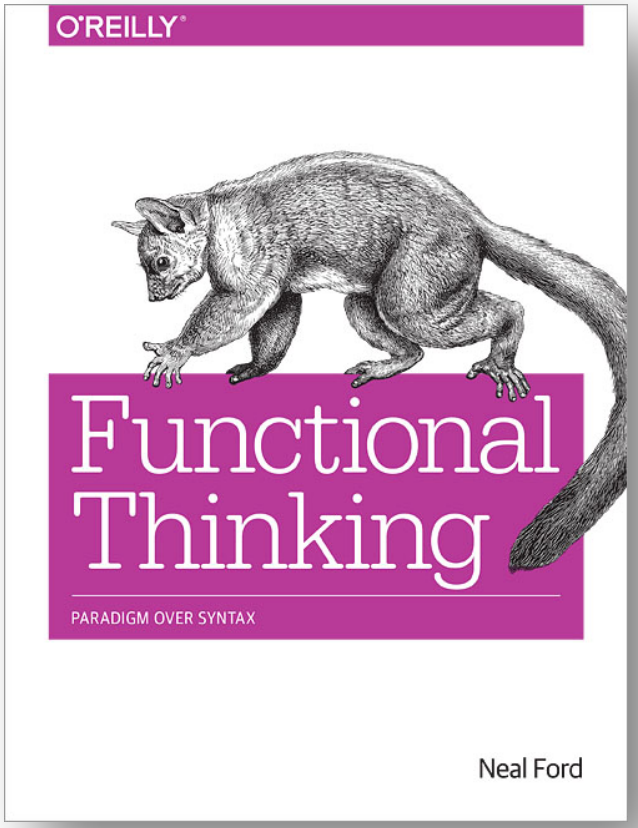
ThoughtWorks®

NEAL FORD

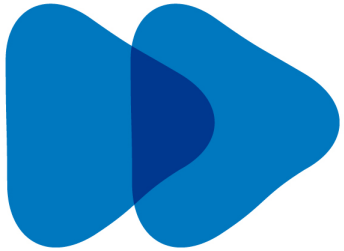
Director / Software Architect / Meme Wrangler

@neal4d *nealford.com*



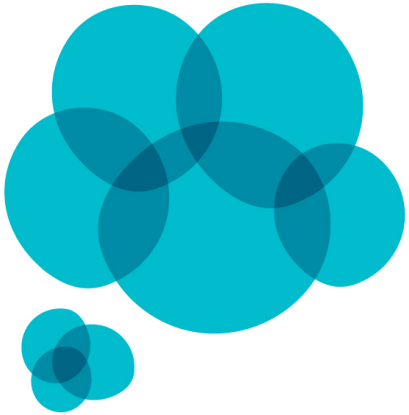
FUNCTIONAL THINKING

in 8 , , Groovy , Scala

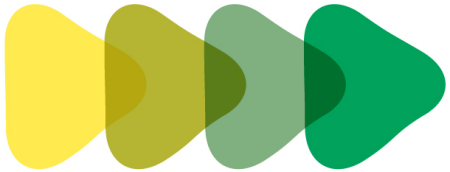


Shift

Multiparadigm

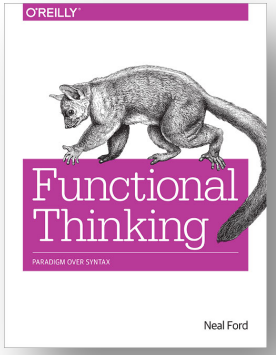


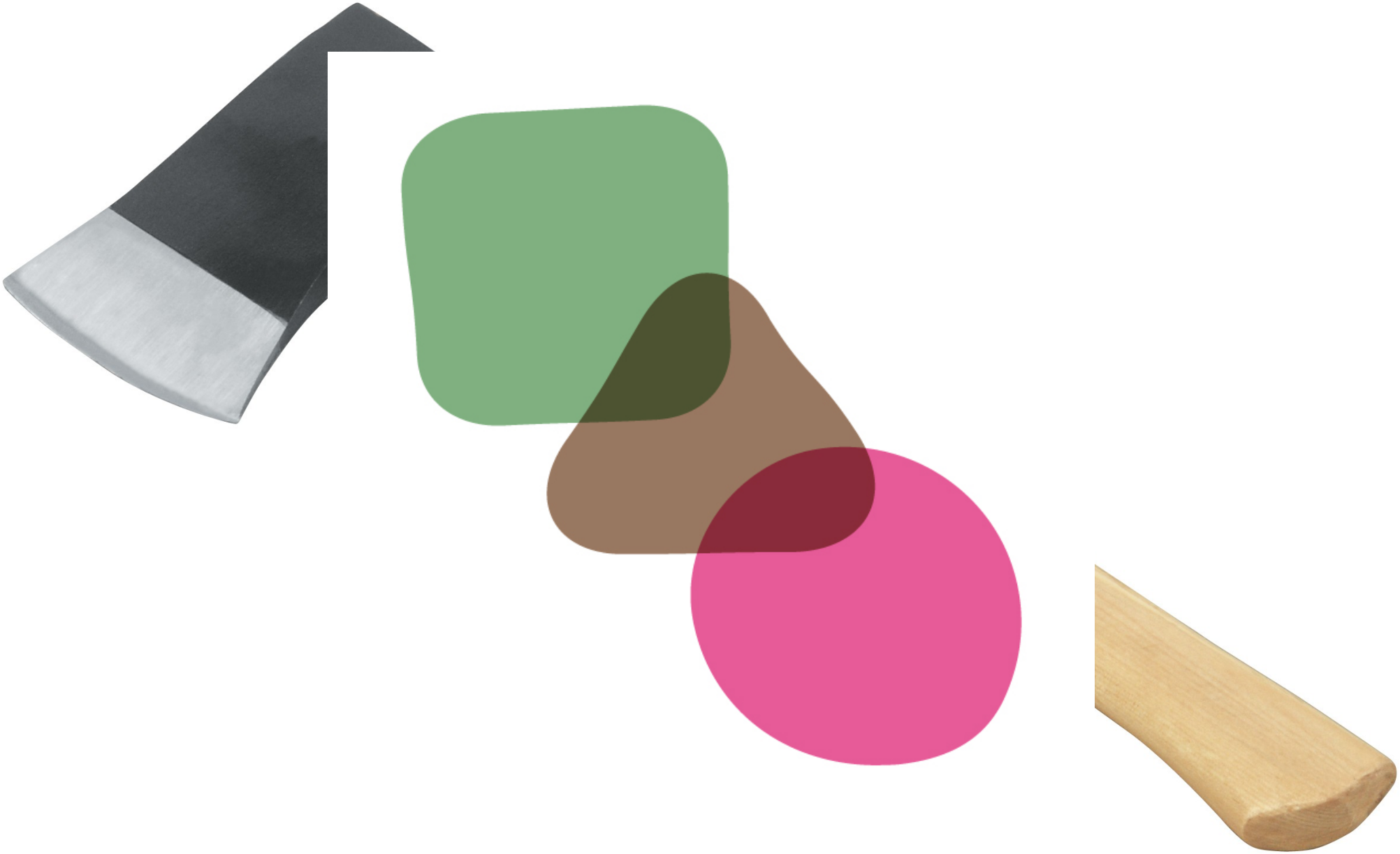
Smarter
(not Harder)



