agent
technology
center

Architecture of Software Systems

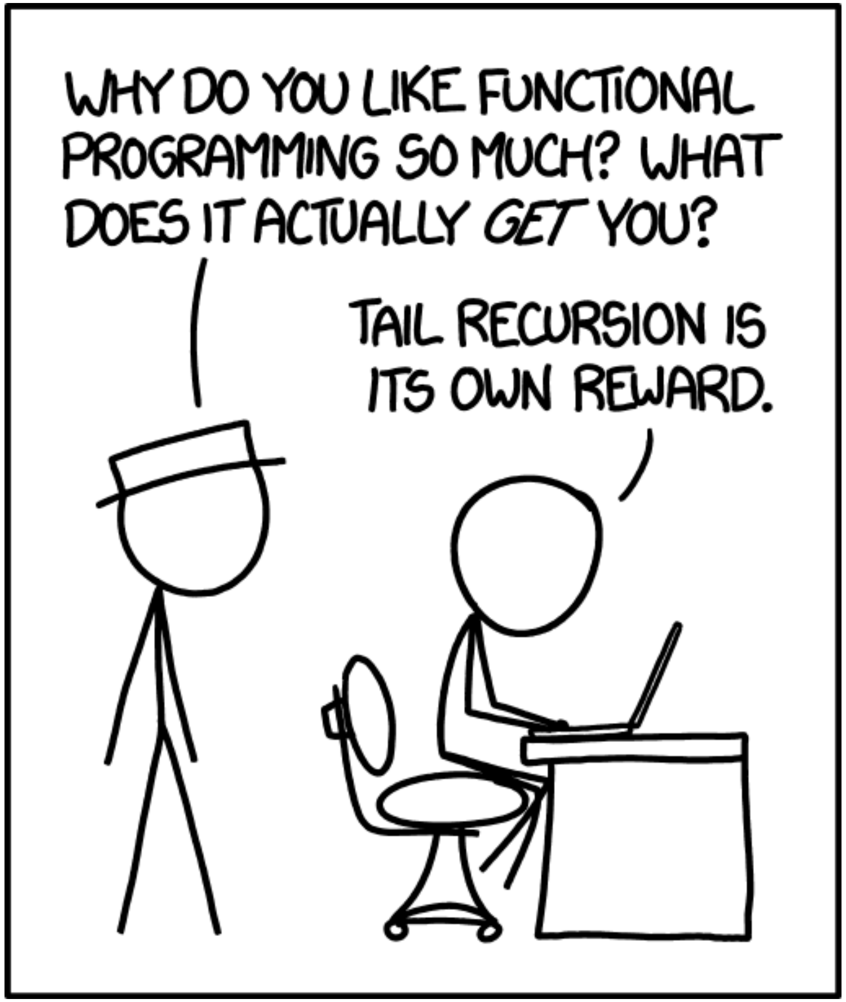
Functional Programming

Jan Michelfeit

2017

Functional Programming

- Why should I care?
- What is it?
- Practical functional programming
- Functional principles in software architecture
- Advanced topics
- Takeaways



WHY FUNCTIONAL PROGRAMMING?

What’s wrong with these snippets?

```
DateFormat format = new SimpleDateFormat("yyyy-MM-dd");
ExecutorService threadPool = Executors.newFixedThreadPool(5);
List<Future<Date>> results = new ArrayList<>();
for(int i = 0 ; i < 10 ; i++){
  results.add(threadPool.submit(
    () -> format.parse("2017-10-22")));
}
for(Future<Date> result : results){
  System.out.println(result.get());
}
```

```
class Person {
  private String name;
  // …
  @Override
  public boolean equals(Object o)
  @Override
  public int hashCode() { … }
}
Set<Person> set = new HashSet<>();
Person p = new Person();
set.add(p);
p.setName("Daniel");
```

```
void printItems(Iterator<String> items) {
  int itemsCount = Iterators.size(items);
  for (int i = 0; i < itemsCount; i++) {
    System.out.println(items.next());
  }
}
```

What’s wrong with these snippets?

```
DateFormat format = new SimpleDateFormat("yyyy-MM-dd");
ExecutorService threadPool = Ex                    adPool(5);
List<Future<Date>> results = ne
for(int i = 0 ; i < 10 ; i++){
  results.add(threadPool.submit
    () -> format.parse("2017-10-22")));
}
for(Future<Date> result : results){
  System.out.println(result.get());
}
```

Does this throw an exception?

