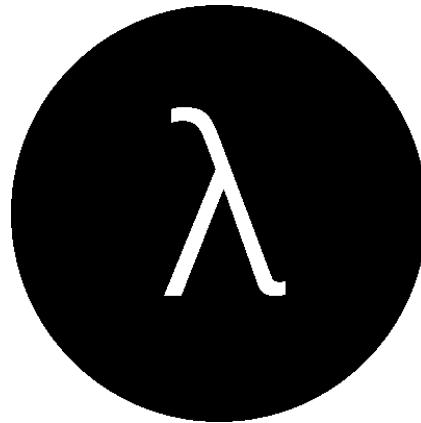
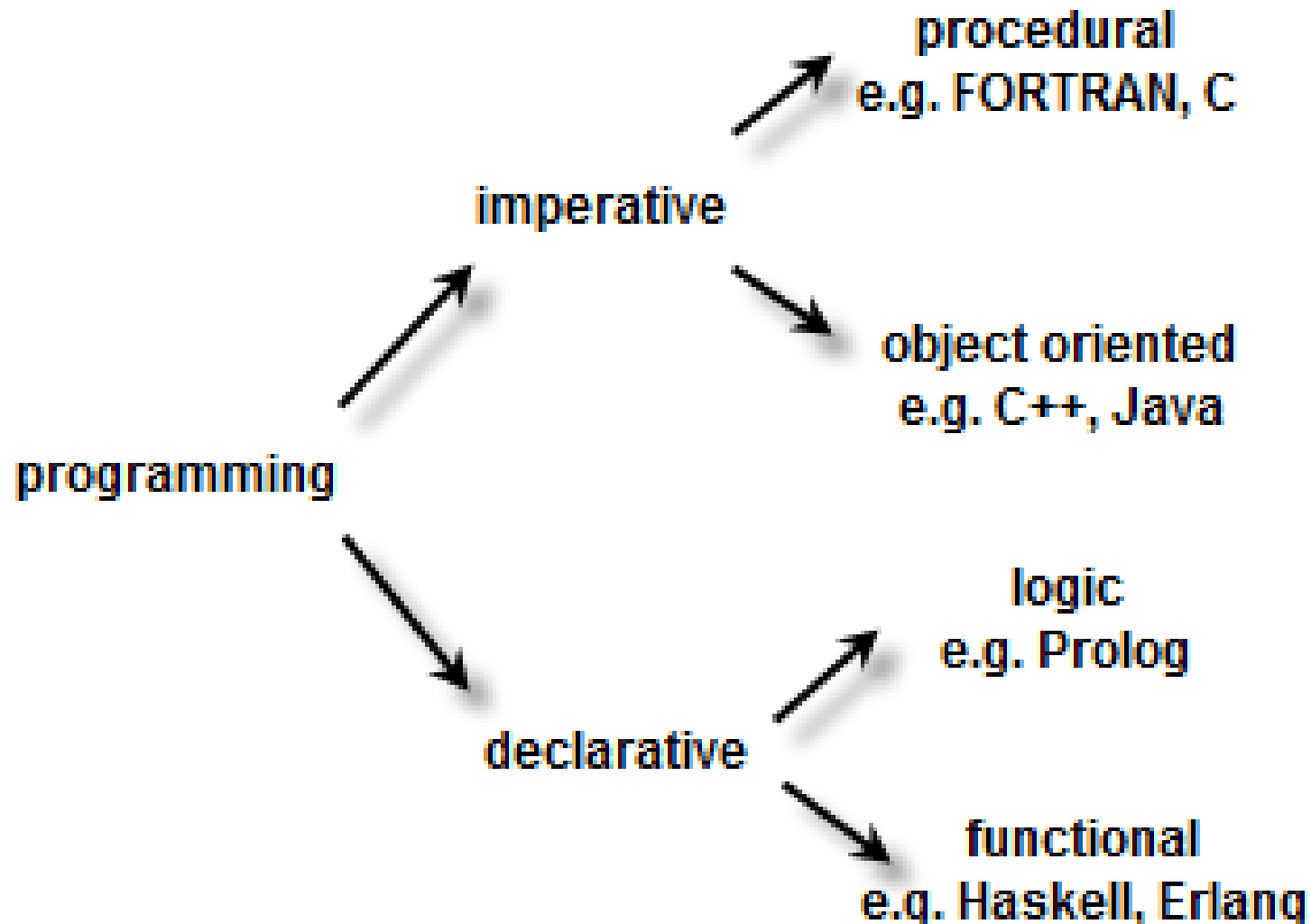


