React Preparation

Contents

Introduction

Before we kick things off fully with React, we have to get a few things in order to help us on our journey. These are all just JavaScript features, but they are used frequently in React code. You can use them in your vanilla JS projects now too.

Destructuring

This is a new way of getting information from larger structures. It's a nice new syntax to help make your code easier to read. It will shake you up a bit at first, but you will get used to it quickly, and we have some challenges for you to get through to make it stick.

Let's look at it now:

```
console.log('destructuring setting off')

clothes = ["red t-shirt", "blue dress", "green shoes"]

// This is one way to get the information we need out of the array
let redTShirt = clothes[0]
console.log(redTShirt)

let [ redTShirtDestruct, blueDress, greenShoes ] = clothes
// let redTShirtDestruct = clothes[0]
// let blueDress = clothes[1]
// ...
console.log(redTShirtDestruct)
console.log(blueDress)
console.log(greenShoes)

const nums = [4,7,2,9,1,5]
const [ first, second, third ] = nums
console.log(first, second, third)

const [ , , , fourth, , sixth ] = nums
console.log(fourth, sixth)

const [ head, ...otherNums ] = nums
console.log(head, otherNums)
```

```
const rugbyPlayer = {
    name: "Justin Olam",
    weight: 105,
    team: "Storm"
}

const { team } = rugbyPlayer

console.log(team)

const { weight, name } = rugbyPlayer
console.log(weight, name)

const { name: playerName } = rugbyPlayer
console.log(playerName)

const mcenroe = {
    name: 'John McEnroe',
    totalMajors: 7,
    slamBreakdown: {
        ausOpen: [],
        rolandGarros: [],
        wimbledon: [1981, 1983, 1984],
        usOpen: [1979, 1980, 1981, 1984]
    }
}

const { slamBreakdown: { wimbledon: [ , wimYear ] } } = mcenroe
console.log(wimYear)
const finalWim = mcenroe.slamBreakdown.wimbledon[2]
console.log(finalWim)
```

Destructuring Challenges

```
// Destructuring Challenges

// A
veg = ['cauliflower', 'pumpkin', 'carrot', 'leek', 'onion']
// 1. Use destructuring to each element out of this array, and store in
separate variable one at a time (that is, practice just getting one
element out each line).
// 2. Retrieve just the first two elements.
// 3. Retrieve the second and third elements.
// Use the plat operator for these:
// 4. Retrieve the first, and the rest (the rest being only from that
point on in the array).
// 5. Retrieve the first, third, and the rest.
// 6. Retrieve the second, fourth, and the rest.

// B
const recordPlayer = {
```

```
    brand: "Sanyo",
    model: "FFT9000",
    speeds: ["33", "45"],
    options: ["extra speakers", "power cord", "stand"],
    blurb: "State of the art record playing experience"
}
// 1. Use destructuring to get out the brand.
// 2. Retrieve the model.
// 3. Retrieve the blurb, and use string interpolation to make it part of
a larger sentence.
// 4. Get out each of the above but using an alias.
// 5. Dig further to get out "power cord".
// 6. Dig further to get out "33".
// 7. Retrieve "extra speakers", and the rest.
// 8. Retrieve "power cord", and the rest.

// C
fruitAndVeg = [
    ['avocado', 'apple'],
    ['grape', 'guava'],
    ['pear', 'pumpkin', 'peach']
]
// 1. Retrieve "pear" and the rest in that array
// 2. Retrieve the whole second array.
// 3. Retrieve just apple.

// D
federer = {
    name: 'Roger Federer',
    totalMajors: 20,
    slamBreakdown: {
        ausOpen: [2004, 2006, 2007, 2010, 2017, 2018],
        rolandGarros: [2009],
        wimbledon: [2003, 2004, 2005, 2006, 2007, 2009, 2012, 2017],
        usOpen: [2004, 2005, 2006, 2007, 2008]
    }
}
// 1. Retrieve the name attribute.
// 2. Retrieve the total majors.
// 3. Retrieve the whole US Open array.
// 4. Pull out the fourth element of the Australian Open array.
// 5. Store in constants the first three items in the Wimbledon array, and
also the rest of the array in the one line.
```

Classes

Classes in JS provide a way to do more traditional style object-oriented programming in JS. Everything in JS is an object at the deepest level. JS uses a prototype system under the hood, although what that means is not within the scope of this document. The JS class system is a way to introduce the class syntax that users of more traditionally OO languages are familiar with.