Does this print anything?

```
void printItems(Iterator<String> items) {
  int itemsCount = Iterators.size(items);
  for (int i = 0; i < itemsCount; i++) {
    System.out.println(items.next());
  }
}
```

```
class Person {
  private String name;
  // …
  @Override
  public boolean equals(Object o)
  @Override
  public int hashCode() { … }
}
Set<Person> set = new Hash
Person p = new Person();
set.add(p);
p.setName("Daniel");
```

Does set.contains(p) returns true?

What’s wrong with these snippets?

```
DateFormat format = new SimpleDateFormat("yyyy-MM-dd");
ExecutorService th
List<Future<Date>>
for(int i = 0 ; i
  results.add(thr
    () -> format.p
}
for(Future<Date> result : results){
  System.out.println(result.get());
}
```

Works

… or NumberFormatException: multiple points

… or NumberFormatException: For input string: "101101.E101E22”

… or ArrayIndexOutOfBoundsException: -1

(DateFormat is not thread-safe)

```
void printItems(Iterator<String> items) {
  int itemsCount = Iterators.size(items);
  for (int i = 0; i < itemsCount; i++) {
    System.out.println(items.next());
  }
}
```

NoSuchElementException
(Iterator reuse)

```
class Person {
  private String name;
  // …
  @Override
  public boolean equals(Object o)
  @Override
  public int hashCode() { … }
}
Set<Person> set = new HashSet<>();
Person p = new Person();
set.add(p);
p.setName("Daniel");
```

set.contains(p) returns false
because of hashCode() behavior

What’s wrong with these snippets?

- What did the examples have in common?
  - Mutable state
- Do you like global variables?
- Should String be mutable?

Mutable state can make things
really hard to reason about, debug, …

Functional Programming

- Basic idea:
  - avoid **mutable state** and **side-effects**
  - **compose** programs from **functions** that always give the **same result for the same arguments**
- Advantages
  - leads to code that is **safer**, **modular**, **composable**
  - easier to **reason** about, **test**, and **debug**
  - well suited for **parallelization**

WHAT IS FUNCTIONAL PROGRAMMING?

Basic Terminology

Immutability

- **Data structure (object) is immutable if it's (observable) state cannot be modified after it is created.**
- Examples (Java): String, ImmutableList (Guava library), any class with all fields final & immutable

Referential transparency

- An **expression** *e* is referentially transparent if for all programs *p* every **occurrence** of *e* in *p* **can be replaced with the result of evaluating** *e*, without affecting the observable result of *p*.
- E.g., replace all occurrences of “1+2” with “3”
- Allows creation of local state, as long as it's not observable

Basic Terminology

Pure function

- **Function** f is pure if the expression $f(x_1, \ldots, x_n)$ is referentially transparent for all referentially transparent inputs $x_1, \ldots, x_n$
- Function output **may depend only on arguments, not on external mutable state**
- Typically "no side-effects" - only observable output should be the return value
- Example: mathematical functions (sin, max, +, ...)

Side effect

- Modifies state outside of its scope, or has an observable interaction with its calling functions or the outside world
- examples
  - reassigning a variable, modifying a data structure in place
  - throwing an exception or halting with an error (depends on context)
  - user interaction
  - reading/writing a file

Programming Paradigms
Imperative Programming

- Program: sequence of **commands changing state**
- Commands usually don't have value
  => data exchanged through *state*

- “Functions” − subroutines (unit of modularity)
- Function invocation can give different results at different times
  - depending on the state of the executing program
- Closer to hardware / traditional programmer thinking

Programming Paradigms
(Pure) Functional Programming

- Models computation as the **evaluation of expressions, using pure functions** and immutable data
- Avoids mutable state and side-effects

