FUNCTIONAL PROGRAMMING IN REACT

1. What is Functional Programming?

The functional programming is a concept of creating pure functions for software logic. It avoids concepts of mutable data and a shared state as used in Object-oriented programming.

Functional programming is a programming paradigm designed to handle pure mathematical functions. This paradigm is totally focused on writing more compounded and pure functions.

The functional programming is based more on expressions and declarations rather the statements. The functional programming depends only on the arguments passed to the function.

2. Why Functional Programming?

If we were to write an application that followed all of the functional programming principles our code would be:

2.1 Concise

- A Concise code stands for a code that is precisely written and easy to understand. The code should be short and neat.

2.2 Maintainable

- Maintainable code simply means “code that is easy to modify or extend”. At the heart of maintainability is carefully constructed code that is easy to read; code that is easy to dissect in order to locate the particular component relating to a given change request; code that is then easy to modify without the risk of starting a chain reaction of breakages in dependent modules.

2.3 Easier to debug

- Code that’s easy to debug is code that checks to see if things are correct before doing what was asked of it, code that makes it easy to go back to a known good state and trying again, and code that has layers of defense to force errors to surface as early as possible.

2.4 Testable

- Testable code is a code that can be verified programmatically and in a very granular way. When a test fails, we want it to tell us exactly why so that we know what went wrong and can correct the problem immediately.

2.5 Readable

- Readable code is simply code that clearly communicates its intent to the reader. Most likely, the code we write will be read by other developers, who will either want to understand or modify the way our code works.

2.6 Reusable

- In programming, reusable code is the use of similar code in multiple functions. Code reusability defines the methodology you can use to use similar code, without having to re-write it everywhere.

3. Functional Programming concepts

Functional programming has a few important concepts that we need to know and understand. By implementing these concepts in your applications, you will end up with more functional code.

This will make a huge difference in your application, making it more readable, usable, manageable, easy to test and bug-free.

3.1 Functional Programming says you should avoid the following

3.1.1 Avoid mutations

- Avoid changing state (aka avoid mutations, aka immutability), suggests us to make copies of the state and editing the copy, rather than editing the original state.
- As an example, if you had an array of team members and wanted to add someone new, instead of editing the current array you should copy it and edit that.
- This may also be written as 'you should transform your state'.

3.1.2 Avoid side effects

- Avoid functions that change the 'outside world' (aka avoid side effects), is similar to the above in that your functions should only copy and edit the input, rather than editing the original input.
- Sometimes side effects are required, for example, logging to console, writing to the screen, triggering an external process, writing to a file, etc., but where ever possible you should not be 'editing' the outside world, you should be 'adding' to it.
- Anytime you do need side effects, you should separate and isolate the actions from the rest of your application as much as possible.

3.1.3 Avoid sharing state

- The state in your application should never be 'shared' (aka avoid sharing state).

- For state to not be ‘shared’, it means that every time you need to ‘change it’ you should duplicate it and edit the duplicate, thus state is never ‘shared’ as such.

3.2 Functional Programming says you should do the following

3.2.1 Write pure functions

- Writing functions that are predictable, only do one thing and do not change the ‘environment’ around it (aka write pure functions).
- They have no ‘side effects’ and given the same input, they always return the same output.

3.2.2 Function composition

- Combine smaller functions into bigger functions that build a full application (aka be thoughtful about your function composition).
- This helps us achieve those desired application characteristics mentioned as earlier.

3.2.3 Write declarative code

- You should write code that displays ‘what’ should happen rather than ‘how’ it should happen (aka write declarative code).
- An example of this would be choosing to use the map function, instead of a for loop, because the map function is a more concise version of a loop.

4. Functional Programming in React

4.1 Write pure functions

Pure functions return the same value given the same input and do not mutate our data. When writing functions, you should try to follow these rules to ensure they are pure

1. The function should take at least one argument (the original state)
2. The function should return a value or another function (the new state).
3. The function should not change or mutate any of its arguments (it should copy them and edit that using the spread operator).

This helps ensure our applications state is immutable, and allows for helpful features such as easier debugging and more specifically features such as undo / redo, time travel via the redux devTool chrome extension.

In React, the UI is expressed with pure functions as you can see in the following code snippet.

```
const Header = props => <h1>{props.title}</h1>
```

It does not cause side effects and is up to another part of the application to use that element to change the DOM (which will also not cause harmful side effects).