Advance

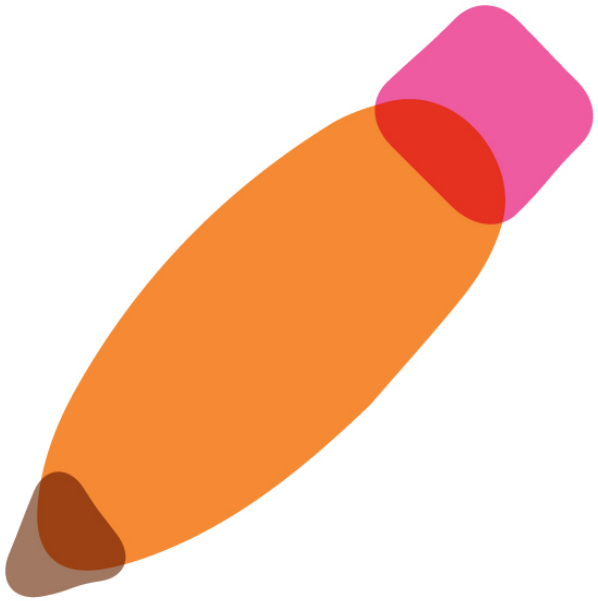
AGENDA

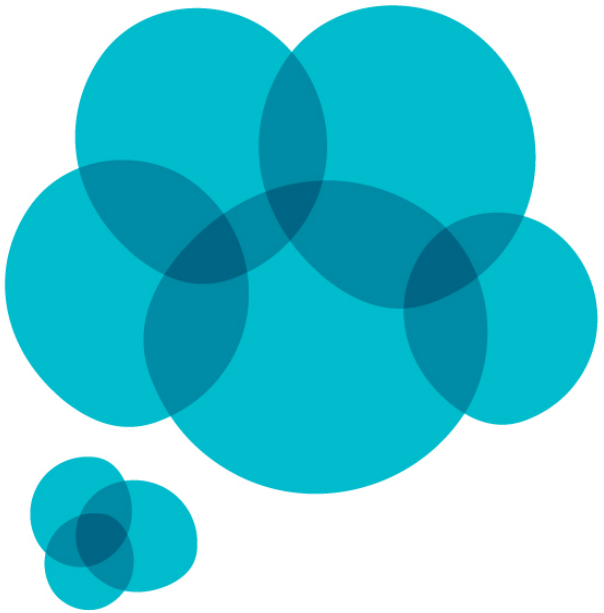




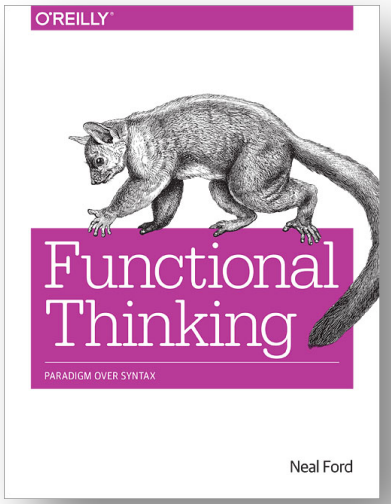

GASOLINE CHAIN SAW
52CC
OZWAY

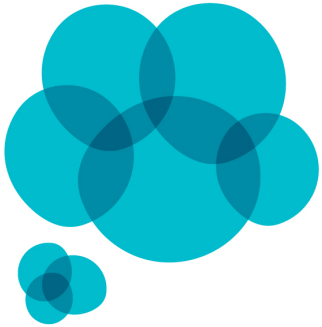
Learning a new language: syntax





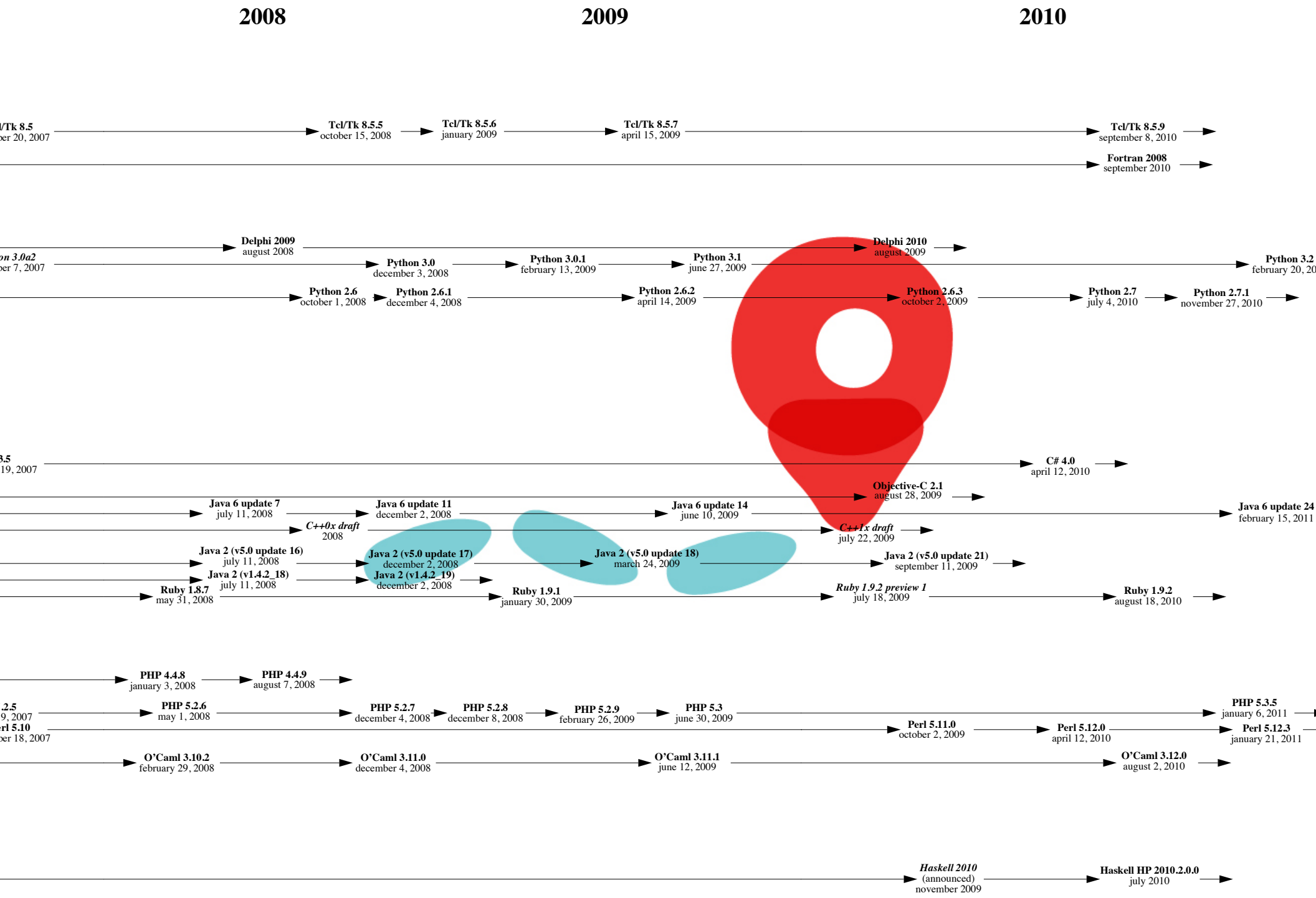
Learning to think differently

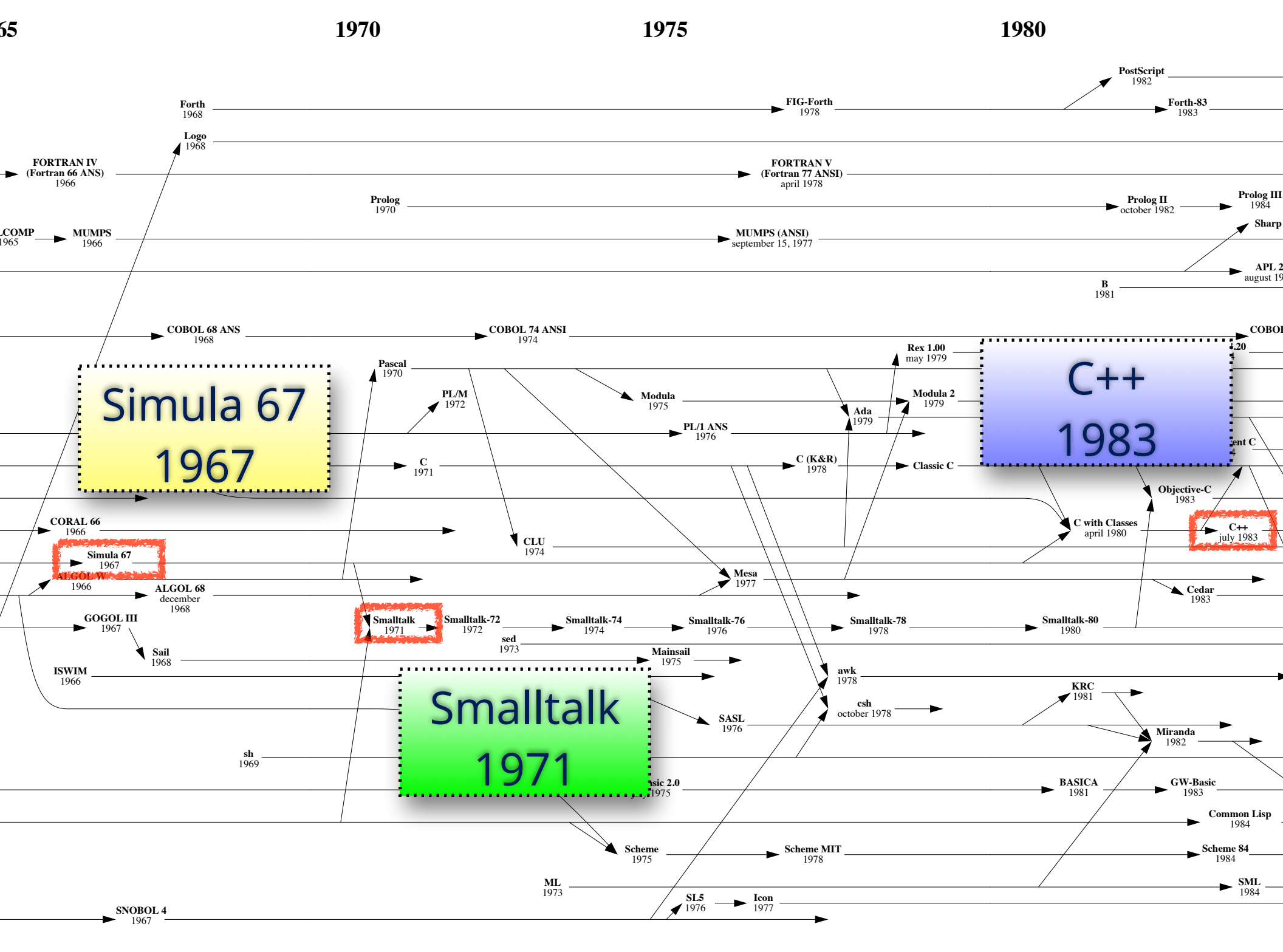


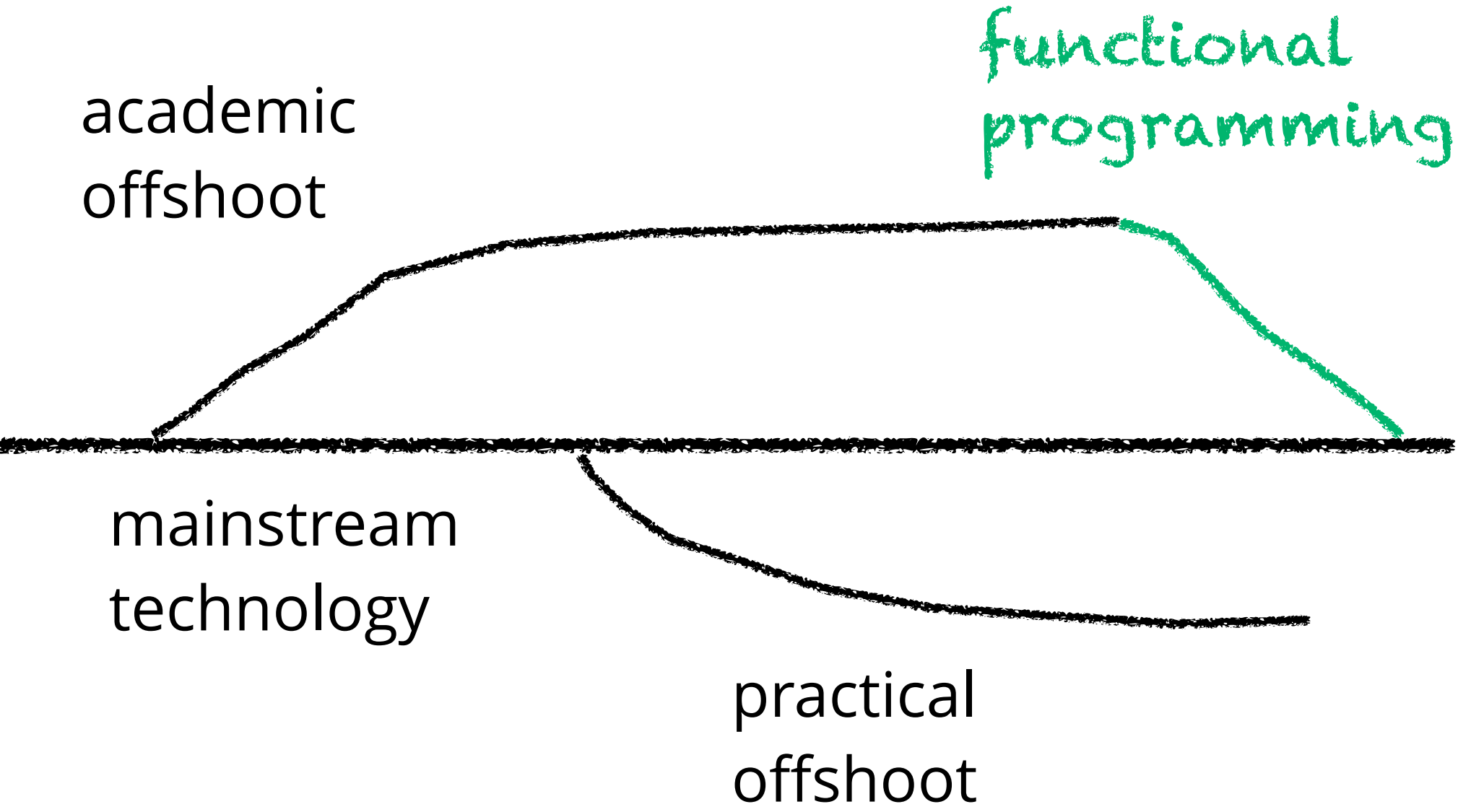


"Functional" is a way of thinking, not a language, framework, or tool

http://oreilly.com/news/languageposter_0504.html

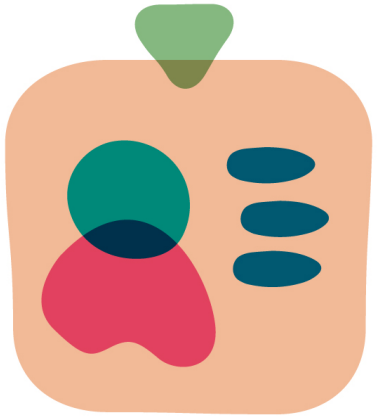



65
1970
1975
1980
Forth
1968
FIG-Forth
1978
PostScript
1982
Forth-83
1983
Logo
1968
FORTRAN IV
(Fortran 66 ANS)
1966
FORTRAN V
(Fortran 77 ANSI)
april 1978
Prolog
1970
Prolog II
october 1982
Prolog III
1984
LCOMP
1965
MUMPS
1966
MUMPS (ANSI)
september 15, 1977
Sharp
APL 2
august 19
B
1981
COBOL 68 ANS
1968
COBOL 74 ANSI
1974
COBOL
Simula 67
1967
Pascal
1970
PL/M
1972
Modula
1975
Rex 1.00
may 1979
Modula 2
1979
Ada
1979
C++
1983
PL/1 ANS
1976
C
1971
C (K&R)
1978
Classic C
ent C
Objective-C
1983
CORAL 66
1966
C with Classes
april 1980
C++
july 1983
CLU
1974
Simula 67
1967
ALGOL W
1966
Mesa
1977
Cedar
1983
ALGOL 68
december
1968
GOGOL III
1967
Smalltalk
1971
Smalltalk-72
1972
Smalltalk-74
1974
Smalltalk-76
1976
Smalltalk-78