Programming Paradigms
(Pure) Functional Programming

- Based on Lambda Calculus
  - Turing-complete computation model
- (Closer to *human* thinking (unless obfuscated))
- Special case of **declarative programming**
  - expresses the logic of a computation without describing its control flow
  - describing *what* the program must accomplish, rather than *how*
  - e.g., HTML, Excel, most parts of functional languages

Typical Features of Functional Languages

Functions as first-class citizens

- can be passed as arguments to other functions or be returned as a result of a function
- functions accepting and/or returning functions are **higher-order functions**

Function
(String) => Int

```
List("list", "of", "words").map(word => word.length)
// List(4,2,5)
```

```
val f = (x: Int) => -x
val g = f.compose((x: Int) => x * x)
g(3) // -9
```

Typical Features of Functional Languages

Lazy evaluation

- expression evaluation delayed until the value is needed
- evaluation order is irrelevant with pure functions (no side-effect can ever change expression value)

```
-- infinite data structure
numsFrom n = n : numsFrom (n+1)
take 5 (numsFrom 0)
-- result: [0,1,2,3,4]
```

```
Source.fromFile("numbers.txt")
  .map(line => line.toInt)
  .exists(n => n < 0)
// file is read only until first negative number
```

Functional Programming Advantages

Parallel with structured vs. unstructured programming:

- Structured programming forbids
  - goto
  - multiple entries or exits from a block of code

=> seemingly less power?

But:

- encourages modular design => simpler, smaller modules
  - easier, quicker to code
  - easier to reuse
  - easier to test
- mathematically more tractable (easier to analyze, tooling)

Functional Programming Advantages I

FP forbids side-effects in functions, mutable variables

- seemingly less power?

But:

- Lack of side effects **eliminates** a major source of **bugs**
- Evaluation order irrelevant
  - **parallelization friendly**
  - enables practical **lazy evaluation**
- Higher order functions & lazy evaluation are a **new "glue" for composition** of modules

Functional Programming Advantages II

FP forbids side-effects in functions, mutable variables

- seemingly less power?

But:

- encourages even **better modularization**, **composability**
- easier **testing** (input values determine output, avoids setup of state)
- **declarative** (captures intention), usually more conscious and understandable (unless obfuscated)
- immutable data structures - **thread-safe, cacheable**
- mathematically more tractable
- easier debugging, ...

PRACTICAL
FUNCTIONAL PROGRAMMING

Sum - Imperatively

```
private int sum(List<Integer> list) {
    int result = 0;
    for (int i = 0; i < list.size(); i++) {
        result += list.get(i);
    }
    return result;
}
```

- *Why is this not functional?*
- Mutable variables *result, i*
  (but no externally visible state)

Sum - functionally

```
private int sum(List<Integer> list) {
    int result = 0;
    for (int i = 0; i < list.size(); i++) {
        result += list.get(i);
    }
    return result;
}
```

Functional version?

(Hint: think of mathematical induction)

```
def sum(list: List[Int]): Int = {
  if (list.empty) 0
  else list.head + sum(list.tail)
}
```

Count - functionally

```
def sum(list: List[Int]): Int = {
  if (list.empty) 0
  else list.head + sum(list.tail)
}
```

Functional version of computing list size?

```
def count(list: List[Int]): Int = {
  if (list.empty) 0
  else 1 + count(list.tail)
}
```

Generalizing…

```
def sum(list: List[Int]): Int = {
  if (list.empty) 0
  else list.head + sum(list.tail)
}
```

- *What was specific for sum()?*
  - 0 and +

Fold

```
def foldRight(
              list: List[Int],
              initial: Int,
              op: (Int, Int) => Int): Int = {
 if (list.isEmpty) initial
 else op(
         list.head,
         foldRight(list.tail, initial, op)
        )
}
```

