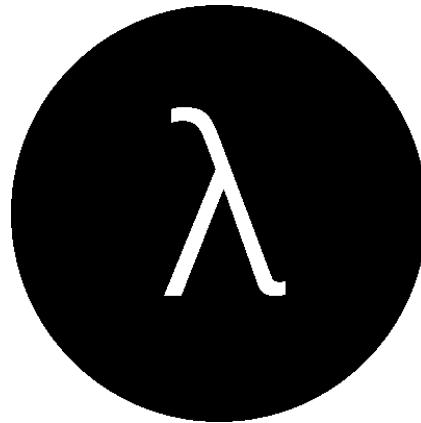
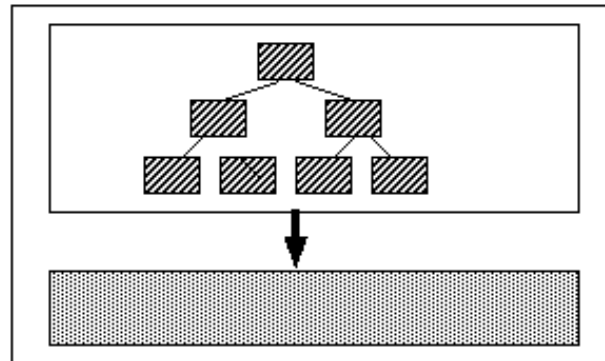


Functional Programming



Programming Paradigms

Procedural Languages

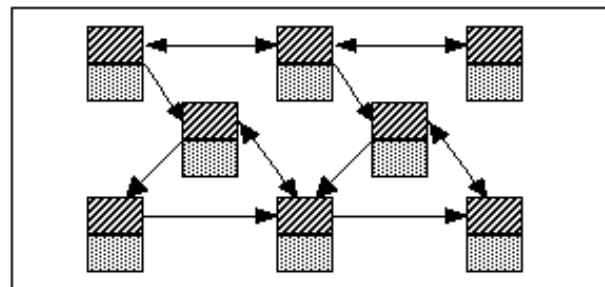


Computation involves code operating on Data

 Code

 Data

Object-Oriented Languages



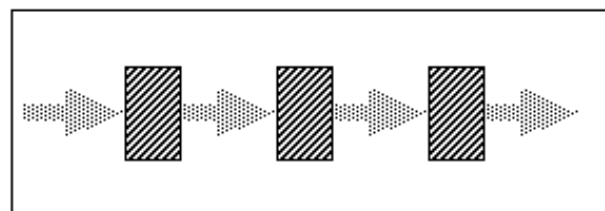
An object encapsulates both code and data

 Code

 Data

Computation involves objects interacting with each other

(Pure) Functional Languages



Data has no independent existence

 Code (Functions)

Computation involves data flowing through functions



What is FP?

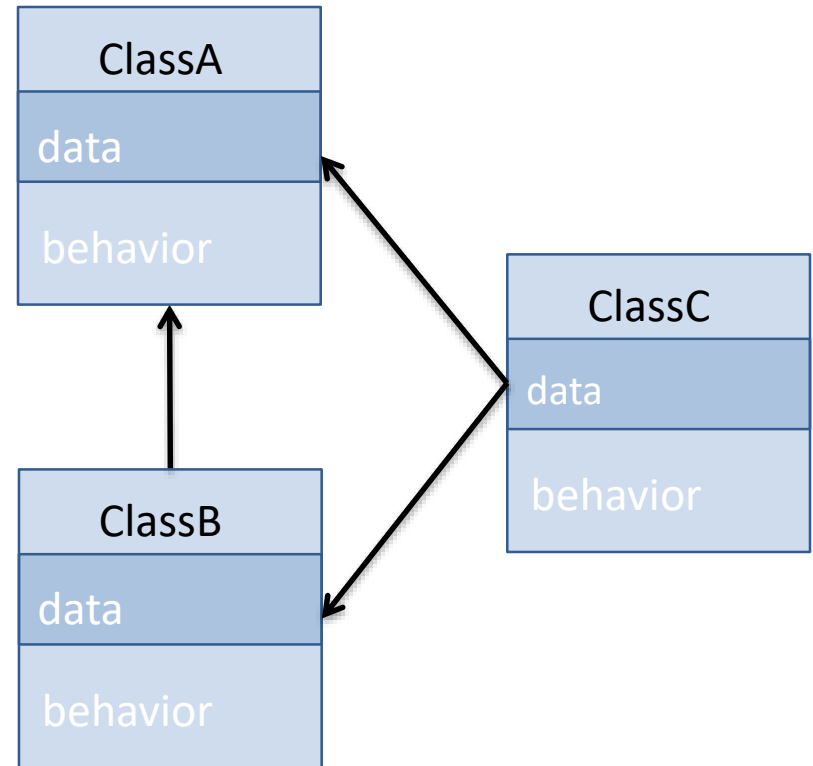
- It's a declarative programming style where functions have no side effects and input data is read only
 - Basic unit of abstraction is the *function*
 - *Pure functions avoiding externally observable side effects*
 - Process read-only data (immutable). If something needs to be changed - create new instance instead
- Why FP?
 - Reduce complexity and allows creating modular, testable and reliable apps
 - Programs are clearer, more concise and compact
 - Enable parallel processing