1978
Smalltalk-80
1980
sed
1973
Sail
1968
Mainsail
1975
ISWIM
1966
awk
1978
Smalltalk
1971
csh
october 1978
KRC
1981
SASL
1976
Miranda
1982
sh
1969
BASICA
1981
GW-Basic
1983
Common Lisp
1984
Scheme
1975
Scheme MIT
1978
Scheme 84
1984
ML
1973
SML
1984
SL5
1976
Icon
1977
SNOBOL 4
1967


functional
programming
academic
offshoot
mainstream
technology
practical
offshoot

“OO makes code understandable by encapsulating moving parts.
FP makes code understandable by minimizing moving parts.”

Michael Feathers, author of “Working with Legacy Code”

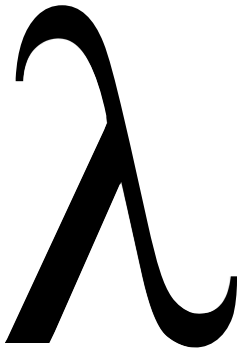


The Company Process

- Given a list of names:
- remove single letter entries
- capitalize first letter
- return a comma-separated list

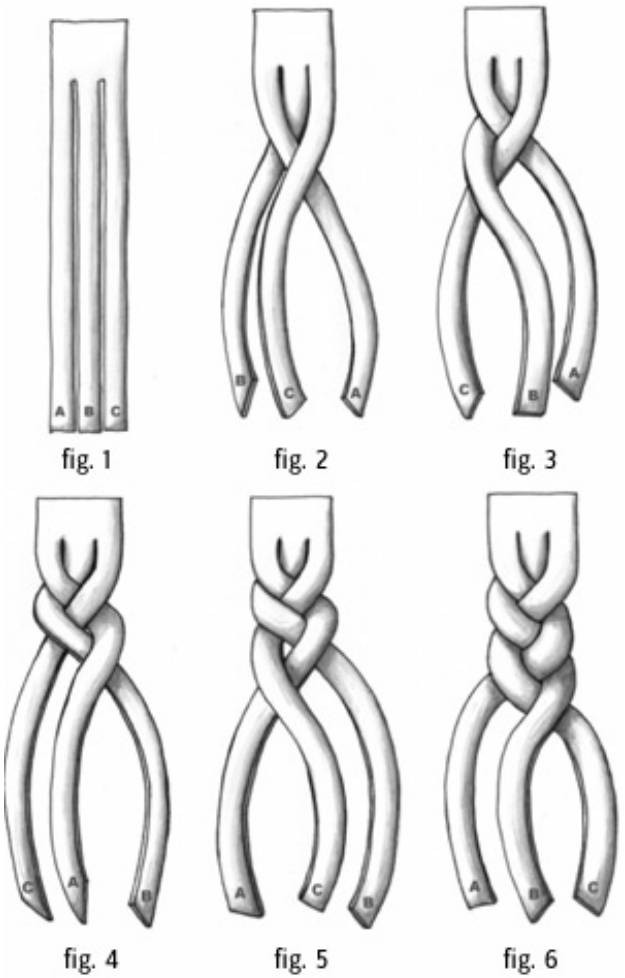
```
public class TheCompanyProcess {
    public String cleanNames(List<String> listOfNames) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < listOfNames.size(); i++) {
            if (listOfNames.get(i).length() > 1) {
                result.append(capitalizeString(listOfNames.get(i))).append(",");
            }
        }
        return result.substring(0, result.length() - 1).toString();
    }

    public String capitalizeString(String s) {
        return s.substring(0, 1).toUpperCase() + s.substring(1, s.length());
    }
}
```

