:

- **Example**:

```
function calculate(a, b) {
const ad = a + b;
const subt = a - b;
const mult = a * b;
const div = a / b;
                        return [ad, subt, mult, div];
                }
                const [ad, subt, mult, div] = calculate(24, 8);
                console.log(calculate(24, 8)) // prints [32, 16, 192, 3]
```

- **Example (rest syntax for array assignment using destructuring)**: Below the variables ad receives values of the first element of the returned array. And the args variable receives all the remaining arguments, which are the last three elements of the returned array.

```
function calculate(a, b) {
const ad = a + b;
const subt = a - b;
const mult = a * b;
const div = a / b;
                        return [ad, subt, mult, div];
                }
                const [ad, ...args] = calculate();
                console.log(calculate(24, 8)); // prints [32, 16, 192, 3]
```

- **Nested array destructuring**: In the example below, because the third element of the returned array is another array, you need to use the nested array destructuring syntax to destructure it. The same for the array containing "Yellow", "Gray" elements. **Example**:

  ```
  function getProfile() {
      return ["John", "Doe", ["Red", "Green", "Blue", ["Yellow",
              "Gray"]]];
      }
      let [pro1, pro2, [pro3, pro4, pro5, [pro6, pro7]]] = getProfile();
      console.log(pro3); // prints red
      console.log(pro7); // prints Gray
  ```

- **Swapping values of variables using destructuring**: The array destructuring makes it easy to swap values of variables without using a temporary variable. **Example**:
  - **Swapping variables using temporary variable**: Swapping variables using a temporary variable is classic. As the name suggests, this approach requires an additional temporary variable.

    ```
    let alem = 10,
    balcha = 20;
    console.log(alem); // 10
    console.log(balcha); // 20
    let tempVar = alem;
    alem = balcha;
    balcha = tempVar;
    console.log(alem); // 20
    console.log(balcha); // 10
    ```

    - **Swapping variables values using destrcuturing**: Knowing how to destructure an array, it's easy to use it for swapping variables. This approach let us write cleaner and shorter code.

      ```
      let alem = 10,

      balcha = 20;

              console.log(alem); // 10

              console.log(balcha); // 20

              [alem, balcha] = [balcha, alem];

              console.log(alem); // 20

              console.log(balcha); // 10
      ```

- **Swapping values of an array with array destructuring**:
    - **Swapping values of an array with temporary variable**: Introduce a new variable and let it hold one of the two array values which are willing to swap. The array value which we let the temporary variable hold is reassigned by the second array value. Finally, (second variable) is given the value of temp which is a. **Example**:

```
let array = [0, "first", 2, "third", 4];

        console.log(array[1]); // prints first

        console.log(array[2]); // prints 2

        // temporary variable to swap array elements

        let tempVar = array[1];

        array[1] = array[2];

        array[2] = tempVar;

        console.log(array[1]); // prints 2

        console.log(array[2]); // prints first
```

        - **Swapping values of an array with array destructuring**:

            let array = [0, "first", 2, "third", 4];

            console.log(array[1]); // prints first

            console.log(array[2]); // prints 2

            // destructure to swap 2nd indexed element with 3rd

            [array[1], array[2]] = [array[2], array[1]];

            console.log(array[1]); // prints 2

            console.log(array[2]); // prints first

- **Object Destructuring**: The object destructuring is a useful JavaScript feature to extract properties from objects and bind them to variables. In short, you use object destructuring to extract some properties of an object
    - **Syntax of object destructuring**: const { identifier , identifier2} = expression;
        - **Identifier**: It is the name of the property to access
        - **Expression**: It represents the object. **Note**: After the destructuring, the variable identifier contains the property value.

- **Extracting of object properties before ES6**: Looking at the example below, such a way to access properties and assign them to variables requires boilerplate code. By writing var studentName = students.name, you have to mention the name binding 2 times, and the same for studentAge. That's where the object destructuring syntax is useful: you can read a property and assign its value to a variable without duplicating the property name.

```
const students = {
    name: "Haile",
    age: 22,
};
var studentName = students.name;
var studentAge = students.age;
console.log(studentName); // prints Haile
console.log(studentAge); // prints 22
```