Functional Style

Imperative/OO	Functional
Tell the computer what to do	Describe what you want done
Abstraction via objects/ data structures	Abstraction via functions
Re-use via classes/inheritance	Re-use via composition
<p>Iteration/looping</p> <pre>function factorial(n) { let result = 1; while (n > 1) { result *= n; n--; } return result; }</pre>	<p>Recursion: Function calls itself until a condition is met and it stops</p> <pre>function factorial(n) { if (n < 2) return 1; else return n * factorial(n - 1); }</pre>

The OO World

- Unit of work: Classes
- Data and behavior tightly coupled (i.e. shared mutable state)
- Unit testing is more challenging



The FP World

Unit of work: Function

Data and behavior loosely coupled

Unit testing is easier

function doWork(objectA): objectB

function doMoreWork(objectB): objectC

function displayResults(objectC)

FP Key Features & Principles

Features

- Anonymous functions / Lambda (λ) expressions
- First-class functions: function is just like a value
- Higher-order functions
- Closure

Principles

- Pure functions
- Singular functions

Anonymous functions / Lambda (λ) expressions

// Anonymous function

```
let square = function(a) {  
    return a ** 2;  
}
```

// or Lambda expression

```
square = (a) => a ** 2;
```

// display true

```
console.log( typeof square === 'function' )
```


First-class functions

- JavaScript treats functions as first-class citizens as they can be:
 - Stored in variables
 - Passed to functions
 - Returned from functions

```
let mathFuncs = [  
  function(a) { return a * 2 },  
  function(b) { return b * b },  
  function(c) { return c + 2 }  
];  
console.log( mathFuncs[1](3) ); // => 9
```

Pure functions => No side effects

- Pure Functions:
 - Have no side effects (memory or I/O)
 - Depend on arguments only
- **Referential transparency** - if a function is called with the same arguments, it will always return the same value
- No data dependency between pure functions:
 - Order and parallelization will not effect how the function runs