Java 8 company Process



```
public String cleanNames(List<String> names) {
    if (names == null) return "";
    return names
            .stream()
            .filter(name -> name.length() > 1)
            .map(name -> capitalize(name))
            .collect(Collectors.joining(","));
}
public String cleanNamesP(List<String> names) {
    if (names == null) return "";
    return names                                    ring(1, e.length());
            .parallelStream()
            .filter(n -> n.length() > 1)
            .map(e -> capitalize(e))
            .collect(Collectors.joining(","));
}
```

```java
public class TheCompanyProcess {
    public String cleanNames(List<String> listOfNames) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < listOfNames.size(); i++) {
            if (listOfNames.get(i).length() > 1) {
                result.append(capitalizeString(listOfNames.get(i))).append(",");
            }
        }
        return result.substring(0, result.length() - 1).toString();
    }

    public String capitalizeString(String s) {
        return s.substring(0, 1).toUpperCase() + s.substring(1, s.length());
    }
}
```

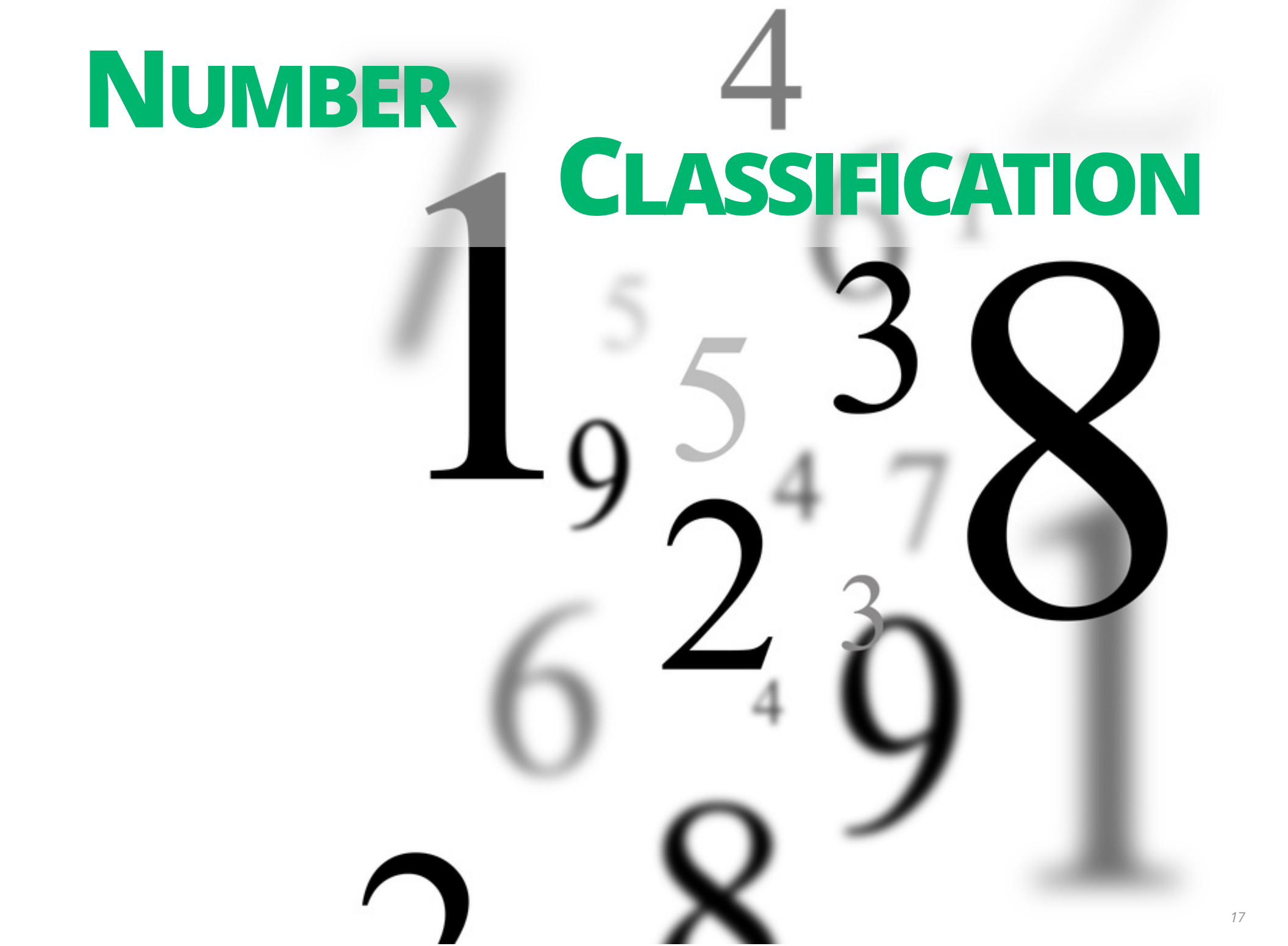


complect, *transitive verb*:
intertwine, embrace, especially
to plait together

```java
public String cleanNames(List<String> names) {
    if (names == null) return "";
    return names
            .stream()
            .filter(name -> name.length() > 1)
            .map(name -> capitalize(name))
            .collect(Collectors.joining(","));
}
```

Functional programming allows you to operate at a higher level of abstraction.

Number Classification



17

Perfect numbers

$$\Sigma(f(\#)) - \# = \#$$

classification

$\sum(f(\#)) = 2\#$ perfect

$\sum(f(\#)) > 2\#$ abundant

$\sum(f(\#)) < 2\#$ deficient

Aliquot Sum

Divisor function – Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/Divisor_function#Definition

Reader

4 matches aliquot Done

Magyar
日本語
Polski
Português
Русский
Svenska
中文
Edit links

The **aliquot sum** s(n) of *n* is the sum of the proper divisors (that is, the divisors excluding *n* itself, A001065), and equals $\sigma_1(n) - n$; the aliquot sequence of *n* is formed by repeatedly applying the aliquot sum function.

Example [edit]

For example, $\sigma_0(12)$ is the number of the divisors of 12:

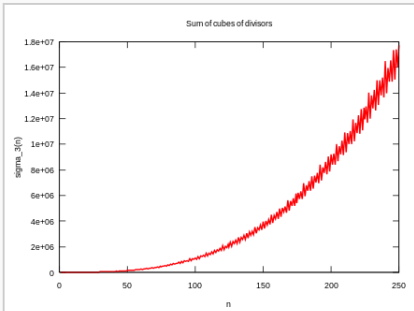
$$\begin{aligned}\sigma_0(12) &= 1^0 + 2^0 + 3^0 + 4^0 + 6^0 + 12^0 \\ &= 1 + 1 + 1 + 1 + 1 + 1 = 6,\end{aligned}$$

while $\sigma_1(12)$ is the sum of all the divisors:

$$\begin{aligned}\sigma_1(12) &= 1^1 + 2^1 + 3^1 + 4^1 + 6^1 + 12^1 \\ &= 1 + 2 + 3 + 4 + 6 + 12 = 28,\end{aligned}$$

and the aliquot sum s(12) of proper divisors is:

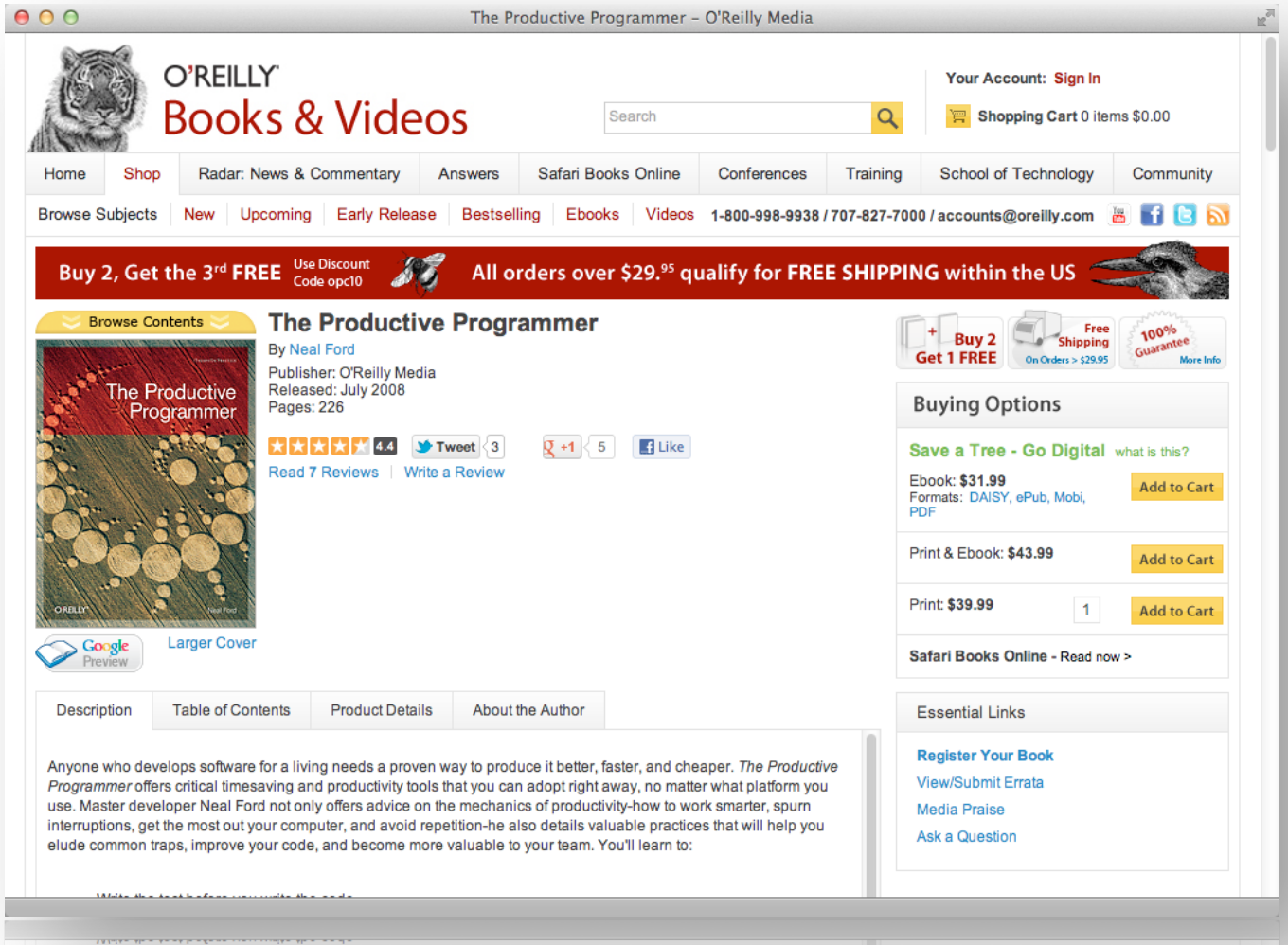
$$\begin{aligned}s(12) &= 1^1 + 2^1 + 3^1 + 4^1 + 6^1 \\ &= 1 + 2 + 3 + 4 + 6 = 16.\end{aligned}$$