- **Extracting of object properties using destructuring (after ES6)**: In the below example, the let { theName, theAge } = students is an object destructuring assignment. This statement defines the variables theName and theAge, then assigns to them the values of properties students.theName and students.theAge correspondingly. It visible that the object destructuring is handier than the literal object way because neither the property names nor the object variable is duplicated. Look at the example below:

```
let students = {
        theName: "Haile",
        theAge: 22,
};
let { theName, theAge } = students;
console.log(theName); // prints Haile
console.log(theAge); // prints 22
```

- **Note**: If the destructured object doesn't have the property specified in the destructuring assignment, then the variable is assigned with undefined. **Example**:

```
let students = {
theName: "Haile",
theAge: 22,
                };
                let { theName, theGrade } = students;
                console.log(theName); // prints Haile
                console.log(theGrade); // prints undefined
```

- **Using object destructuring to rename property identifiers**: Using the above example, let us assume you want to create the “theName” property name to studentName. To do so, look at the example below. **Example**:

```
let students = {
        theName: "Haile",
        theAge: 22,
        };
let { theName: studentName, theAge } = students;
console.log(studentName); // prints Haile
console.log(theAge); // prints 22
console.log(theName); // prints theName is not defined
```

- **Destructuring for nested objects**: Often objects can be nested in other objects. In other words, some properties of an object can contain objects.
    - **Syntax**: const { nestedObjectProp: { identifier } } = expression;

```
let students = {
        theName: "Haile",
        theAge: 22,
        theGenderAndHeight: {
                theGender: "Male",
                theHeight: 133,
        },
};
let {theName,theAge,theGenderAndHeight: { theGender,
theHeight }} =  students;
console.log(theName); // prints Haile
console.log(theAge); // prints 22
console.log(theGender); // prints male
```

- **Using the “rest” syntax in object destructuring**: The rest syntax is useful to collect the remaining properties after the destructuring. For instance, in the below example, you can see that we put “theRemainingProperties” variable during destructuring to hold the remaining properties (theAge, theGender and theMood)
    - **Syntax**: const { identifier, ...restOfProperties } = expression;
    - Example:

```
let students = {
        theName: "Haile",
        theAge: 22,
        theGender: "Male",
        theMood: "Happy",
};
let { theName, ...theRemainingProperties } = students;
console.log(theName); // prints Haile
//the below prints an object {theAge: 22, theGender: 'Male', theMood: 'Happy'}
console.log(theRemainingProperties);
console.log(theRemainingProperties.theMood); // prints happy
```

- **Destructing an object inside a function parameter**: This is basically to allow functions to accept an object as their parameter so that the object's property values can be extracted and used in the body of the function
  - **Before ES6**:

    ```
    function myFunc(myparam) {
        var name = myparam.name;
        var age = myparam.age;
        return name + " " + age;
    }
    console.log(myFunc({ name: "John", age: 25 }));// prints John    25
    ```

  - **After ES6 (using object destructuring)**: The above function can be rewritten using object destructuring like this:

    ```
        function myFunc({ name, age }) {
            return name + " " + age;
        }
        console.log(myFunc({ name: "John", age: 25 }));// prints John
        25
    ```

- **To read more on destructuring, visit the below link**:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

10.6 Class based components: converting functional components into class components

- **React class component**: After ES6, to create a class component in React you only need to define the component as a plain JavaScript class that we have covered above.
- **React class component before and after ES6**: When React was initially released in 2013 and until ES6 was released in 2015. JavaScript did not have classes. Because JavaScript didn't have a built-in class system then, React developers used React’s 'create-react-class' module to create class components. With ES6, React now allows you to implement component classes that use ES6 JavaScript classes. The result is the same, both ways can create a component class. But the syntax is different.
  - **Syntax of React before ES6**: We do not need to use this syntax now

```
var createReactClass = require('create-react-class');
        var Greeting = createReactClass({
                render: function() {
                  return <h1>Hello</h1>;
                }
        });
```

- **Syntax of React after ES6**: Creating a class component is pretty simple if you follow the following steps:
    - Import the React and Component from React
    - Define a class that extends Component and has a render function.
    - Export your component to make it available for other compoeents
    - This is the syntax we now use to create class components.

```
import React, { Component } from 'react';
                    class YourComponent extends Component {
                              render() {
                                        return <h1>Hello</h1>;
                              }
                    }
                    export default YourComponent;
```

    - **Note**: The above syntax is the one that we will use to create class components

- **Explaining each term/keyword in class component**
    - **import React, { Component } from 'react'**:
        - **Import React**: If you have created a new react app using creat-ereact-app recently, you will notice that your React app works fine without even importing React from “react”. We needed to import the “react” library because it is used to convert our JSX into vanilla JavaScript using the library’s createElement() method for the browser. However, after the release of React 17, our JSX is converted to vanilla JavaScript automatically without using React library’s createElement() method.