Functional Programming

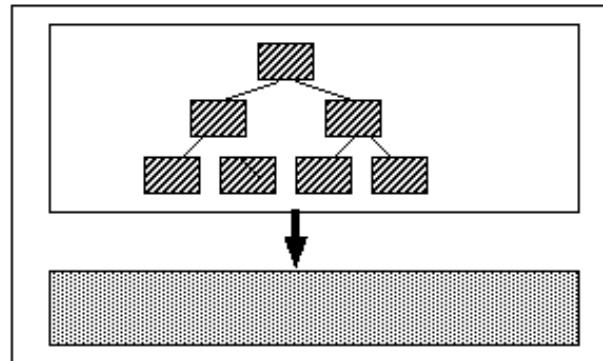


Programming Paradigms



Programming Paradigms

Procedural Languages

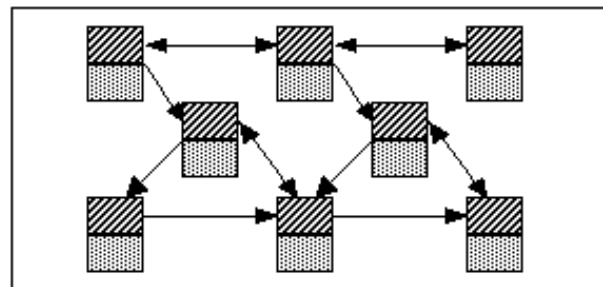


Computation involves code operating on Data

 Code

 Data

Object-Oriented Languages

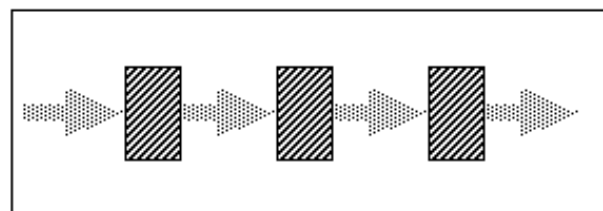


An object encapsulates both code and data


 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?

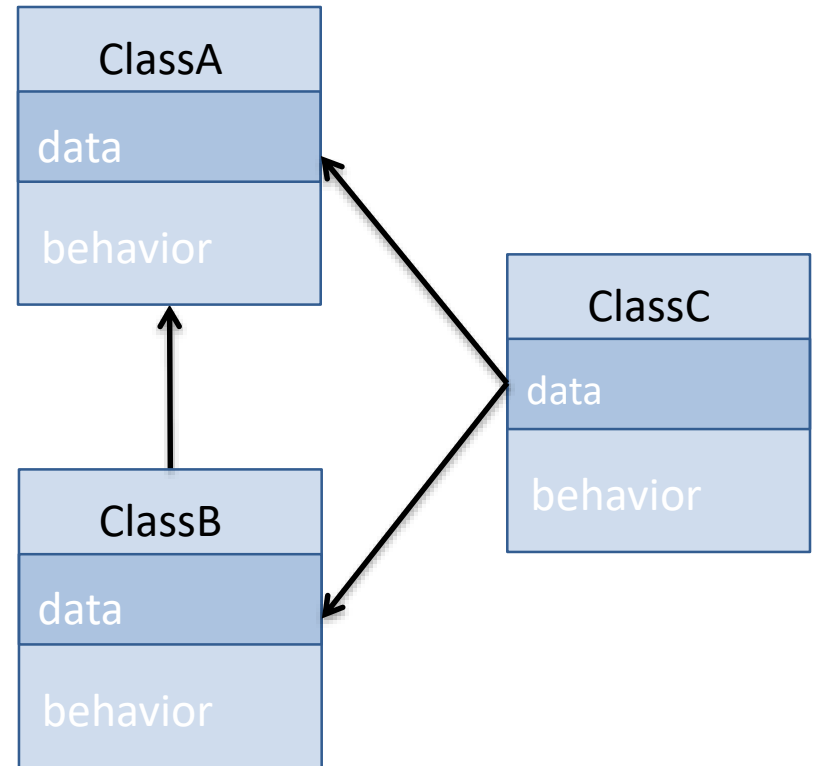
- It's a declarative programming style where functions have no side effects and input data is read only
 - Basic unit of abstraction is the *function*
 - *Pure functions avoiding externally observable side effects*
 - Process read-only data (immutable). If something needs to be changed - create new instance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