Using Fold

```
def sum(list:List[Int]) = foldRight(list, 0, (a, b) => a + b)

def prod(list:List[Int]) = foldRight(list, 1, (a, b) => a * b)

def count(list:List[Int]) = foldRight(list, 0, (a, b) => 1 + b)

def exists(list:List[Int], condition: Int=>Boolean) =
        foldRight(list, false, (n, b) => condition(n) || b)

def copy(list:List[Int]) =
        foldRight(list, List(), (n, list) => n +: list)
```

Fold - Observations

- *foldRight()* is a higher-level function
- *sum(), count(),* … are composed from smaller **reusable building blocks**
- *Is this possible in Java?*
  - yes, but functional language made the modularization easier and more obvious
  - **functional thinking** is more important than a particular language
- Notice the similarity with algebra (monoid)

Typical Operations on Collections

Imperative programming: add(), get(), **set()/put()**

Functional Programming *(examples for Traversable[T] in Scala − e.g., List[T]; simplified)*

- **map**(f: A=>B) : Traversable[B]
- **flatMap**(f: A=>Traversable[B]) : Traversable[B]
- **filter**(f: A=>Boolean) : Traversable[B]
- **foldLeft**(zero: B)(op: (B, A) => B): B
- **foldRight(), fold()**
- **head** : A
- **tail** : Traversable[A]
- **groupBy(f**: A=>K): Map[K, Traversable[A]]
- **++**(t: Traversable[A]) : Traversable[A]
- ... many more - take(), drop(), takeWhile(), slice(), zip(), grouped(), ...
- **Methods do not change inputs, but return new collections as result !**
- **Computation may be lazy**

Collections Operations - Examples

Proper Scala syntax for sum(), prod(), …

```
def sum(list:List[Int]) = list.foldRight(0)((a, b) => a + b)
def prod(list:List[Int]) = list.foldRight(1)((a, b) => a * b)
def count(list:List[Int]) = list.foldRight(0)((a, b) => 1 + b)
def exists(list:List[Int], condition: Int=>Boolean) =
        list.foldRight(false)((n, b) => condition(n) || b)
def copy(list:List[Int]) =
        list.foldRight[List[Int]](List())((n, list) => n +: list)
```

Collections Operations - Examples

Print all middle names (JavaScript)

```
var names = [
  "James Paul McCartney", "William Bradley Pitt",
  "Laura Witherspoon", "Hannah Dakota Fanning" ];

names
    .map(function(name) { return name.split(" "); })
    .filter(function(parts) { return parts.length >= 3; })
    .map(function(parts) { return parts[1]; })
    .forEach(log)
// forEach not functional !!
```

Collections Operations - Examples

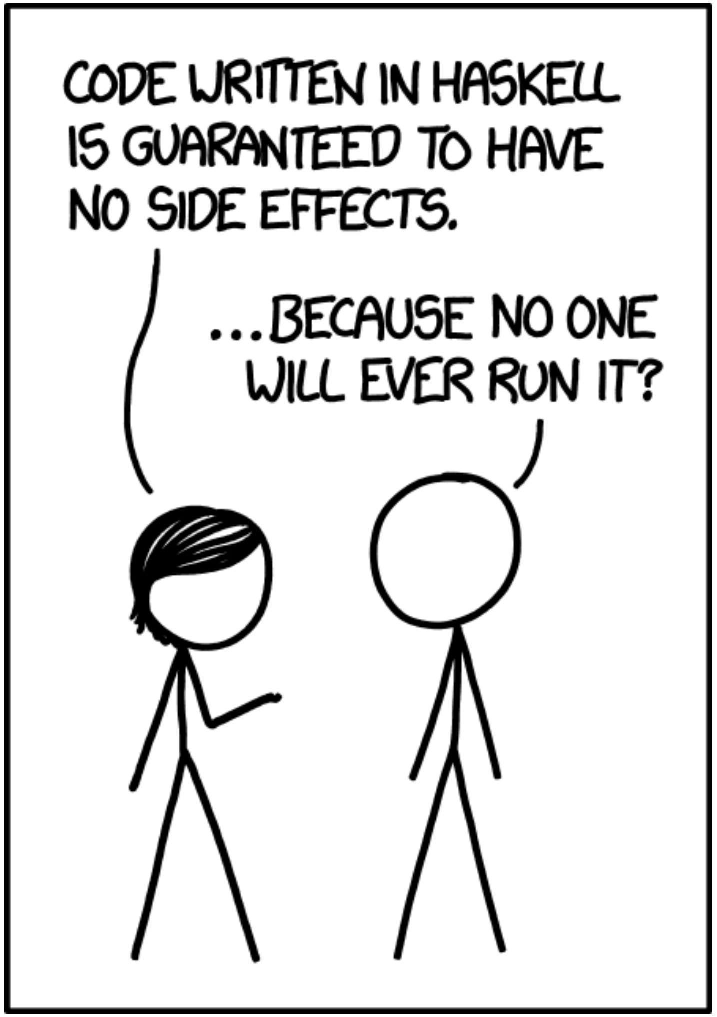
Get length of the longest word (Scala)

```
val lines = List("Scala collections", "have nice methods")
lines
  .flatMap(line => line.split(" ").toList)
   // List(Scala, collections, have, nice, methods)
  .map(word => word.length)
   // List(5, 11, 4, 4, 7)
  .fold(-1)(Math.max)
   // 11
```

```
lines
  .map(line => line.split(" ").toList)
   // List(List(Scala, collections), List(have, nice, methods))
  .map(x => x.length)
  // List(2, 3)
```