4.2 Spread operator (...)

The spread operator is an essential tool in writing pure functions and helps us ensure our application is immutable. See the below pure function. As you can see it's copying the original array into a new one.

```
let colorList = [
    {color: 'Red'},
    {color: 'Green'},
    {color: 'Blue'}
]

// The wrong way - colorList is mutated because we have pushed
// something into the existing array. It's also not declarative.

var addColor = function(color, colorList) {
    colorList.push({color : color })
    return colorList;
}

// The right way - colorList is immutable // and is declarative code.

const addColor = (color, colorList) => [...colorList, {color}];
```

Let's look at another example, whereby we need to pull the last element from an array. Notice we are using ES6 destructuring to create our variables.

```
const numbersArray = [1,2,3,4,5,6]

const [lastNumberInArray] = [...numbersArray].reverse()
// 6

// We have created a new numbers array using the spread operator.
// We then reversed it so we can pull out what was the last number in the array.
// It would be the same as writing the below less declarative way.

const lastNumberInArray = [...numbersArray].reverse()[0]
```

Thus, the spread operator is crucial in helping us not mutate our state

4.3 Write declarative code

Writing code declaratively essentially means writing the least amount of code you can. The simplest way of understanding this is to take a look at the below example where we use the native JavaScript map function to achieve our goal in one line rather than three.

```
// imperative
const makes = [];
for (let i = 0; i < cars.length; i += 1) {
  makes.push(cars[i].make);
}

// declarative
const makes = cars.map(car => car.make);
```

An example of the declarative nature of React is it's render method.

The below code renders a welcome message into the browser. It is a clean, simple way of writing something that would be very convoluted in without the help of the render function.

```
const { render } = ReactDom

const Welcome = () => (
  <div id="welcome">
    <h2>Hello!</h2>
  </div>
)

render(
  <Welcome />,
  document.getElementById('target')
)
```

4.4 Higher Order Functions

These are functions that are defined by their behavior. Higher order functions either have another function passed in as an argument, or, return another function.

Higher order functions include a number of useful tools to help us write functional software. Those tools include:

4.4.1 Map

Map applies a function to each element in an array and returns the array of updated values. The below example of the map function takes a list of colours, edits an existing colour, and returns a new list.

```
let colorList = [
    {color: 'Red'},
```

```
    {color: 'Green'},
    {color: 'Blue'}
]

const editColor = (oldColor, newColor, colorList) =>
colorList.map(item => (item.color === oldColor) ? ({...item, color: newColor}) : item)

const newColorList = editColor('Blue', 'Dark Blue', colorList);

console.log(newColorList);

// [ {color: 'Red'}, {color: 'Green'}, {color: 'Dark Blue'} ]
```

We can use the map function to transform an object into an array. The example below shows how we can transform an object of book titles and their author into a more useful array.

```
const booksObject = {
    "Clean Architecture": "Robert C Martin",
    "JavaScript Patterns": "Stoyan Stefanov"
}

const booksArray = Object.keys(booksObject).map(key => ({bookTitle: key, author:booksObj
ect[key]}));

console.dir(booksArray);

// [
//    {bookTitle: "Clean Architecture", author: "Robert C Martin"},
//    {bookTitle: "JavaScript Patterns", author: "Stoyan Stefanov"}
// ]
```

4.4.2 Filter

The below example of the filter function takes a list of members, creates a new list and removes the desired member so we have an up to date members list.

If the function you pass in returns true, the current item will be added to the returned array and thus you have filtered your array. Also, note the reject function, which works inversely to filter.

```
const userList = [
    {name: 'Bob', member: true},
    {name: 'Fred', member: true},
    {name: 'Keith', member: false}
]

const isMember = user => user.member === true
```

```
const members = userList.filter(isMember);

console.log(members);

// [{name: 'Bob', member: true},{name: 'Fred', member: true}]

// Notice how we have separated out isMember to its own function. This is declarative co
de and
// means we can reuse the function in the following way.
// Also, reject is just the opposite of filter.

const nonMembers = userList.reject(isMember)

console.log(nonMembers)

// [{name: 'Keith', member: false}]
```

4.4.3 Reduce

The third method is the reduce function. This is the 'multitool' and provides a more general function for when map and filter aren't appropriate.

The important thing to notice about reduce is that is requires a few more parameters than the others. The first parameter is the callback function (which also takes parameters) and the second parameter is the starting point of your iteration

The below function creates a new array of users that are older than 18.

```
const users = [
    { name: 'Keith', age: 18 },
    { name: 'Bob', age: 21 },
    { name: 'Fred', age: 17 },
    { name: 'George', age: 28 },
  ];

  const usersOlderThan21 = users.reduce((result, item)=>{
      item.age >= 18 ? result[item.name] = item.age : null
      return result
  }, {})

  // {Keith: 18, Bob: 21, George: 28}
```

4.4.4 Recursive functions

A recursive function is a function that calls itself, until it doesn't! It's as simple as that. You may choose a for loop when you only had one or two levels of recursion. The issue is when you have lots of levels of recursion, the for loop suddenly begins to become very unwieldy.

