Functional Programming: 101

Introduction to functional programming with JavaScript

What is functional programming?

- Derived from λ-calculus (*lambda calculus*)
- In the late 1950s, John McCarthy took the concepts derived from λ-calculus and applied them to a new programming language called Lisp
- Lisp implemented the concept of higher-order functions and functions as first-class members or first-class citizens.

What is functional programming?

- JavaScript supports functional programming because JavaScript functions are first-class citizens.
- JavaScript functions can be
  - Assigned to variables,
  - Added to arrays & objects
  - Sent to & returned from other functions

Let's look at some examples

```
var log = function(message) {
  console.log(message)
};

log("In JavaScript functions are variables")
```

The same function using an arrow function

```
const log = message => console.log(message)
```

Functions can be added to objects like variables

```
const obj = {
    message: "They can be added to objects like variables",
    log(message) {
        console.log(message)
    }
}

obj.log(obj.message)
```

We can also add functions to arrays in JavaScript

```
const messages = [
  'They can be inserted into arrays',
  message => console.log(message),
  'like variables',
  message => console.log(message),
];


messages[1](messages[0]);
messages[3](messages[2]);
```

Functions can be sent to other functions as arguments

```
const insideFn = logger =>
  logger("They can be sent to other functions as arguments");


insideFn(message => console.log(message))
```

Functions can also be returned from other functions

```
var createScream = function(logger) {
    return function(message) {
        logger(message.toUpperCase() + "!!!")
    }
}

const scream = createScream(message => console.log(message))

scream('functions can be returned from other functions')
scream('createScream returns a function')
scream('scream invokes that returned function')
```

The same function using an arrow function

```
const createScream = logger => message =>
    logger(message.toUpperCase() + "!!!")
```

In conclusion

- Functions are data
- In JavaScript, functions can represent data in your application since they can be saved, retrieved, or flow through your applications just like variables

Imperative vs Declarative

Imperative vs Declarative

- Functional programming is a part of a larger programming paradigm: ***declarative programming***.
- Declarative programming is a style of programming where applications are structured in a way that prioritizes describing what should happen over defining how it should happen.

Imperative vs Declarative

Imperative

```
var string = "This is the midday show with Cheryl Waters";
var urlFriendly = "";

for (var i=0; i<string.length; i++) {
  if (string[i] === " ") {
    urlFriendly += "-";
  } else {
    urlFriendly += string[i];
  }
}

console.log(urlFriendly);
```

Declarative

```
const string = "This is the mid day show with Cheryl Waters"
const urlFriendly = string.replace(/ /g, "-")

console.log(urlFriendly)
```

Imperative vs Declarative

- In a declarative program, the syntax itself describes what should happen and the details of how things happen are abstracted away.
- Declarative programs are easy to reason about because the code itself describes what is happening.

Let's look at another example

Imperative

```
var target = document.getElementById('target');
var wrapper = document.createElement('div');
var headline = document.createElement('h1');

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

Declarative

```
const { render } = ReactDOM

const Welcome = () => (
    <div id="welcome">
        <h1>Hello World</h1>
    </div>
)

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

Functional Concepts

Functional Concepts

- The core concepts of functional programming
  - Immutability
  - Purity
  - Data transformation
  - Higher-order functions
  - Recursion and
  - Composition

Immutability

Immutability

- To mutate is to change, so to be immutable is to be unchangeable
- In a functional program, data is immutable, it never changes.
- Instead of changing the original data structures, we build changed copies of those data structures and use them instead.

Let's look at some examples

```
let color_lawn = {
  title: 'lawn',
  color: '#00FF00',
  rating: 0
};
```

```
function rateColor(color, rating) {
  color.rating = rating
  return color
}

console.log(rateColor(color_lawn, 5).rating)     // 5
console.log(color_lawn.rating)                   // 5
```

Rewrite the rateColor function

```
var rateColor = function(color, rating) {
   return Object.assign({}, color, {rating:rating})
}

console.log(rateColor(color_lawn, 5).rating)      // 5
console.log(color_lawn.rating)                    // 4
```

The same function using an arrow function

```
const rateColor = (color, rating) =>
    ({
        ...color,
        rating
    })
```