Example

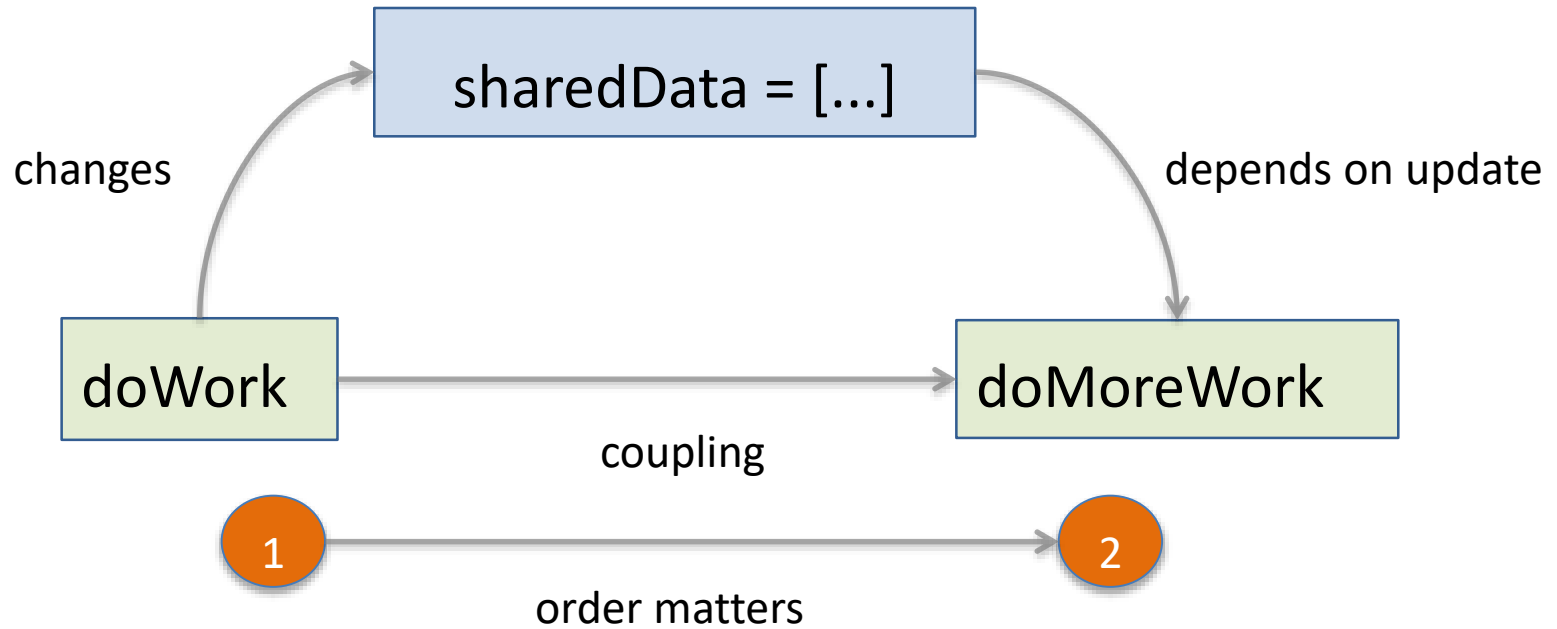
//Not Pure

```
let name = 'Ali';  
function greet() {  
    console.log(`Hi, I'm ${name}`);  
}
```

//Pure:

```
function greet(name) {  
    retrn `Hi, I'm ${name}`;  
}
```

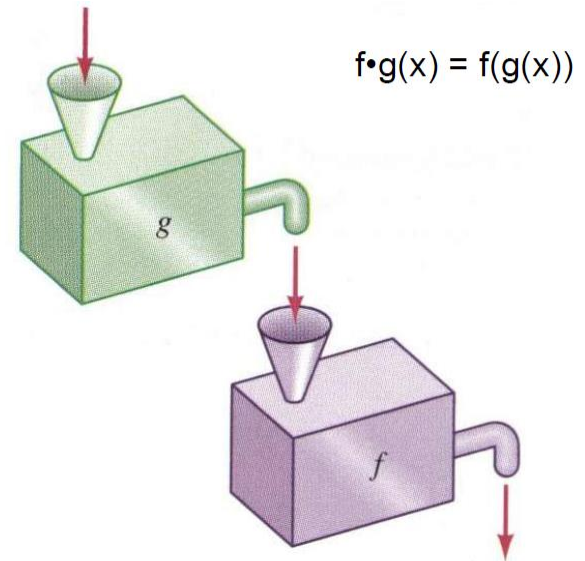
Side effects



Singular Functions

- Singularity principle: functions are supposed to perform only one task
- Simple functions typically have fewer arguments than complex functions
- Simple functions are easy to test, but also **composable** and **chainable**

Composition is the backbone of modularity in FP



Functions Chaining example

```
const encode = input => input.split(' ')  
    .map(str => str.toLowerCase())  
    .join('-');
```

```
console.log( encode('I am happy') ); // 'i-am-happy'
```