The benefit of a recursive function, is that you can simply make a function call itself again and again till your rule is met. A recursive function can do what a for loop can, but in a much concise way.

```
// For loop

for (i = 0; i < 11; i++) {
    console.log(i);
}
// 0, 1, 2, 3 ...

// Recursive function

let countToTen = (num) => {
    if (num === 11) return
    console.log(num)
    countToTen(num + 1)
}

countToTen(0)
// 0, 1, 2, 3 ...
```

4.4.5 Currying functions

Currying is a function that holds onto a function which you can reuse at a later point in time. This allows us to break our functions down into their smallest possible responsibility which helps with reusability.

```
const add = (a, b) => a + b

    const a = add(0,1) // 1
    const b = add(10, 1) // 11
    const c = add(20, 1) // 21

    // We can see we are adding one alot, so much
    //we should abstract this further and make a curried function.

    const curriedAdd = (a) => (b) => a + b

    const add1 = curriedAdd(1);

    const d = add1(0) // 1
    const e = add1(10) // 11
    const f = add1(20) // 21

    // maybe we also want to have an add2 function?
```

```
    const add2 = curriedAdd(2);

    const g = add2(0) // 2
    const h = add2(10) // 12
    const i = add2(20) // 22

    // as you can see we have a reuseable add function
    // that we can apply as and where we need it.
```

We can create a curried version of map, which allows us to create functions that can be run on an array, for example a doubleAll function.

```
   // we can create a curried version of map which takes a function
    // and maps across over it and returns a new function which
    // will run our original function multiple times.

    const arr = [1, 2, 3, 4];

    const curriedMap = fn => mappable => mappable.map(fn);
    const double = n => n * 2;

    const doubleAll = curriedMap(double);

    doubleAll(arr)

    // [2,4,6,8]
```

4.5 Chaining functions

It's worth remembering that functions can be chained together as well. This is another way that helps you to combine your smaller functions into larger ones.

```
const numbers = [1,2,4,5,7,8,9,10];

let isEven = (num) => num % 2 == 0
let double = (num) => num * 2

let doubleAllEvenNumbers = numbers
    .filter(isEven)
    .map(double)

console.log(doubleAllEvenNumbers)
//[ 4, 8, 16, 20 ]
```

4.6 Compose

Similar in the way we can combine smaller functions by chaining them together, we can merge them through a function that is commonly named compose().

Compose is a non-native function to JavaScript and you can create it yourself as you can see from the below example. This helps with readability and maintenance.

```
// create our compose funciton

const compose = (...fns) => {
    (arg) => {
      fns.reduce(composed, f) => f(composed), arg)
    }
  }

  // create our single responsibility functions
  var sayLoudly = string => {
      return string.toUpperCase();
  }

  var exclaim = string => {
      return string + '!!';
  }

  // compose our single responsibility functions into a single one

  var shout = compose(sayLoudly, exclaim);

  exclaim('crumbs');

  // crumbs!!

  shout('crumbs')

  // CRUMBS!!
```

4.7 Promises

JavaScript can only do one thing at a time as it is a single threaded programming language. If we needed to load some blog posts from an API we ideally wouldn't want our whole page to have to wait for this information before loading.

In the past, we used callback functions to handle but very quickly it landed us in 'callback hell', which was where you would have to nest numerous callbacks which ended up in very bloated code.

In recent years, ES6 has introduced Promises to deal with asynchronous behavior. These are going to be integral to most software applications and so are required knowledge for the modern JavaScript engineer.

```
const getBlogPosts = (endpoint) => new Promise((resolves, rejects) => {
    const api = `https://jsonplaceholder.typicode.com/${endpoint}`
    const request = new XMLHttpRequest()
    request.open('GET', api)
    request.onload = () =>
        (request.status === 200) ?
        resolves(JSON.parse(request.response)) :
        reject(Error(request.statusText))
    request.onerror = err => rejects(err)
    request.send()
  })

  const processBlogPosts = (postsJson) => console.log(postsJson.title, postsJson.body)

  getBlogPosts('posts/1').then(
    posts => processBlogPosts(posts),
    error => console.log(new Error('Cannot get posts'))
  )
```

We can make the Promise function even more declarative using the **async/await** functions. Look at the example below where we have created a function called get getBlogPosts which returns a promise.

We then create an **async** function which can then **await** for the promise to be returned. We can use try to handle a successful response and catch to handle a failed response.

```
const getBlogPosts = (endpoint) => {
    return new Promise((resolves, reject) => {
      const api = `https://jsonplaceholder.typicode.com/${endpoint}`
      const request = new XMLHttpRequest()
      request.open('GET', api)
      request.onload = () =>
          (request.status === 200) ?
          resolves(JSON.parse(request.response)) :
          reject(Error(request.statusText))
      request.onerror = err => rejects(err)
      request.send()
    })
  }

  const processBlogPosts = async (apiEndPoint) => {

    try {
      const blogPosts = await getBlogPosts(apiEndPoint);
      console.log('Success', blogPosts)
    }
```

```
  catch {
    console.error('Could not get blog posts')
   }

}

processBlogPosts('posts/1')

//Success
// {title: "Blog Post title", content: "The content of the blog post"}
```

5. Conclusion

Functional programming is a very useful style of writing code and has been used by React and Redux for good reason.

Remember that it's very easy to slip away from functional programming while writing JavaScript so you need to stay focused. The following few simple rules will help you stay on target.

1. Keep data immutable.
2. Keep functions pure (functions should take at least one argument and return data or a function).
3. Keep your code as concise as possible.
4. Use recursion over looping (will help solve complex issues in a neater way).

6. References

- https://dev.to/thomas_hoadley/functional-programming-basics-before-you-learn-react-part-1-5b9l/
- https://www.telerik.com/blogs/functional-programming-javascript
- https://www.wearetechtonic.com/functional-programming-in-react/