FP-oriented Languages

- Haskell
- Scala
- F#
- Erlang
- R
- …
- Some features in traditional OO languages
  - Java 8, LINQ, C++11



Going Parallel - Imperative

How to make this parallel?

```
private int sum(List<Integer> list) {
    int result = 0;
    for (int i = 0; i < list.size(); i++) {
        result += list.get(i);
    }
    return result;
}
```

General approach:

1. (Recursively) split to subtasks
2. Execute subtasks in parallel
3. Re-combine results of subtasks

Going Parallel - Imperative

Does this work?

```
private int result;
public int sum(List<Integer> list) {
    result = 0;
    for (Integer n : list) {
        threadPool.submit(() -> result += n);
    }
    // ... wait for tasks finished ...
    return result;
}
```

No! Access to mutable variable `result` is not synchronized

- Results will be non-deterministic

Going Parallel - Functional

In Scala:

```
list.par.sum
```

```
list.par.fold(0)((a,b) => a+b)
```

Internally:

1. Recursively splits list
2. Folds each part in parallel
3. Applies fold operation to partial results

Going Parallel - Functional

- Combining partial results may be non-deterministic
  - E.g. List(1,2,3,4):
    (((0 + 1) + 2) + 3) + 4
    or ((0 + 1) + 2) + ((0 + 3) + 4)
- => Fold operation must be **associative**
  - (a+b)+c = a+(b+c)
  - remember monoid
- Notice we used `fold()` instead of `foldRight()`
  - `fold()` expects associative operation

- Exercise:
  try parallelizing imperative & functional QuickSort

Going Parallel - Takeaways

- Parallelization can get complex, frameworks help
- Side effects & parallelism may lead to non-determinism
- Parallel access to mutable state requires **synchronization**
  - e.g. `AtomicInteger`, `ConcurrentHashMap`, `synchronized`, `...`
  - synchronization is costly
- Non-associative re-combination of results may lead to non-determinism
  - (commutativity is not required)

- Parallelization with immutable data is much easier & efficient
  - avoids synchronization

FUNCTIONAL PRINCIPLES
IN SOFTWARE ARCHITECTURE

Functional Principles

- Statelessness
- Immutability & parallelization
- Functional APIs

MapReduce

- Framework for parallel processing of big data (multi-terabyte)
  on large clusters of commodity hardware
- Executes MapReduce jobs
  1. Split input data (on a distributed file system) into records
  2. Process each record with a *map* task
     (considering data locality)
  3. Merge results with a *reduce* task