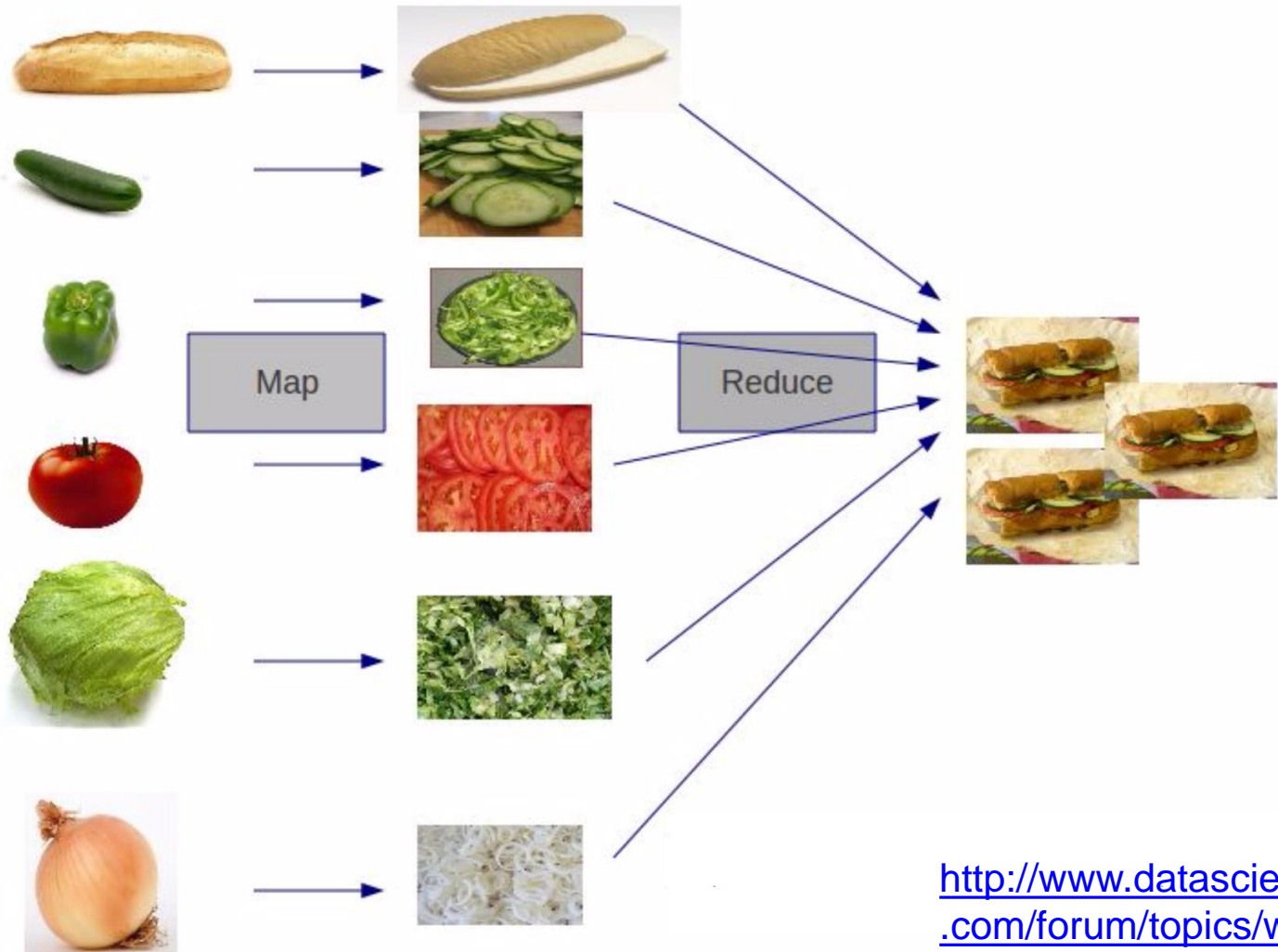
Higher-Order Functions

- Functions can be inputs/outputs
- **Higher-order function:** are functions that can either take other functions as arguments or return them as results

Receive function, return value

- The receiving function provides the syntactic skeleton of the computation
- The sent function provides the semantics for the computation
- Implementation of the “Strategy design pattern”
e.g., array **map** and **reduce** functions
=> Passing parameterized behavior into an algorithm

Map, Reduce and Filter are examples of High-Order functions



<http://www.datasciencecentral.com/forum/topics/what-is-map-reduce>

Receive value, return function

- Keep the received value in a closure
- Return a new function that uses that value

```
function multiplyBy(factor) {  
    return function (num) {  
        return num * factor;  
    }  
}  
  
let times10 = multiplyBy(10);  
console.log( times10(5) ); // 50  
  
let numArray = [1,2,3].map( times10 );  
console.log( numArray );
```

Closure

- Functions can nest: A function can be defined inside another function
 - Computation done in an inner function can access data found in the outer function
 - Such access creates a construct called closure
- Closures remain alive even when the execution of the outer function has finished execution
- An object is data with functions. A closure is a function with data.