Object oriented programming as a paradigm at its most basic is a way of constructing code where the basic elements are objects that interact with other objects. Objects are little batches of attributes (things that define the object), and methods (things that the object can do).

The central idea is in terms of the code is that classes are blueprints for objects, and objects are instances or examples of the class. You can think of an analogy, such as the class being the design of a car, and the constructor function acting like an iteration of a factory producing a car. The car itself is the object, the instance - the blueprint made real. The class acts like the generalised idea, and the cars themselves are the reality. When you produce the instances, you are usually setting some of the attributes to be specific to the instance, although there will also be many aspects (such as the methods) that will tend to be common to many objects from the same class.

Let's have a look at some examples:

```
console.log("Having a look at classes.")

// This is the way we have been making objects in JS.
player1 = {
    name: "Justin Olam",
    weight: 105,
    teams: ["Melbourne Storm", "PNG National Team"]
}

// But you can see if we have several to make, then we have a lot of
boilerplate to write to instantiate an object.
player2 = {
    name: "Billy Slater",
    weight: 88,
    teams: ["Melbourne Storm", "Australian National Team"]
}

// Now we can create a RugbyPlayer class that will set out the basic
structure of a rugby player for us. We can set default values in here too,
but also lay out all the values that we will need to 'customise'.
class RugbyPlayer {
    // The constructor function runs when you make a new object. We can
set the attributes that are customisable via arguments to this constructor
function. We can also set the default values at this time too (pay, for
example here).
    constructor(name, weight, teams, age, currentTeam) {
        this.name = name
        this.weight = weight
        this.age = age
        this.teams = teams
        this.currentTeam = currentTeam
        this.pay = 100000
    }

    // This is an instance method, meaning that it is only available to an
instance, or object.
    movePlayerTeam(team) { // instance method
```

```
        this.currentTeam = team
        this.teams.push(team)
    }

    playerBirthday() { // instance method
        this.age += 1
    }
}

// Here we are creating two instances, and setting their values. Remember,
it is the constructor function that runs when we instantiate an object.
const justinOlam = new RugbyPlayer("Justin Olam", 105, ["Storm", "PNG
National"], 30, "Storm")
const billySlater = new RugbyPlayer("Billy Slater", 88, ["Storm",
"Australia"], 35, "Storm")

// We now have two RugbyPlayer objects to work with, and can access their
attributes in the usual way.
console.log(justinOlam.weight)
console.log(justinOlam.age)
console.log(billySlater.age)

// This is an example of calling an instance method (again, a method only
available on an instance, or object).
billySlater.playerBirthday()

// Here we are just checking the result of using the above method. Part of
what we are doing here is showing that running that method only has an
effect on the particular instance on which it is called. We can see that
only our "Billy Slater" object has had its age incremented.
console.log(justinOlam.age)
console.log(billySlater.age)

// Here we are runnning another example to play with and see how this all
works.
class Car {
    constructor(colour) {
        this.colour = colour
        this.running = false
    }

    carSmash = () => {
        console.log(this.colour)
        console.log("Car smash works?")
    }

    carStart() {
        if (this.running) {
            console.log(`The ${this.colour} is already running!`)
            return
        }
        this.running = true
        console.log(`The ${this.colour} car is on the move!`)
    }
```

```
    carStop() {
        if (!this.running) {
            console.log(`The ${this.colour} is already stopped!`)
            return
        }
        this.running = false
        console.log(`The ${this.colour} car has stopped.`)
    }
}

// We create three instances here.
car1 = new Car("yellow")
car2 = new Car("green")
car3 = new Car("red")

// Running an instance method on our car3 instance.
car3.carSmash()

// Here we are expanding our little world to include a CarPark object.
This is an example of how you might build up your world using objects.
class CarPark {
    constructor(name) {
        this.name = name
        this.cars = []
    }
    // This instance will 'put a car into a CarPark instance' by pushing a
Car instance into the CarPark instance, which mimics the concept of
putting a car into a carpark.
    addCar(carInstance) {
        this.cars.push(carInstance)
    }
}

// Creating a CarPark instance.
const kingParking = new CarPark("King Parking")

// Calling some instance methods.
car1.carStart()
car1.carStart()

console.log(`${car2.colour} car status is ${car2.running}`)

car2.carStart()
car3.carStop()

// Adding cars to the CarPark instance
kingParking.addCar(car1)
kingParking.addCar(car3)
kingParking.addCar(car2)

console.log(kingParking.cars)
console.log(kingParking.cars[2].colour)
```