- `map: (K1,V1)=> List[(K2,V2)];`
- `reduce: (K2, List[V2]) => List[V3]`

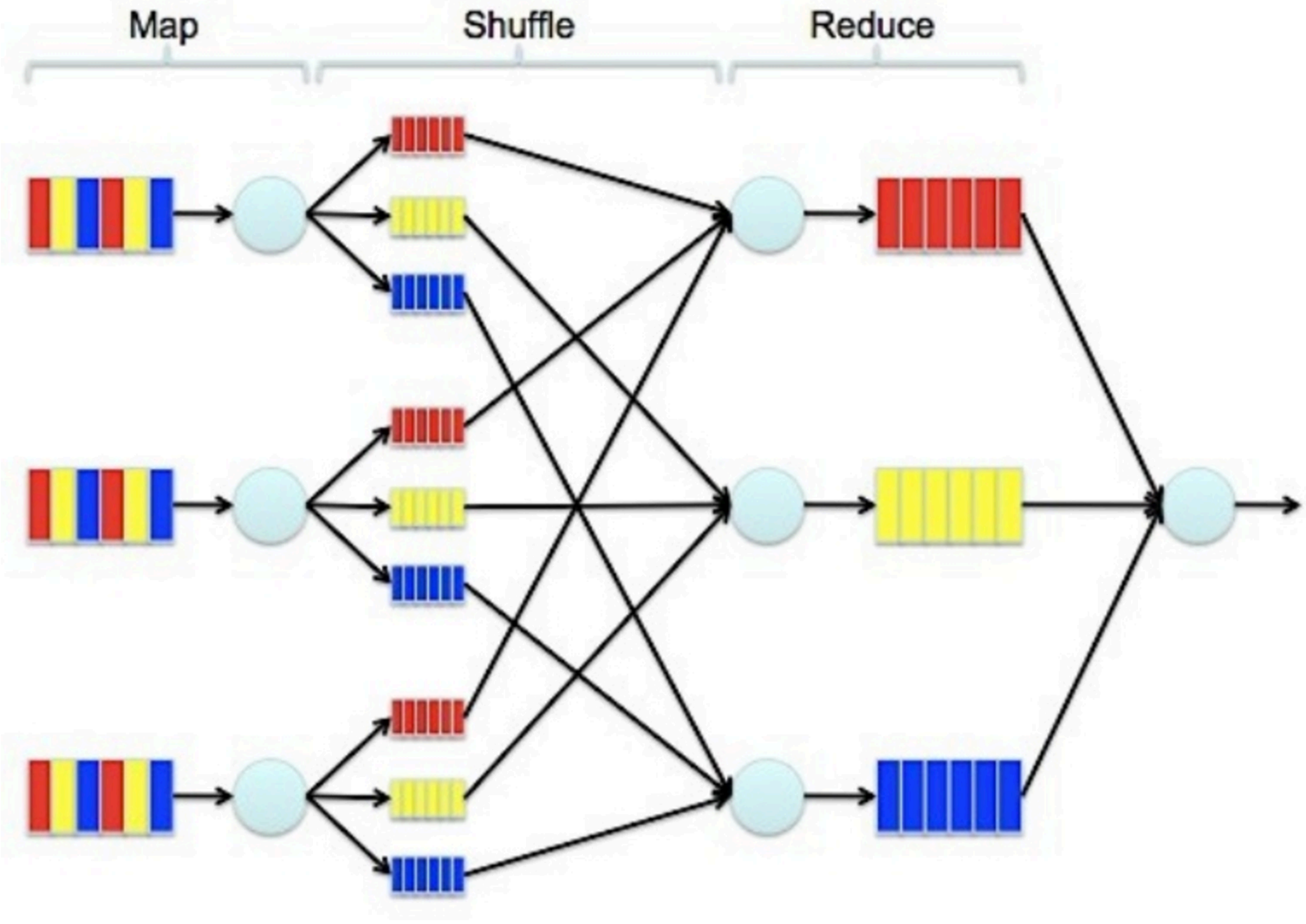


Image from - http://mm-tom.s3.amazonaws.com/blog/MapReduce.png

MapReduce - Observations

- Distributed computation
  - no shared memory => cannot share state
- Map and reduce are higher-level functions
  - mappers and reducers are first class citizens, preferably immutable
- The principle can be **applied** even **when programming with threads**
  - basically parallel *map()* + *fold()* (or *reduce()*)

