Functional Programming

Rich Web Application Technologies

Programming Paradigms

- A paradigm is a term meaning *a way of doing things*, such as programming
- From a Web App perspective, there are five programming paradigms that you will encounter - although, formally, many more exist

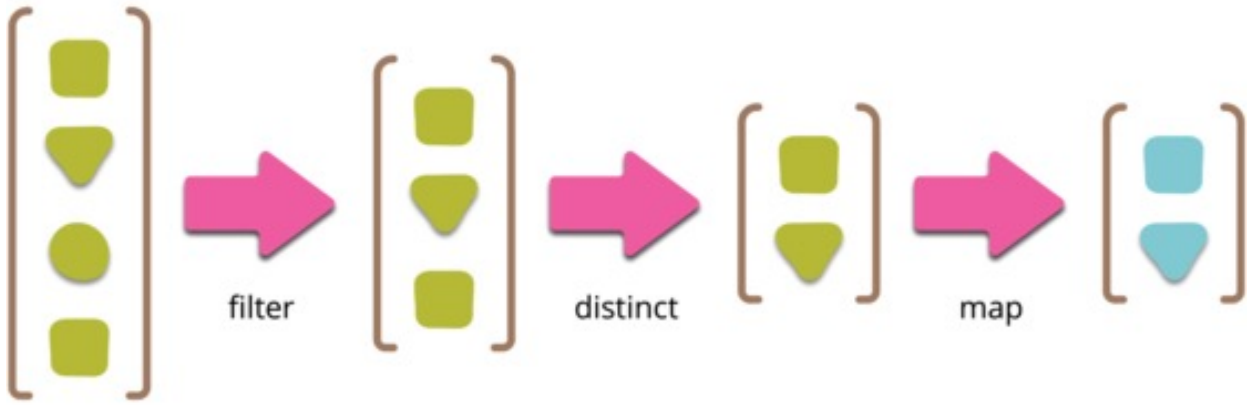
Imperative	Control flow as an explicit sequence of commands
Procedural	Imperative style with *procedure / function* call support
Declarative	Code states the desired result but not how to get it
Object-oriented	Sending messages to objects which have state and behaviour
Functional	Computation by pure function calls that avoid any global state - based on *Category Theory* in mathematics

Functional Programming

- The big idea in functional programming is the separation of the functions that operate on data from the data on which they operate
- This separation lends itself to the idea of function composition where in chained pipelines of functions operate on data transforming at each stage and passing the results onto the next phase
- The benefit is the resultant composition can implement potentially complex transformations in a very readable and maintainable way
- The functional expression of a problem is often more obvious and easier to reason about

Functional Programming

- A functional program works like the pipeline where a copy of the original data is processed through a collection of processes (functions), generating an output at the end. This pipeline takes raw material in one end, and gradually builds a product that comes out the other end



Functional Programming

- In functional programming, one of the main objectives is to make every data impossible to be changed. Therefore, functions should not promote any change on the values they process. This leads to the principle of Immutability.
- The ideal is to have a data structure that cannot be changed – such as tuples in Python.
- Immutability is useful in many situations. One of them is when we perform parallel computing, which consists in having a collection of computer processes happening at the same time. In this case, more than one process can be working on the same data, so immutability is essential to avoid bugs.
- Not changing the data naturally leads to a reduction on the number of bugs and make it easier to identify where certain errors come from.

Pure Functions

- A pure function operates on, but does not change, the values passed to it
- The result of calling a pure function with the same values should be the same very time
- That means that pure functions are not allowed to have *side effects*, i.e. alter something outside the function such as a variable
- The disadvantage of this process is to use extra memory to store the copy of the original data. However, pure functions are easier to debug as we can always expect to have the same output when using the same input.
- Pure functions produce new values as their results and don’t alter, in any way, their input values
    - I.e. pass-by-value semantics

Pure Functions

```
function change(val){
         val = val * 2;
         return val;
}
let x = 50;
let double = change(x);
console.log(x);
//output: 50
console.log(double);
//output: 100
console.log(x);
//output: 50
```