Sum of cubes of divisors, $\sigma_3(n)$ up to $n = 250$

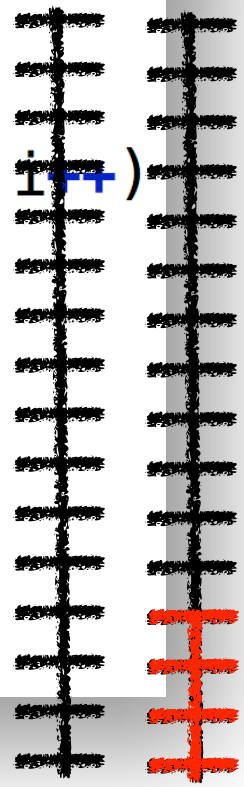
aliquotSum = Σ(f(#)) - #

Imperative Number Classifier



imperative TDD sample:
shop.oreilly.com/product/9780596519544.do

```
public class ImpNumberClassifier {
    private int _number;
    private Map<Integer, Integer> _cache;

    private boolean isFactor(int candidate) {
        return _number % candidate == 0;
    }

    private Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(_number);
        for (int i = 2; i <= sqrt(_number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(_number / i);
            }
        return factors;
    }
```



```
public class ImpNumberClassifier {
    private int _number;
    private Map<Integer, Integer> _cache;

    public ImpNumberClassifier(int targetNumber) {
        _number = targetNumber;
        _cache = new HashMap<>();
    }

    private boolean isFactor(int candidate) {
        return _number % candidate == 0;
    }

    private Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(_number);
        for (int i = 2; i <= sqrt(_number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(_number
            }
        return factors;
    }

    private int aliquotSum() {
        int sum = 0;
        for (int i : getFactors())
            sum += i;
        return sum - _number;
    }

    private int cachedAliquotSum() {
        if (_cache.containsKey(_numb
            return _cache.get(_numbe
        else {
            int sum = aliquotSum();
            _cache.put(_number, sum)
            return sum;
        }
    }

    public boolean isPerfect() {
        return cachedAliquotSum() ==
    }

    public boolean isAbundant() {
        return cachedAliquotSum() >
    }

    public boolean isDeficient() {
        return cachedAliquotSum() <
    }
}
```

```
private int aliquotSum() {
    int sum = 0;
    for (int i : getFactors())
        sum += i;
    return sum - _number;
}
```

```
public boolean isPerfect() {
    return cachedAliquotSum() == _number;
}

public boolean isAbundant() {
    return cachedAliquotSum() > _number;
}

public boolean isDeficient() {
    return cachedAliquotSum() < _number;
}
```

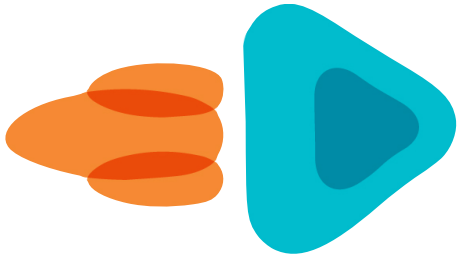
```
public class ImpNumberClassifier {
    private int _number;
    private Map<Integer, Integer> _cache;

    public ImpNumberClassifier(int targetNumber) {
        _number = targetNumber;
        _cache = new HashMap<>();
    }

    private boolean isFactor(int candidate) {
        return _number % candidate == 0;
    }

    private Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(_number);
        for (int i = 2; i <= sqrt(_number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(_number / i);
            }
        return factors;
    }

    private int aliquotSum() {
        int sum = 0;
        for (int i : getFactors())
            sum += i;
        return sum - _number;
    }

    private int cachedAliquotSum() {
        if (_cache.containsKey(_number))
            return _cache.get(_number);
        else {
            int sum = aliquotSum();
            _cache.put(_number, sum);
            return sum;
        }
    }

    public boolean isPerfect() {
        return cachedAliquotSum() == _number;
    }

    public boolean isAbundant() {
        return cachedAliquotSum() > _number;
    }

    public boolean isDeficient() {
        return cachedAliquotSum() < _number;
    }
}
```

- internal state

testable
lazy initialization

- cohesive
- composed
- testable

Slightly more functional version...



```
public class NumberClassifier {

    public static boolean isFactor(final int candidate, final int number) {
        return number % candidate == 0;
    }

    public st
        Set
        fac
        fac
        for

        ret
    }

    public
        int
        int
        for

        }
        ret
    }

    public
        ret
    }

    public
        ret
    }

    public
        ret
    }
}
```

```
public static Set<Integer> factors(final int number) {
```

```
public static int aliquotSum(final Collection<Integer> factors) {
    int sum = 0;
    int targetNumber = Collections.max(factors);
    for (int n : factors) {
        sum += n;
    }
    return sum - targetNumber;
}

public static boolean isPerfect(final int number) {
    return aliquotSum(factors(number)) == number;
}

public static boolean isAbundant(final int number) {
    return aliquotSum(factors(number)) > number;
}

public static boolean isDeficient(final int number) {
    return aliquotSum(factors(number)) < number;
}
```

inefficient for multiple classifications

```
public class NumberClassifier {

    public static boolean isFactor(final int candidate, final int number) {
        return number % candidate == 0;
    }

    public static Set<Integer> factors(final int number) {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < number; i++)
            if (isFactor(i, number))
                factors.add(i);
        return factors;
    }

    public static int aliquotSum(final Collection<Integer> factors) {
        int sum = 0;
        int targetNumber = Collections.max(factors);
        for (int n : factors) {
            sum += n;
        }
        return sum - targetNumber;
    }

    public static boolean isPerfect(final int number) {
        return aliquotSum(factors(number)) == number;
    }

    public static boolean isAbundant(final int number) {
        return aliquotSum(factors(number)) > number;
    }

    public static boolean isDeficient(final int number) {
        return aliquotSum(factors(number)) < number;
    }
}
```

public static

public static

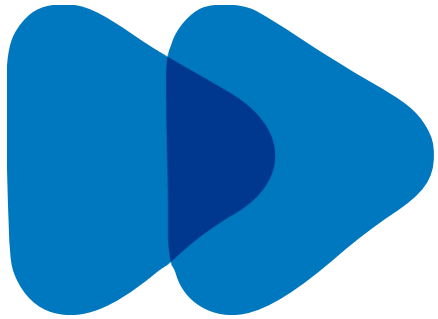
public static

public static

public static

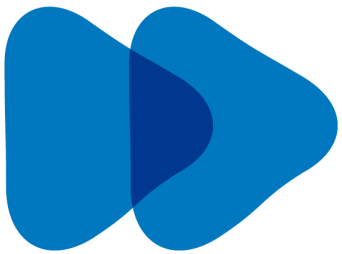
- ☐ No internal state
- ☐ less need for scoping
- ☐ testable
- ☐ refactorable
- ☐ lazy initialization?

functional version in

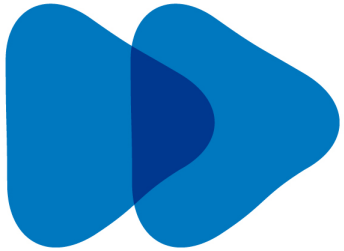


```
public class NumberClassifier {

    public static IntStream factorsOf(int number) {
        return range(1, number + 1)
                .filter(potential -> number % potential == 0);
    }

    public static int aliquotSum(int number) {
        return factorsOf(number).sum() - number;
    }

    public static boolean isPerfect(int number) {
        return aliquotSum(number) == number;
    }

    public static boolean isAbundant(int number) {
        return aliquotSum(number)> number;
    }

    public static boolean isDeficient(int number) {
        return aliquotSum(number) < number;
    }

}
```

Functional programming allows you to operate at a higher level of abstraction.

