Functional Programming

Introduction to functional programming

Lect. 9 15/04/24
Dr. Hassanain Al-Taiy

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.

Lect. 9 15/04/24
Dr. Hassanain Al-Taiy

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
```

Lect. 9 15/04/24
Dr. Hassanain Al-Taiy

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 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.

Lect. 9 15/04/24
Dr. Hassanain Al-Taiy

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.map

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

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

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

Lect. 9 15/04/24
Dr. Hassanain Al-Taiy

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
```

Lect. 9 15/04/24
Dr. Hassanain Al-Taiy

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"
```

Lect. 9 15/04/24
Dr. Hassanain Al-Taiy

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())
```