        - **Component**: React has different built-in methods (called React lifecycle methods). These built-in methods are found under a React class called Component. You will need to import the Component class if you want to use this class in your component
    - **class YourComponent extends Component"**: If you want your new component to inherit properties of the Component class, in addition to importing this class, you have to make sure that your component to extend the Component class.
    - **render()**: The render () method in class-based components returns your JSX (then converted to HTML) that will be rendered later in our index.html
    - **export default YourComponent**: This is basically making your new component available for other components to use it
- **Class vs functional components in React**: In addition to their syntax, there are major differences between class-based and functional components. Those differences will be clearer once we get to learn about props and states in later classes. For now, just focus on learning how to convert your functional components to class-based components
- **Steps to converting functional components into class-based components**:
    - **Step 1**: Replace the function key word with class
    - **Step 2**: Inherit properties of Component class by extending
    - **Step 3**: define the render method in your subclass then in your render() method, return the JSX you want to return
    - **Step 4**: If you are passing props, make sure to use the key word "this" to specifically identify your current class

10.7 React props

- **Passing data between React components**: Data sometimes needs to be able to move from children to parent, from parent to children, or between sibling components. Let us assume you want to change the data in a component wihtout actually hard coding the change in that component. Example: You have a component called GreetPeople that renders the message Hello followed by name (name that changes from one name to another). One way to have the greet name to change is to create different components that render the Hello message with different names. Another way to have the name change dynamically is using props
- **Props**: Props is just a shorter way of saying properties. They are properties that hold information about a component. Props are variables that get passed to the component from parent component (like a parameter is passed to a function). We use props in React to pass data from one component to another (from a parent component to a child component(s))
  - Props are objects of React and contain the number of key-value pairs.
  - Props are objects with immutable properties, meaning, we cannot change their value throughout the component. This is because changing value of props won't make sense, it will be like changing the argument of an adder function to fixed parameter.
  - Props can be available in both classes based and function-based components without any issue.
- **Using props to pass data from parent to child components**: This is the easiest direction in React to transfer data. If you have access to data in your parent component that you need your child component to have access to, you can pass data from parent as a prop to the child by initiating the prop within the parent. Let us start by defining what props are and how to use props in our components
  - **Note**: React's data flow between components is uni-directional (from parent to child only).
  - **Note**: You can use props in both functional and class components. You can even use functional component for your parent component and use a class component for your child component while using props.

- **Why use props**:
  - We use props in React to pass data from one/ parent component to a child
  - Props allow us reuse components dynamically. It is just like creating a function that takes an argument. Different components will have their own specific properties. Meaning the data will not be static as the component's data can be modified to fit what we want.
- **Steps to use props**: To understand the steps below, please look at the class demo on props (HeaderLinks.js, Fourth.js components on the Apple website)
  - **Step 1**: Create two components; a parent and a child component
  - **Step 2**: Import the child component in the parent component and return it.
  - **Step 3**: Import and return the parent component in your app.js
  - **Step 4**: Initiate/declare the props in your parent component
  - **Step 5**: Send props into a parent component,
    - **By returning your props as HTML attribute**s:
      - **Example**: return <ChildNavigation LinkURL="/Mac" />; or
    - **By returning your props as string**:

      let urlName = "/Mac ";

      return < ChildNavigation LinkURL ={urlName} />; or
    - **By returning your props as objects**:

      let urlName= { theURL: "/Mac" };

      return < ChildNavigation LinkURL={urlName. theURL} />;
    - **Why use {} in JSX**: Curly braces { } is special syntax in JSX. It is used in JSX (which is basically HTML) to instruct React to process the expression inside the braces as JavaScript during compilation. That JavaScript expression can be a variable, function, an object, or any code.

- **Step 6**: Receive/access props in your Child component via the “this” instance.
    - **Example for function components:** In a functiosnal component, the props are received in the function signature as arguments like below:
        - props. LinkURL
    - **Example for class components:** Unlike functional components, props in class component are not received in the render methods signature. However, they are received in React’s class component via the “this” instance of the class
        - this.props. LinkURL
    - **Meaning of {this.props} in class components**: Props are properties that are passed to class components by default as they are made available by React library. In our component, we need to bind the word “this” with the inherited properties, props in our case, so that props are available to the component.