Closure

```
const digitName = function () {  
    let names = [  
        "zero", "one", "two", "three", "four",  
        "five", "six", "seven", "eight", "nine"  
    ];  
  
    return function (n) {  
        return names[n];  
    };  
};
```

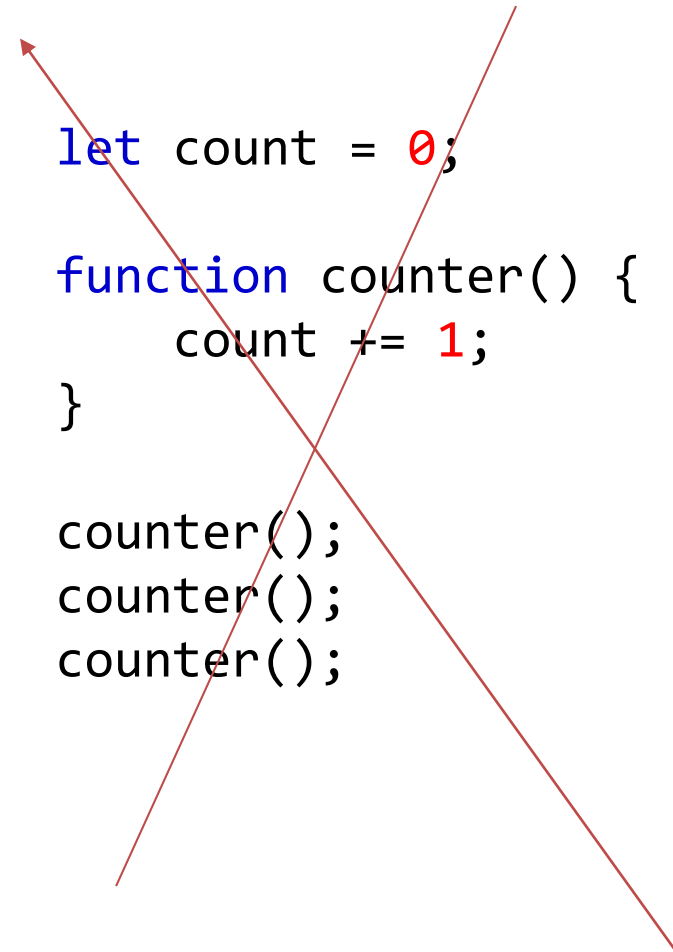
```
const getDigitName = digitName();  
console.log ( getDigitName(3) );    // "three"  
console.log ( getDigitName(9) );    // "nine"
```

Closure (Encapsulation use case)

*//Data Encapsulation through Closure
//Avoids the use of Global Variables*

```
function makeCounter() {  
    let i = 0;  
    return function () {  
        return ++i;  
    }  
}
```

```
const counter = makeCounter();  
console.log( counter() ); // 1  
console.log( counter() ); // 2
```

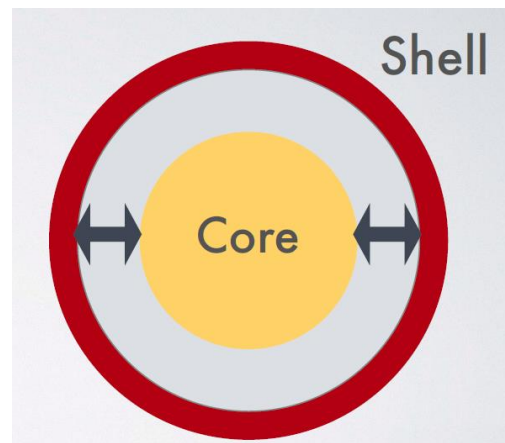


```
let count = 0;  
  
function counter() {  
    count += 1;  
}  
  
counter();  
counter();  
counter();
```

Another example <https://jsfiddle.net/vnkuZ/>

Combining OOP AND FP

- **Imperative Shell** - OOP, IO, side-effects:
 - Imperative and have side-effects (e.g., deals with IO)
 - Thin layer
- **Functional Core** - FP, logic, pure functions:
 - Deals with immutable data and use pure functions
 - Contains application logic



Summary

- Immutability - inputs are read-only, less bugs
- Pure functions - no side-effects, less bugs
- Composition - reusing functions
- Don't loop over arrays use high order functions such as **.map**, **.reduce** and **.filter**