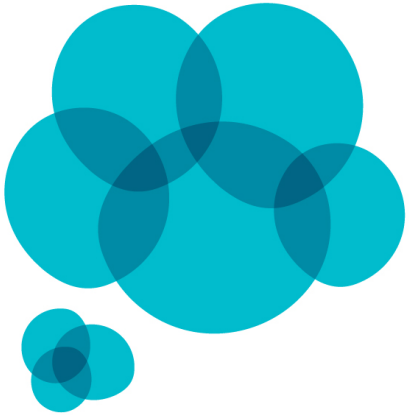


Shift

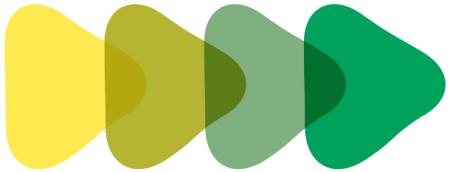


Shift

Multiparadigm

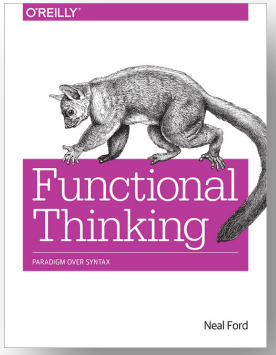


Smarter
(not Harder)



Advance

AGENDA



Shift

Building Blocks

filter

Performs a boolean test on each element and only returns those that pass.

```
public static IntStream factorsOf(int number) {
    return range(1, number + 1)
            .filter(potential -> number % potential == 0);
}
```


```
def factors(number: Int) =
  (1 to number) filter (isFactor(_, number))
```


```
(defn factors [number]
  (filter #(is-factor? % number) (range 1 (inc number))))
```

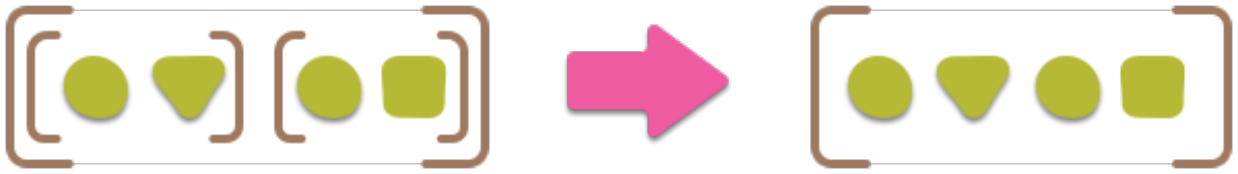
map

Applies given function to each element and returns a transformed collection.

```
public static List fastFactorsOf(int number) {
    List<Integer> factors = range(1, (int) (sqrt(number) + 1))
            .filter(potential -> number % potential == 0)
            .boxed()
            .collect(toList());
    List factorsAboveSqrt = factors
            .stream()
            .map(e -> number / e).collect(toList());
    factors.addAll(factorsAboveSqrt);
    return factors.stream().distinct().collect(toList());
}
```


```
static def factors(number) {
  def factors = (1..round(sqrt(number)+1)).findAll({number % it == 0})
  (factors + factors.collect {number / it}).unique()
}
```

flatten



Removes nesting from a collection.

```
(flatten [[1 2 [3 4] 5] 6 [7 8]])
; => (1 2 3 4 5 6 7 8)
```

```
[ [1, 2, 3], [4, 5, 6], [7, 8, 9] ].flatten()
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
(words.collect {it.toList()}).flatten()
// [t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x, j, ...
```

flat-map

Map a function over a collection and flatten the result by one-level.

```
List(List(1, 2, 3), List(4, 5, 6),
     List(7, 8, 9)) flatMap (_.toList)
// List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```


```
words flatMap (_.toList)
// List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x, ...
```

```
(mapcat #(clojure.string/split % #"\s+")
        ["two birds" "three green peas"])
;; => ("two" "birds" "three" "green" "peas")
```

reduce

Uses the supplied function to combine the input elements, often to a single output value.

```
(defn aliquot-sum [number]
  (- (reduce + (factors number)) number))
```

```
static def sumFactors(number) {
  factors(number).inject(0, {i, j -> i + j})
}
```

fold-left/fold-right

Uses the supplied function to combine input elements, often to a single return value.

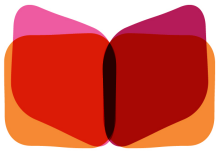
```
def sum(factors : Seq[Int]) =
  factors.foldLeft(0)(_ + _)
```

Scala

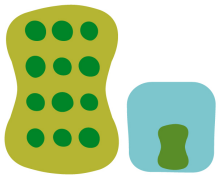
fold-right

```
List.range(1, 10) reduceRight(_ - _)

// 8 -     9  = -1
// 7 - (-1) = 8
// 6 -     8  = -2
// 5 - (-2) = 7
// 4 -     7  = -3
// 3 - (-3) = 6
// 2 -     6 = -4
// 1 - (-4) = 5
// result: 5
```

5 !



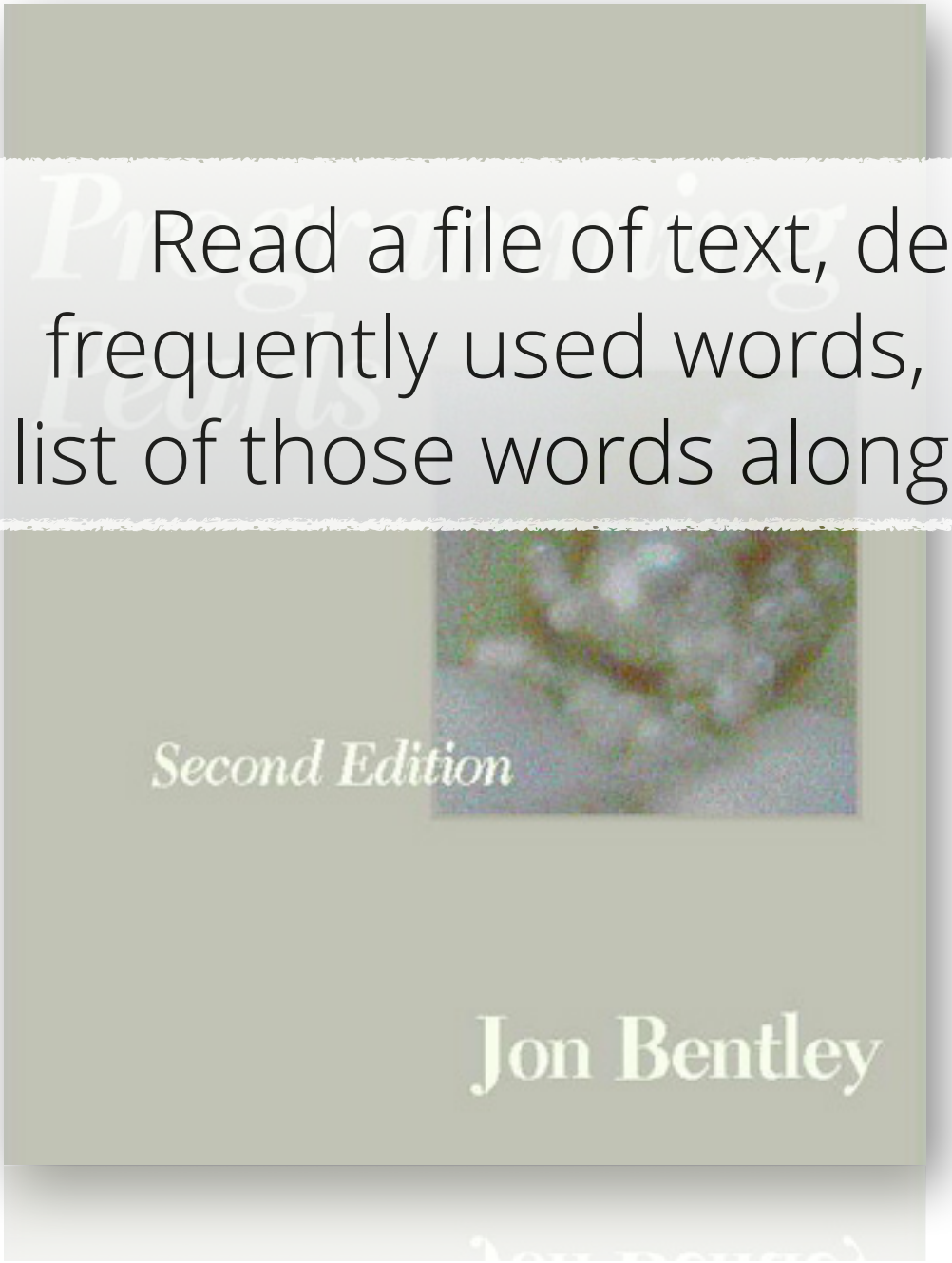
versus



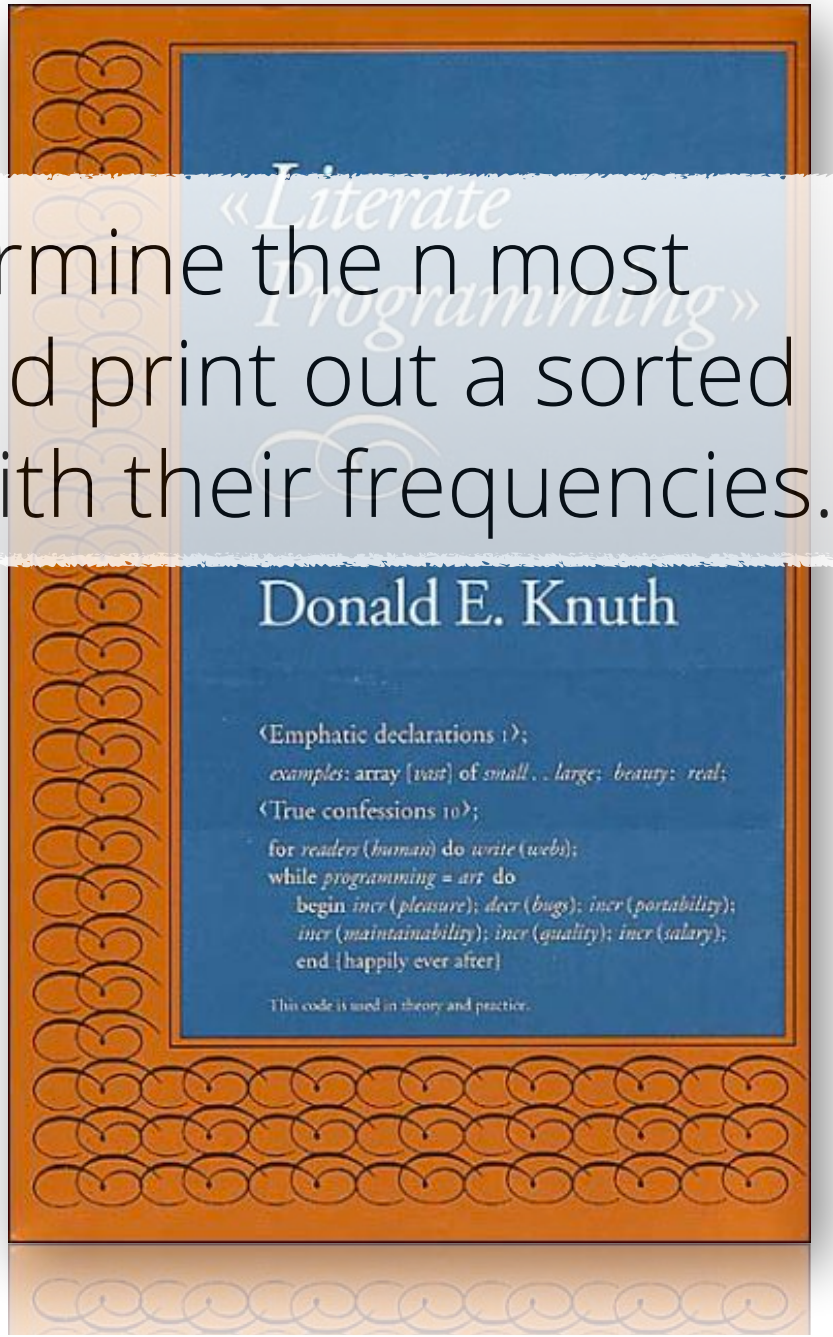
- both use an accumulator
- **`reduce`**: you supply seed value for accumulator
- **`fold`**: starts with nothing* in the accumulator
- **`foldLeft/reduce`**:
  - Use a binary function or operator to combine the first element of the list with the initial value of an accumulator.
  - Repeat step 1 until the list is exhausted and the accumulator holds the result of the fold.

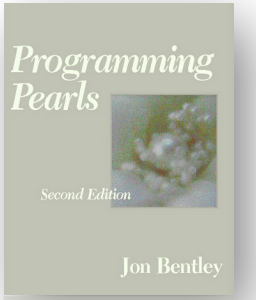
* for some appropriate value of “nothing”

An Old Challenge



Read a file of text, determine the n most frequently used words, and print out a sorted list of those words along with their frequencies.