MapReduce – Example

- Word count
  - Map: for each word emit tuple (*word, 1*)
  - Reduce: Sum 1s for each word
- Higher level frameworks often used in practice - e.g. Scalding

```
class WordCountJob(args: Args) extends Job(args) {

  TypedPipe.from(TextLine(args("input")))
  .flatMap { line => line.toLowerCase.split("\\s+") }
  .groupBy { word => word } // use each word for a key
  .size // in each group, get the size
  .write(TypedText.tsv[(String, Long)](args("output")))
}
```

Stateless Components

- **Service Statelessness principle**
  - “Guidelines in favor of making the service stateless by shifting away the state management overhead from the services to some other external architectural component” (wiki)
- State can be externalized to a dedicated component (database, distributed in-memory cache, ...)
- Why
  - lower component complexity
  - makes **high availability** and **horizontal scaling** easier
    - E.g., consider a customer-facing web server and sticky sessions
- Why not
  - can increase overall system complexity
  - affects performance, response times

Functional Framework APIS

E.g., D3.js

```
var paragraphs = document.getElementsByTagName("p");
for (var i = 0; i < paragraphs.length; i++) {
  var paragraph = paragraphs.item(i);
  paragraph.style.setProperty("color", "white", null);
}
```

```
d3.selectAll("p").style("color", "white");
```

```
d3.selectAll("p").style("color", function(d, i) {
  return i % 2 ? "#fff" : "#eee";
});
```

ADVANCED TOPICS

Side-effects in Practice

- FP discourages side-effects
  - But what about the user? What can the program do?

- Some languages allow side-effects as programmer's responsibility
- Pure FP languages (e.g., Haskell) allow only explicit side-effects wrapped as **monads**

Monad

- Represents a **computation** with a **sequential structure** and **possible side-effect**
  - Defines what it means to chain operations together
- Allows refactoring side-effects out of functions
- “Promise” to produce a value of a certain type
- Allows separating computation description and execution

Monad

- Formally: type constructor, bind & return operations, monad laws [1], [2]
- Informally:
  - generic data structure `M[A]`
  - with constructor `of(): A => M[A]`
  - Operation `flatMap() : (A=>M[B]) => M[B]`
- E.g., `Optional<T>` in Java:
  - `static <T> Optional<T> of(T value)`

  -
    ```
    Optional<U> flatMap(
                Function<T, Optional<U>>
    mapper)
    ```

Monad - Examples

- I/O Monad in Haskell – wraps all computations with a global effect
  - `getChar :: IO Char`
    - pure function that returns a side-effecting computation
    - does not necessarily cause an immediate effect
    - **=> can be used in another pure function**
  - `putChar :: Char -> IO ()`
- Scala
  - `Option[T], Future[T], Set[T], List[T]`
- LINQ operators - e.g.,
  `M<T> SelectMany(this M<S> src, Func<S, M<T>> f)`

Monad - Examples

```
object OrderService {
  def loadOrder(username: String): Future[Order] = ???
}
object PurchaseService {
 def purchase(order : Order) : Future[PurchaseResult] = ???
 def logPurchase(result:PurchaseResult) : Future[LogResult]
= ??? }


val logResult = OrderService.loadOrder("customerUsername")
    .flatMap(order => PurchasingService.purchase(order))
    .flatMap(result => PurchaseService.logPurchase(result))
```

Replacing Imperative Loops