- **Using props in class components**: See the example below

// Child component here

```
import { Component } from "react";

        class FufiChild extends Component {

                render() {

                        return <h1>Hello {this.props.Name}</h1>;

                }

        }

export default FufiChild;
```

```
// Parent component here
    import React, { Component } from "react";
    import FufiChild from "./FufiChild";
     class FufiParent extends Component {
        render() {
            return <FufiChild Name="My name is Sen" />;
        }
    }
    export default FufiParent;
```

- **Using props in function components**:

```
import React from "react";
function FufiChild(props) {
    return (
        <h1>Hello {props.Name}</h1>
    );
}
export default FufiChild;
```

- **Multiple props**: You can use as many props as you like. Let us see how you can use multiple props in 2 examples below
    - **Example 1**:

//Child component

```
import React from "react";
function FufiChild(props) {
        return (
                <h1>
                        This is {props.Name}: {props.Age} and {
                props.Gender}
                </h1>
        );
}
export default FufiChild;
```

// Parent component

```
import React, { Component } from "react";
import FufiChild from "./FufiChild";
class FufiParent extends Component {
        render() {
                return <FufiChild Name="Abebe" Age="he is 55"
                Gender="male" />;
        }
}
export default FufiParent;
```

- **Example 2 (spread attributes)**: Using spread attributes to make passing props easier. For this example, use the above child component.

  // Parent component

```
import React, { Component } from "react";
            import FufiChild from "./FufiChild";
             class FufiParent extends Component {
                        render() {
                                    let allProps = {
                                                Name: "Abebe",
                                                            Age: "he is 44",
                                                            Gender: "male",
                                    };
                                    return <FufiChild {...allProps} />;
                        }
            }
            export default FufiParent;
```

- **Destructuring props**
  - **Destructuring (revision)**: In our previous discussion about destructuring we said that the destructuring concept was introduced in ES6 to allow us to extract multiple pieces of data from an array or object and assign them to their own variables. Destructuring, does not change the array or the object, it makes a copy of the desired object or array element by assigning them in its own new variables so that we can use this new variable anytime.
  - **Destructuring props in functional components**: We achieve destructuring of our props in functional components as we pass in the props argument in the function.
    - Notice below that destructuring our props gives us access to both the “Name” and the “Age” properties, which we need for different children components.

      ```
      //Parent component

      import React, { Component } from "react";

      import ChildDestructuringProps from "./ChildDestructuringProps";

       class ParentDestructuringProps extends Component {

              render() {

                      return (

                              <div >

                                      <ChildDestructuringProps    Name="Abebe"
                                      Age="is 55" />

                              </div>

                      );

              }

      }
      ```

```
//Child component

import React from "react";

function ChildDestructuringProps(props) {

        const { Name, Age } = props;

        return (

                <div>

                        {Name} {Age}

                </div>

        );

}

 export default ChildDestructuringProps;
```

- **Destructuring props in class components**: When we access the props using the "this" keyword, we have to use this/ this.props throughout the program, but by the use of destructuring, we can discard this/ this.props by assigning them in new variables. This is very difficult to monitor props in complex applications.
    - **So, how do we destructure props in class components?** We destructure props in the render() function under the child component by assigning all of our props on a new variable. Please note that unlike functional components, in class components we do not acheive destructuring when the props are passed in in the function's argument. We achieve the destructuring of props in class components wherever the variables are called.
    - Notice below that destructuring our props gives us access to both the "Name" and the "Age" properties, which we need for different children components. **Example**:

//Parent component

```
import React, { Component } from "react";

import ChildDestructuringProps from "./ChildDestructuringProps";

class ParentDestructuringProps extends Component {

        render() {

                return (

                <div >

                        <ChildDestructuringProps Name="Abebe"
                        Age="is 55" />

                </div>

        );

}

}

export default ParentDestructuringProps;
```

// Child component

```
import React, { Component } from "react";

class ChildDestructuringProps extends Component {

    render() {

        const { Name, Age } = this.props;

        return (

            <h1>{Name} {Age}</h1>

        );

    }

}


export default ChildDestructuringProps;
```