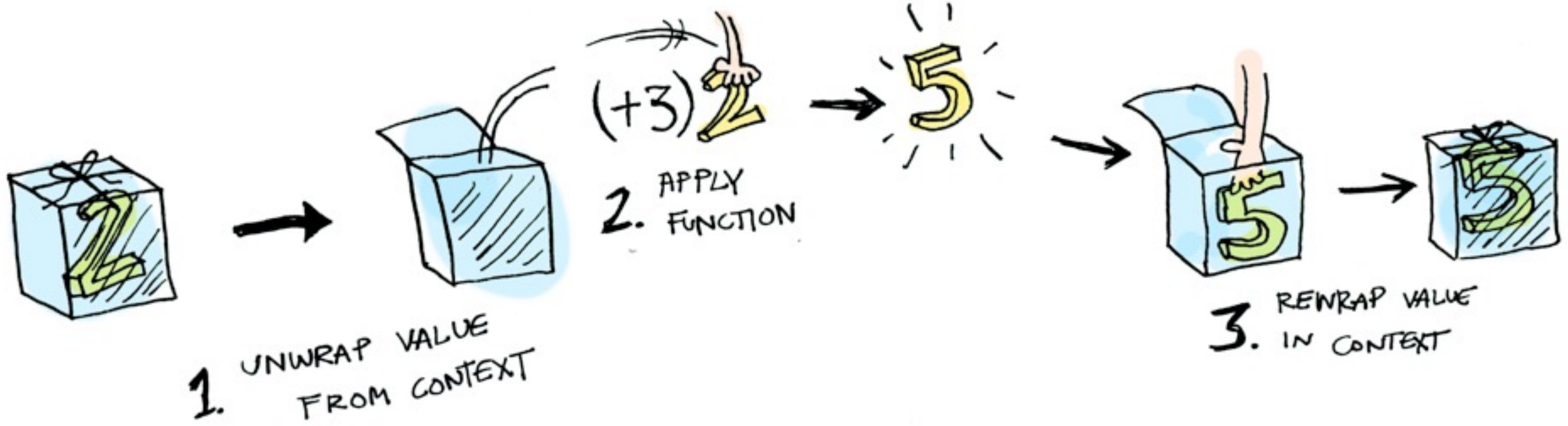
Functors

- A functor is a data object that can hold elements of any data type and which implements the *map* operation (a function)
- The functor's map() function takes, as an argument, another function and calls that function for each element of the functor resulting in a new functor
- `map()`'ing over a functor always produces a new functor of the same size
- In the following example, the functor starts as an array of strings and ends as an array of floating point numbers

```
// Javascript arrays are functors

['123', '456', '789']
     .map(s => parseInt(s))
     .map(n => n / 10)                // => [12.3, 45.6, 78.9]
```

Functors


(+3)
1. UNWRAP VALUE FROM CONTEXT
2. APPLY FUNCTION
3. REWRAP VALUE IN CONTEXT

Filter

- Filter calls a predicate function (one returning a true or false value) and grabs each value where said predicate returned true, returning them in a new collection
- A predicate is a function that returns a boolean value.
- An example: suppose we want to get the list of books with ratings higher than 4.5

```
let books = [{id: 1, name: 'book 1',
rating : 5}, {id: 2, name: 'book 2',
rating: 3}];
let result = books.filter(book =>
book.rating >= 4.5)
console.log(result);
```

Reduce

- Reduce executes a **reducer** function (that you provide) on each element of the array, resulting in a single output value.
- The reduce() method executes a provided function for each value of the array (from left-to-right).
- The return value of the function is stored in an accumulator (result/total).

```
<button onclick="myFunction()">Try it</button>
<p>Sum of numbers in array: <span id="demo"></span></p>
<script>
let numbers = [15.5, 2.3, 1.1, 4.7];
let getSum = (total, num) => total + Math.round(num);
let myFunction = (item) =>
document.getElementById("demo").innerHTML =
numbers.reduce(getSum, 0);

</script>
```

Higher Order Functions

- JS supports the concept of higher order functions wherein a function can accept another function as an argument or return a function as a value
- In other words functions are just values, like objects or primitives

```
const myHandlerFunc = (event) => {
  // do something with event ...
}
document.addEventHander('click', myHanderFunc);
```