```
public int factorial(int n) {
  int result = 1;
  while (n > 0) {
    result *= n--;
  }
  return result;
}
```

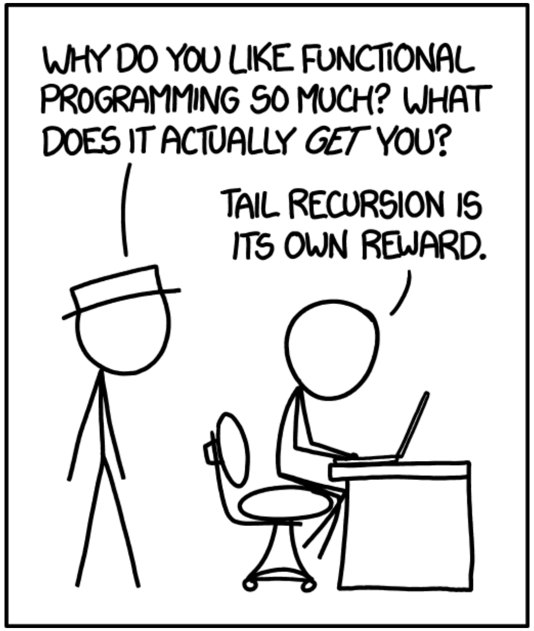
How to replace mutable variables?

```
def factorial(n: Int) = {
    if (n == 1) 1
    else n * factorial(n - 1)
}
```

Recursion

```
def factorial(n: Int) = {
    if (n == 1) 1
    else n * factorial(n - 1)
}
```

- Evaluation:
  - factorial(4)
  - 4 * factorial(3)
  - 4 * (3 * factorial(2))
  - 4 * (3 * (2 * factorial(1)))
  - 4 * (3 * (2 * 1))
  - 4 * (3 * (2))
  - 4 * (6)
  - 24
- Stack size dependent on input argument



Tail Recursion

- **Tail recursive function** - recursive action as the last action
  - variables on the stack will no longer be used
- => Compiler can replace recursion with a loop
- *How to pass intermediate results?*
  - **“Accumulator”** argument

```
def factorial(n: Int, accumulator: Int = 1) = {
    if (n == 1) accumulator
    else factorial(n – 1, accumulator * n)
}
```

- Evaluation:
  - factorial(4, 1) -> factorial(3, 4) -> factorial(2, 12) -> factorial(1, 24) -> 24

CONCLUSION & TAKEAWAYS

Functional Programming - Comparison

	Imperative	Functional
Basic unit	Command	**Expression**
Computation	Command execution	Expression evaluation
Mutability	Mutable data	**Immutable** data
Side-effects	Allowed	**Externalized** via Monads
Function	Unit of modularity; can depend on external state	Pure function; **cannot depend on external state**
Program describes	*How* to accomplish a task	***What*** to accomplish

Cons & Pitfalls

- Functional language may be more distant to *a traditional programmer's* thinking
  - Functional code can be obfuscated by the programmer (beware of “write-only” code)
- Possible stack overflows if recursion used wrong
  - **Use tail recursion if the language supports it**
- Negative impact on performance if used wrong
  - Non-tail recursion has a cost
  - Memory allocations & garbage collection of many immutable objects (but: short-lived immutable objects may be better for GC than long-lived mutable objects)
  - http://flyingfrogblog.blogspot.cz/2016/05/disadvantages-of-purely-functional.html
- Some algorithms hard to write efficiently
  - Some problems solvable in $O(n)$ time with state mutation can require $\Omega(n \log n)$ time in pure, non-lazy functional language
- Difficult to predict the time and space costs of evaluating a lazy functional program
- **But remember: premature optimization is the root of all evil [1], [2]**

Why Functional Programming

- Features
  - Avoids mutable state and side-effects
  - New "glue" for composition: functions as first-class citizens, lazy evaluation
  - Declarative
- Parallelization friendly
- Thread-safe, cacheable data structures
- Pure functions are safer, modular, composable
- Easier to reason about, test, and debug

Further Reading

- Functional Programming in Scala (Chiusano & Bjarnason) (book)
- Series of articles by Libor Škarvada
- Why Functional Programming Matters (Hughes) (paper)
- Functional programming in JavaScript in Eloquent JavaScript (book chapter)
- Haskell documentation
- Comparison of Functional, Declarative and Imperative programming on Stack Overflow
- Programming in Scala (2008, book available online)
- Scala cheat sheets: one, another

The End …


I COULD WRITE SOME GETTERS AND SETTERS
OR USE ONE LITTLE PUBLIC FIELD INSTEAD
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
public string Name;
COMPILE