Let's consider an array of color names

```
let list = [
    { title: "Rad Red"},
    { title: "Lawn"},
    { title: "Party Pink"}
]
```

```
var addColor = function(title, colors) {
  colors.push({ title: title })
  return colors;
}

console.log(addColor("Glam Green", list).length)        // 4
console.log(list.length)                                // 4
```

Rewrite the rateColor function

```
const addColor = (title, array) => array.concat({title})

console.log(addColor("Glam Green", list).length)        // 4
console.log(list.length)                                // 3
```

Using the ES6 spread operator

```
const addColor = (title, list) => [...list, {title}]
```

Pure Functions

Pure Functions

- A pure function is a function that returns a value that is computed based on its arguments.
- Pure functions take at least one argument and always return a value or another function.
- They do not cause side effects, set global variables, or change anything about application state.
- They treat their arguments as immutable data.

What does an impure function look like?

```
var frederick = {
    name: "Frederick Douglass",
    canRead: false,
    canWrite: false
}
```

```
function selfEducate() {
    frederick.canRead = true
    frederick.canWrite = true
    return frederick
}

selfEducate()
console.log( frederick )
```

Let's rewrite the selfEducate function

```
var frederick = {
    name: "Frederick Douglass",
    canRead: false,
    canWrite: false
}
```

```
const selfEducate = person =>
    ({
        ...person,
        canRead: true,
        canWrite: true
    })

console.log( selfEducate(frederick) )
console.log( frederick )
```

Let's examine an impure function that mutates the DOM

```
function Header(text) {
  let h1 = document.createElement('h1');
  h1.innerText = text;
  document.body.appendChild(h1);
}

Header("Header() caused side effects");
```

Let's rewrite the Header function with React

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

When writing pure functions, try to follow these 3 rules:

1. The function should take in at least one argument
2. The function should return a value or another function
3. The function should not change or mutate any of its arguments

Pure functions are naturally testable

- Pure functions do not change anything about their environment and therefore do not require a complicated test setup.
- Everything a pure function needs to operate it accesses via arguments.
- When testing a pure function, you control the arguments, and thus you can estimate the outcome.

Data Transformations

How does anything change in an application if the data is immutable?

- Functional programming is all about transforming data from one form to another.
- Transformed copies of data (i.e. one dataset that is based upon another) can be produced using functions
- JavaScript has two core functions used to achieve this: **Array.map** and **Array.reduce**

Array.join: transform an array into a string

```
const schools = [
  "Yorktown",
  "Washington & Lee",
  "Wakefield"
]
console.log( schools.join(", ") )

// "Yorktown, Washington & Lee, Wakefield"
```

Array.filter: remove items from an array

```
const wSchools = schools.filter(school => school[0] === "W")

console.log( wSchools )
// ["Washington & Lee", "Wakefield"]
```

Array.map

```
const highSchools = schools.map(school => `${school} High School`)

console.log(highSchools.join("\n"))

//  Yorktown High School
//  Washington & Lee High School
//  Wakefield High School
```

Array.map: transform an array of objects into an array of strings

```
const highSchools = schools.map(school => ({ name: school }))

console.log( highSchools )

// [
//   { name: "Yorktown" },
//   { name: "Washington & Lee" },
//   { name: "Wakefield" }
// ]
```

Array.map in conjunction with Object.keys

```
const schools = {
  "Yorktown": 10,
  "Washington & Lee": 2,
  "Wakefield": 5
}

const schoolArray = Object.keys(schools).map(key =>
        ({
            name: key,
            wins: schools[key]
        })
    )

console.log(schoolArray)
```

```
// [
//   {
//     name: "Yorktown",
//     wins: 10
//   },
//   {
//     name: "Washington & Lee",
//     wins: 2
//   },
//   {
//     name: "Wakefield",
//     wins: 5
//   }
// ]
```

Array.reduce: transform an array into a primitive value

```
const ages = [21,18,42,40,64,63,34];

const maxAge = ages.reduce((max, age) => {
    console.log(`${age} > ${max} = ${age > max}`);
    if (age > max) {
        return age
    } else {
        return max
    }
}, 0)

console.log('maxAge', maxAge);
```

```
// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

Array.reduce: transform an array into an object

```
// {
//   "-xekare": {
//     title:"rad red",
//     rating:3
//   },
//   "-jbwsof": {
//     title:"big blue",
//     rating:2
//   },
//   "-prigbj": {
//     title:"grizzly grey",
//     rating:5
//   },
//   "-ryhbhsl": {
//     title:"banana",
//     rating:1
//   }
// }
```