```
public class Words {
    private Set<String> NON_WORDS = new HashSet<String>() {{
        add("the"); add("and"); add("of"); add("to"); add("a");
        add("i"); add("it"); add("in"); add("or"); add("is");
        add("d"); add("s"); add("as"); add("so"); add("but");
        add("be");
        }};

    public Map wordFreq(String words) {
        TreeMap<String, Integer> wordMap = new TreeMap<String, Integer>();
        Matcher m = Pattern.compile("\\w+").matcher(words);
        while (m.find()) {
            String word = m.group().toLowerCase();
            if (! NON_WORDS.contains(word)) {
                if (wordMap.get(word) == null) {
                    wordMap.put(word, 1);
                }
                else {
                    wordMap.put(word, wordMap.get(word) + 1);
                }
            }
        }
        return wordMap;
    }
}
```

```
public class Words {
    private Set<String> NON_WORDS = new HashSet<String>() {{
        add("the"); add("and"); add("of"); add("to"); add("a");
        add("i"); add("it"); add("in"); add("or"); add("is");
        add("d"); add("s"); add("as"); add("so"); add("but");
        add("be");
        }};

    public Map wordFreq(String words) {
        TreeMap<String, Integer> wordMap = new TreeMap<String, Integer>();
        Matcher m = Pattern.compile("\\w+").matcher(words);
        while (m.find()) {
            String word = m.group().toLowerCase();
            if (! NON_WORDS.contains(word)) {
                if (wordMap.get(word) == null) {
                    wordMap.put(word, 1);
                }
                else {
                    wordMap.put(word, wordMap.get(word) + 1);
                }
            }
        }
        return wordMap;
    }
}
```

```
private List<String> regexToList(String words, String regex) {
    List wordList = new ArrayList<>();
    Matcher m = Pattern.compile(regex).matcher(words);
    while (m.find())
    wordList.add(m.group());
    return wordList;
}
```

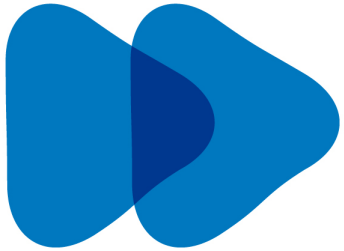
```
public Map wordFreq(String words) {
    TreeMap<String, Integer> wordMap = new TreeMap<>();
    regexToList(words, "\\w+").stream()
            .map(w -> w.toLowerCase())
            .filter(w -> !NON_WORDS.contains(w))
            .forEach(w -> wordMap.put(w, wordMap.getOrDefault(w, 0) + 1));
    return wordMap;
}
```

Use **`filter`** to produce a subset of a collection based on supplied filtering criteria.

Use **`map`** to transform collections in situ.

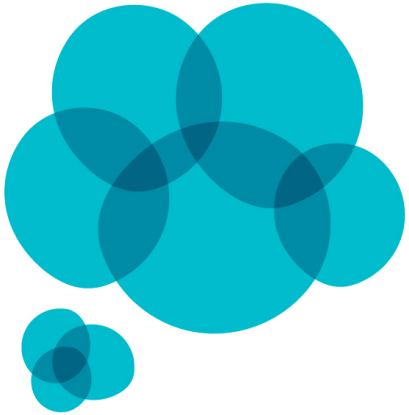
Use **`reduce`** or **`fold`** for piecewise collection processing.

Shift

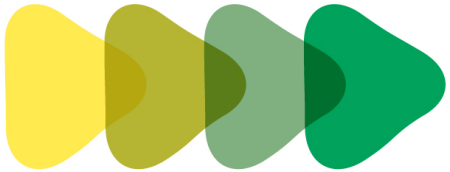


Shift

Multiparadigm

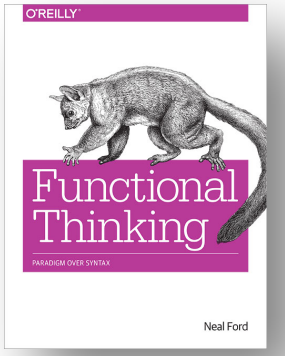


Smarter
(not Harder)



Advance

AGENDA



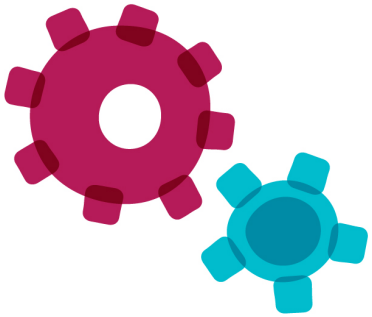
Higher-order / First-class Functions

functions that can either take other functions as arguments or return them as results

closure

Simple Closure Binding

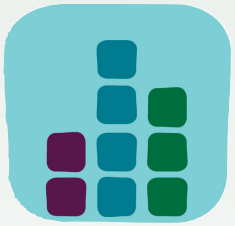

```
class Employee {
  def name, salary
}

def paidMore(amount) {
  return {Employee e -> e.salary > amount}
}

isHighPaid = paidMore(100000)

def Smithers = new Employee(name:"Fred", salary:120000)
def Homer = new Employee(name:"Homer", salary:80000)
println isHighPaid(Smithers)
println isHighPaid(Homer)
// true, false

isHigherPaid = paidMore(200000)
println isHigherPaid(Smithers)
println isHigherPaid(Homer)
def Burns = new Employee(name:"Monty", salary:1000000)
println isHigherPaid(Burns)
// false, false, true
```



Let the language manage state.