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
Iteration/looping	Recursion: Function calls itself until a condition is met and it stops
<pre>function factorial(n) { let result = 1; while (n > 1) { result *= n; n--; } return result; }</pre>	<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 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)
```

Paradigm shift

- Eliminate externally observable side effects
- Control (reduce or eliminate) mutations
- Write declaratively rather than imperatively
 - Describe WHAT a program does Not HOW to do it
- Everything is a value (even functions)
- Functions ALWAYS return values
- Recursion as looping mechanism (eliminate loops)

Mandatory FP Features

- Anonymous functions / function literals (λ)
- First-class functions
- Higher-order functions
- Lexical closure

Anonymous Functions / Function Literals

```
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
- This allows us to use a very cool functional programming technique called **Currying**
- Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument

First-Class Functions

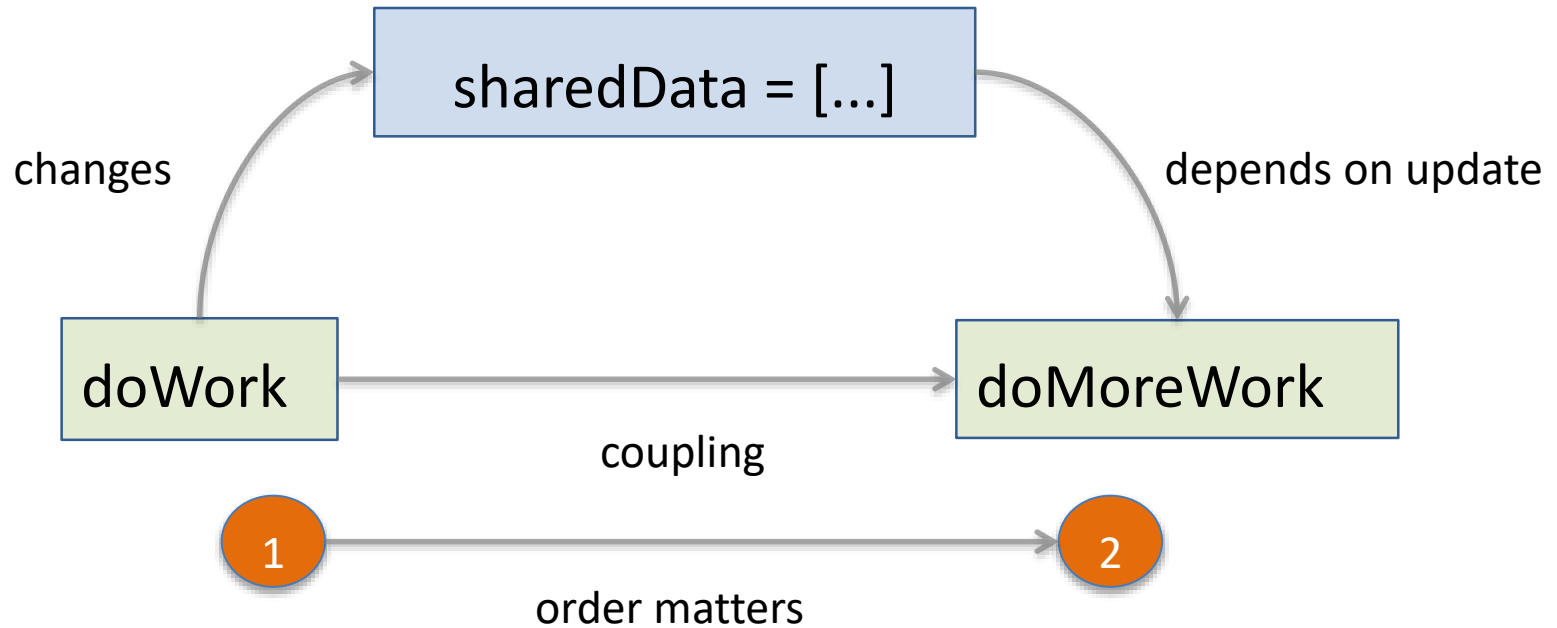
```
let mathFuncs = [  
  function(a) { return a * 2 },  
  function(b) { return b * b },  
  function(c) { return c + 2 }  
];  
console.log( mathFuncs[1](3) ); // => 9
```