Fat Arrow Functions

Fat arrow functions provide a new syntax for defining functions as part of ES6. There are many situations where they are interchangable with the functions that we have been using. However there are some situations where they are not - although we will get to that difference down the track (it relates to how each type of function handles 'this'). For now we are going to just get used to the change in syntax, and will warn you where there are instances where one is preferred over the other.

```
// This is the traditional way to define this function.
const area = function(length, width) {
    return length * width
}

// This is the same function, but using the fat-arrow syntax.
const area2 = (length, width) => {
    return length * width
}

// And again, and this time we are seeing it on one line, with implicit
return (meaning simply that you can leave the keyword 'return' out).
const area3 = (length, width) => { length * width }

const arr = [1,2,3]

// Here is an example of a .forEach loop using this syntax - all on one
line.
arr.forEach((item, i) => { console.log(item) })

// .map, new syntax, but multi-line.
arr.map(item => {
    return item * 2
})
```

Inheritance and further class examples

Inheritance is the process of using one class to form another class. This is usually a narrowing of the definition, as a more general class becomes increasingly specific. For example, you may define an `Animal` class, and then use that class to form a `Dog` class (in JS to `extend`) from that more general class.

In this document we won't be going into too much theory here, but this example can show some more features of the JS class system:

```
console.log("banking app")

// A bank account class
class BankAccount {
    // A constructor that sets only one attribute, the other two are
provided.
```

```
    constructor(name) {
        this.name = name
        this.balance = 0
        this.accountNumber = Math.round(Math.random() * 10000000)
    }

    // The static keyword defines what is in some other languages a
'class' variable or method - that is, a method that is only available on
the class, and not to the instance of a class. In this case we use it to
total the bank's holdings, which will be the total of all the accounts.
This type of knowledge is not the purview of any one account holder, but
the bank overall. There would be several ways to achieve the same thing,
and this is justa a demonstration of a way to use a class variable.
    static totalBankHoldings = 0

    // This method is a class method, and uses a class variable within.
    static addToBankHoldings(money) {
        BankAccount.totalBankHoldings += money
    }

    static takeFromBankHoldings(money) {
        BankAccount.totalBankHoldings -= money
    }

    // An instance method, along with a call to a class method.
    deposit(money) {
        this.balance += money
        console.log(`$${money} deposited into account number
${this.accountNumber}`)
        BankAccount.addToBankHoldings(money)
    }

    withdraw(money) {
        if (this.balance > money) {
            this.balance -= money
            console.log(`$${money} withdrawn into account number
${this.accountNumber}`)
            BankAccount.takeFromBankHoldings(money)
        }
        else {
            console.log("Insufficient funds.")
        }
    }
}

// Here we have an example of inheritance. JS uses the `extends` keyword.
class SpecialBankAccount extends BankAccount {
    constructor(type, name) {
        // `super` is the call to run the constructor of the parent.
        super(name) // runs the BankAccount constructor
        this.type = type
    }

    // The SpecialBankAccount instances will have access to the instance
```

```
methods of the BankAccount class, and also this new method `takeFees`.
    takeFees(money) {
        console.log(`The ${this.type} just took $${money} in fees`)
        this.balance += money
        BankAccount.totalBankHoldings -= money
    }
}

// Here we make use of these classes and look at some outcomes.
manager1 = new SpecialBankAccount("manager", "Greedy Garry")
console.log(manager1.name)
console.log(manager1.type)
console.log(manager1.balance)
manager1.takeFees(50)
console.log(manager1.balance)

console.log(BankAccount.totalBankHoldings)

account1 = new BankAccount("Matt")
console.log(account1.name)
console.log(account1.accountNumber)

account2 = new BankAccount("Gerda")
console.log(account2.balance)

account2.deposit(300)
console.log(account2.balance)

account1.withdraw(500)
account1.deposit(500)

console.log(BankAccount.totalBankHoldings)
```