```
const hashColors = colors.reduce(
    (hash, {id, title, rating}) => {
        hash[id] = {title, rating}
        return hash
    },
    {}
)

console.log(hashColors);
```

Array.reduce: transform arrays into completely different arrays

```
const colors = ["red", "red", "green", "blue", "green"];

const distinctColors = colors.reduce(
    (distinct, color) =>
        (distinct.indexOf(color) !== -1) ?
            distinct :
            [...distinct, color],
    []
)

console.log(distinctColors)

// ["red", "green", "blue"]
```

Higher-Order Functions

Higher-Order Functions

- Higher-order functions are functions that can manipulate other functions.
- They can take functions in as arguments, or return functions, or both.

Higher-Order Functions

- The first category of higher-order functions are functions that expect other functions as arguments. Array.map, Array.filter, and Array.reduce all take functions as arguments. They are higher-order functions.

How can we implement a higher-order function?

```
const invokeIf = (condition, fnTrue, fnFalse) =>
    (condition) ? fnTrue() : fnFalse()

const showWelcome = () =>
    console.log("Welcome!!!")

const showUnauthorized = () =>
    console.log("Unauthorized!!!")

invokeIf(true, showWelcome, showUnauthorized)    // "Welcome"
invokeIf(false, showWelcome, showUnauthorized)   // "Unauthorized"
```

Higher-Order Functions

- Higher-order functions that return other functions can help us handle the complexities associated with asynchronicity in JavaScript.
- Currying is a functional technique that involves the use of higher-order functions.
- Currying is the practice of holding on to some of the values needed to complete an operation until the rest can be supplied at a later point in time.
- This is achieved through the use of a function that returns another function, the curried function.

Currying

```
const userLogs = userName => message =>
    console.log(`${userName} -> ${message}`)

const log = userLogs("grandpa23")

log("attempted to load 20 fake members")
getFakeMembers(20).then(
    members => log(`successfully loaded ${members.length} members`),
    error => log("encountered an error loading members")
)

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> successfully loaded 20 members

// grandpa23 -> attempted to load 20 fake members
// grandpa23 -> encountered an error loading members
```

Recursion

Recursion

- Recursion is a technique that involves creating functions that recall themselves.
- In a challenge that involves a loop, a recursive function can be used instead.

Recursion: Example

```
const countdown = (value, fn) => {
    fn(value)
    return (value > 0) ? countdown(value-1, fn) : value
}

countdown(10, value => console.log(value));
```

```
// 10
// 9
// 8
// 7
// 6
// 5
// 4
// 3
// 2
// 1
// 0
```

Recursion

- Recursion is a good technique for searching data structures.
- You can use recursion to iterate through subfolders until a folder that contains only files is identified.
- You can also use recursion to iterate through the HTML DOM until you find an element that does not contain any children.

Recursion: Example

```
const deepPick = (fields, object={}) => {
   const [first, ...remaining] = fields.split(".")
   return (remaining.length) ?
       deepPick(remaining.join("."), object[first]) :
       object[first]
}
```

```
var dan = {
    type: "person",
    data: {
      gender: "male",
      info: {
        id: 22,
        fullname: {
          first: "Dan",
          last: "Deacon"
        }
      }
    }
  }

deepPick("type", dan);                      // "person"
deepPick("data.info.fullname.first", dan);  // "Dan"
```

Composition

Composition

- Functional programs break up their logic into small pure functions that are focused on specific tasks. Eventually, you will need to put these smaller functions together.
- Specifically, you may need to combine them, call them in series or parallel, or compose them into larger functions until you eventually have an application.
- When it comes to composition, there are a number of different implementations, patterns, and techniques.

Chaining

- Functions can be chained together using dot notation to act on the return value of the previous function

```
const template = "hh:mm:ss tt"
const clockTime = template.replace("hh", "03")
      .replace("mm", "33")
      .replace("ss", "33")
      .replace("tt", "PM")

console.log(clockTime)

// "03:33:33 PM"
```

Composition

- Chaining is one composition technique, but there are others. The goal of composition is to “generate a higher order function by combining simpler functions.

```
const both = date => appendAMPM(civilianHours(date))
```

Composition

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

```
const both = compose(
    civilianHours,
    appendAMPM
)

both(new Date())
```