Pure functions => No side effects

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



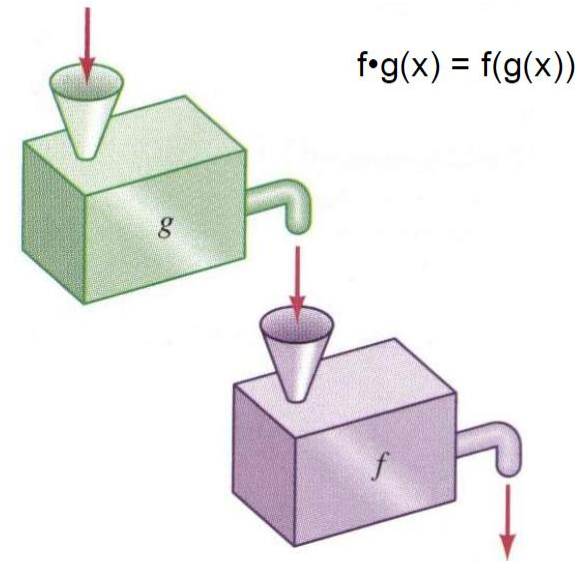
Side effects



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



Higher-Order Functions

```
function multiplyBy(factor) {  
  return function (num) {  
    return num * factor;  
  }  
}  
  
let times10 = multiplyBy(10);  
console.log( times10(5) ); // 50
```

```
let numArray = [1,2,3].map( multiplyBy(10) );  
console.log( numArray );
```

- **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 calculation
- Implementation of the “Strategy design pattern”
e.g., array **map** and **reduce** functions

Receive value, return function

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

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 an 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

```
let digitName = (function (n) {  
    let names = [  
        "zero", "one", "two", "three", "four",  
        "five", "six", "seven", "eight", "nine"  
    ];  
  
    return function (n) {  
        return names[n];  
    };  
})();  
  
console.log ( digitName(3) );    // "three"
```

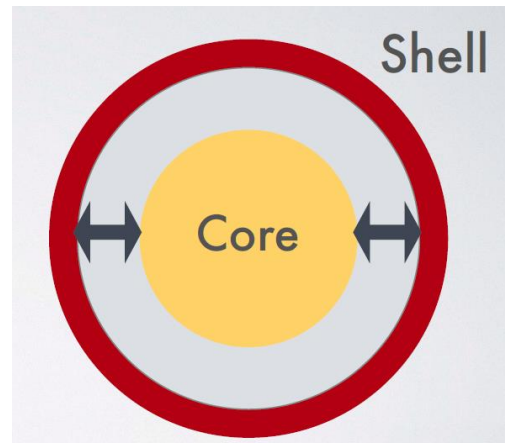
Closure

//Data Encapsulation through Closure

```
function makeCounter() {  
    let i = 0;  
    return function () {  
        return ++i;  
    }  
}  
let counter = makeCounter();  
console.log( counter() ); // 1  
console.log( counter() ); // 2
```

Combining OOP AND FP

- **Imperative Shell** - OOP, IO, side-effects:
 - Imperative. Deals with IO
 - Thin layer
- **Functional Core** - FP, logic, pure functions:
 - deals with immutable data and pure functions
 - contains application logic



Summary

- Immutability - everything is read-only, less bugs
- Pure functions - no side-effects, less bugs
- Composition, currying - reusing functions
- Closures are your friends
- Don't loop over arrays
- Don't fear recursion

Online JavaScript Resources

- Best JavaScript tutorial:
 - <http://www.w3schools.com/js>
- Mozilla JavaScript learning links
 - <https://developer.mozilla.org/en-US/learn/javascript>
- Node.js School
 - <https://nodeschool.io/>