Function Closures

- A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
function init() {
        let name = 'Mozilla'; // name is a local variable created by init
        function displayName() { // displayName() is the inner function, a
closure
                alert(name); // use variable declared in the parent function
}
        displayName();
}
        Init();
```

Function Closures Cont..

- Closures are a way to implement data hiding, which can lead to modules and other nice features.
- The key concept is when you define a function, it can **refer** to not only its own local variables in block scope, but also everything outside the context of the function. Ie count is accessible in nested myFunction below

```
function newCounter() {
          let count = 0;
          return myFunction() {
                    count = count+1;
                    return count;
          }
}
```

Function Currying

- A curried function is a function where the evaluation of its arguments is translated into a sequence of functions which evaluates one argument at a time, so instead of someFunction(a,b), it becomes someFunction(a)(b)

- The intermediate functions can form partial evaluations for later use

```
// Conventional approach
const multiply = (x, y) => x * y;

const curriedMultiplier = x => y => x * y;
const discount = curriedMultiplier(0.98);
console.log(discount(1000)); // => 980
```

Function Currying

- A curried function is a function where the evaluation of its arguments is translated into a sequence of functions which evaluates one argument at a time, so instead of someFunction(a,b), it becomes someFunction(a)(b)

- Instead of taking the arguments all at once, a curry function takes the first argument, returns a new function that takes the second argument, and so on, until al arguments have been fulfilled
- Currying creates nested functions depending on the number of arguments provided
- Curried functions are very efficient in handling events

Function Currying

```
//regular function
function sum(a, b, c) {
        return a + b + c;
}
console.log(sum(1,2,3)); // 6

//curried version
function sum(a) {
        return (b) => {
                return (c) => {
                        return a + b + c
                }
        }
}
console.log(sum(1)(2)(3)); // 6
```

Lazy Evaluation

- Lazy evaluation is the idea that a function defers computing a result until it absolutely needs to
- This makes memory or compute intensive functions more efficient
- Consider below, what's wrong with it?

```
const someValue = expensiveFunction();
// Tons of operations that don't involve someValue
console.log(someValue);
```

- We ran a very demanding function at the beginning to use its returned value at the end of the program. Blocking call. Potentially block browser

Lazy Evaluation Cont..

- This simple problem would be fixed by reordering. However, not all this is simple.
- This makes memory or compute intensive functions more efficient
- In JS, the ES6 *generator function* can be used to achieve this
- What are ES6 generator functions?
- a generator is a function that **can stop midway** and then continue *from where it stopped.*
- A generator object is an iterator. Iterators are lazy because the next value in the sequence is only created/calculated when it is consumed.

Generator Functions

```
function * generatorFunction() {
console.log('This will be executed first.');
yield 'Hello, '; //the yield keyword pauses the generator
function
console.log('I will be printed after the pause');
yield 'World!'; //the yield keyword resumes the generator
function
}
const generatorObject = generatorFunction();
console.log(generatorObject.next().value);
console.log(generatorObject.next().value);
console.log(generatorObject.next().value);
```

Immutability

- The functional programming paradigm enforces the notion of immutability wherein, once set, a variable’s value never changes - only new copies are created
- So we don’t really have variables but rather symbol bindings

```
const a = Object.freeze({
          foo: 'Hello’,
          bar: 'world’,
          baz: '!'
});
a.foo = 'Goodbye';
// Error: Cannot assign to read only property 'foo' of object Object
```

Structural Sharing

- Because functional languages use copies of values rather than mutated values you might be concerned about the inefficient use of memory
- For example a 10,000 element array needing fully copied just to change the value of one of its elements
- In practice, most functional programming languages implement their data structures using a technique known as structural sharing
- The new and old copies share most of their contents and only the differences distinguish them
- Under the hood, the implementation rely on data structures like trees which incur a small but practically insignificant overhead to maintain

Summary

- Of the many programming paradigms you will encounter in computer science, the functional paradigm, which is based on *Category Theory* in mathematics, emphasises composable operations on data
- Not surprisingly, functions are the basic building block
- Functors are ***any***-type data collections which implement a `map()` operation capable of return new functors (of possibly different types)
- Functional programming allows solutions to be easily composed, be more declarative and obvious, be easier to reason about and be easier to get right and debug if wrong