```
def Closure makeCounter() {
  def local_variable = 0
  return { return local_variable += 1 }
}

c1 = makeCounter()
c1()
c1()
c1()

c2 = makeCounter()

println "C1 = ${c1()}, C2 = ${c2()}"
// output: C1 = 4, C2 = 1
```

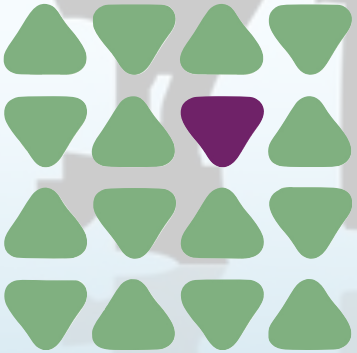
Let the language manage ~~state~~ what it will!



memory management

garbage collection

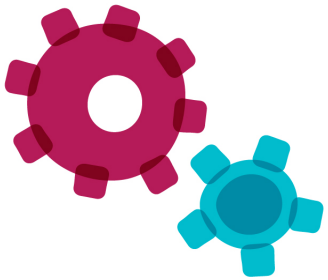
state



concurrency

“Imperative languages use state to model programming, exemplified by parameter passing and hidden context (**`this`**).”

“Closures allow us to model behavior by encapsulating both code and context into a single construct, the closure, that can be passed around like traditional data structures and executed at exactly the correct time and place.”



Currying & Partial Application

- Currying describes the conversion of a multi-argument function into a chain of single-argument functions.
  - describes the transformation process, not the invocation of the converted function.
  - caller can decide how many arguments to apply, thereby creating a derived function with that smaller number of arguments.
- Partial application describes the conversion of a multi-argument function into one that accepts fewer arguments, with values for the elided arguments supplied in advance.

Currying

```
def product = { x, y -> x * y }

def quadrate = product.curry(4)
def octate = product.curry(8)

println "4x4: ${quadrate.call(4)}"
println "8x5: ${octate(5)}"
```


```
def volume = {h, w, l -> h * w * l}
def area = volume.curry(1)
def lengthPA = volume.curry(1, 1)
def lengthC = volume.curry(1).curry(1)
```

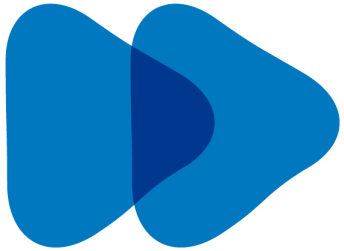
partial application

currying

Currying Usages

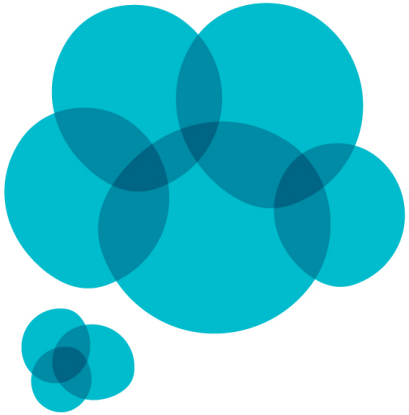
```
def adder = { x, y -> x + y}
def incrementer = adder.curry(1)
```


```
def composite = { f, g, x -> return f(g(x)) }
def thirtyTwoer = composite.curry(quadrate, octate)
```

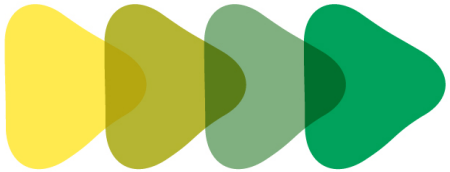


Shift

Multiparadigm

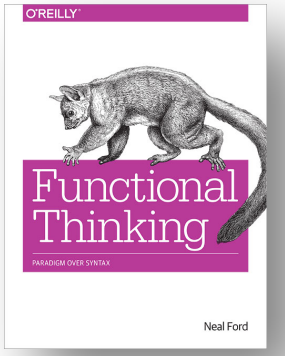


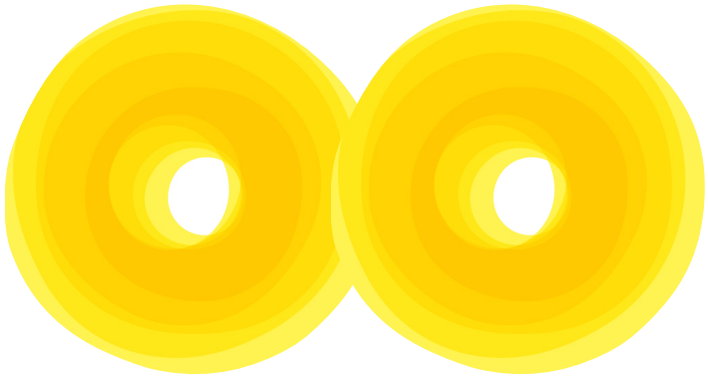
Smarter
(not Harder)



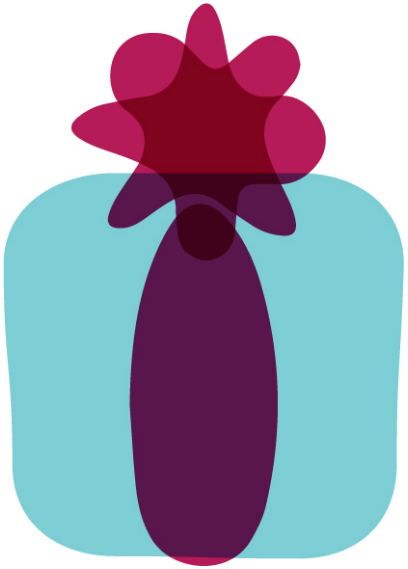
Advance

AGENDA

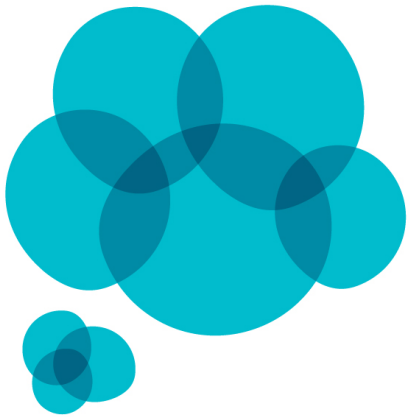




laziness



memoization

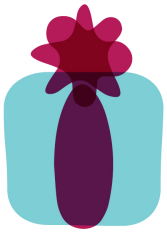


Smarter
(not Harder)

Caching Sum in Classifier

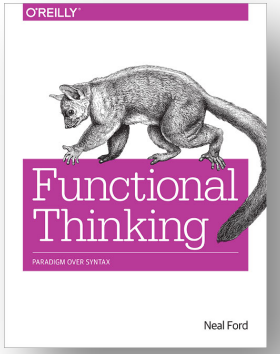
```
class ClassifierCachedSum {
  private sumCache = [:]

  def sumOfFactors(number) {
    if (! sumCache.containsKey(number)) {
      sumCache[number] = factorsOf(number).sum()
    }
   return sumCache[number]

  }
// remainder of code unchanged...
```

```
    def static aliquotSumOfFactors = { number ->
      factorsOf(number).sum() - number
    }
    def static aliquotSum = aliquotSumOfFactors.memoize()
```



```
class ClassifierMemoized {
  def static dividesBy = { number, potential ->
    number % potential == 0
  }
  def static isFactor = dividesBy.memoize()

  def static factorsOf(number) {
    (1..number).findAll { i -> isFactor.call(number, i) }
  }

  def static aliquotSumOfFactors = { number ->
    factorsOf(number).sum() - number
  }
  def static aliquotSum = aliquotSumOfFactors.memoize()

  def static isPerfect(number) {
    aliquotSum(number) == number
  }

  def static isAbundant(number) {
    aliquotSum(number) > number
  }

  def static isDeficient(number) {
    aliquotSum(number) < number
  }
}
```

Memoization Results



Version	Results (smaller is better)
Nonoptimized	577 ms
Nonoptimized (2nd)	280 ms
Cached sum	600 ms
Cached sum (2nd)	50 ms
Cached	411 ms
Cached (2nd)	38 ms
Partially memoized	228 ms
Partially memoized (2nd)	60 ms
Memoized	956 ms
Memoized (2nd)	19 ms

“Language designers will always build more efficient mechanisms because they are allowed to bend rules.”

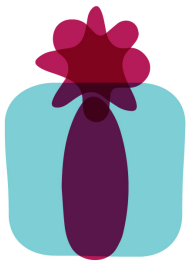
Memoization in 

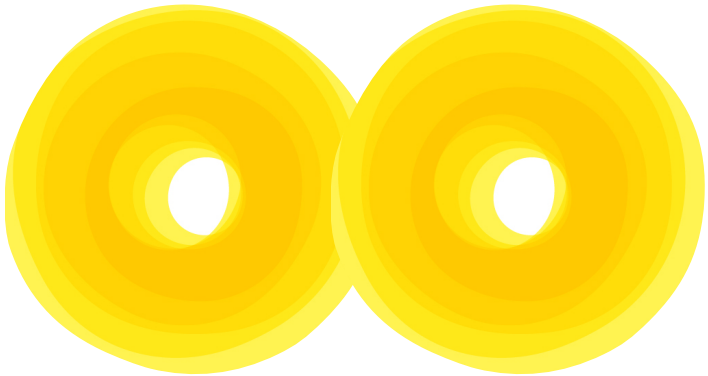
```
(use '[clojure.string :only (join split)])


(let [alpha (into #{} (concat (map char (range (int \a) (inc (int \z))))
                              (map char (range (int \A) (inc (int \Z))))))
      rot13-map (zipmap alpha (take 52 (drop 26 (cycle alpha))))]

  (defn rot13
    "Given an input string, produce the rot 13 version of
     the string. \"hello\" -> \"uryyb\""
    [s]
    (apply str (map #(get rot13-map % %) s))))

(defn name-hash [name]
  (apply str (map #(rot13 %) (split name #"\d"))))

(def name-hash-m (memoize name-hash))
```

Make sure all memoized functions:

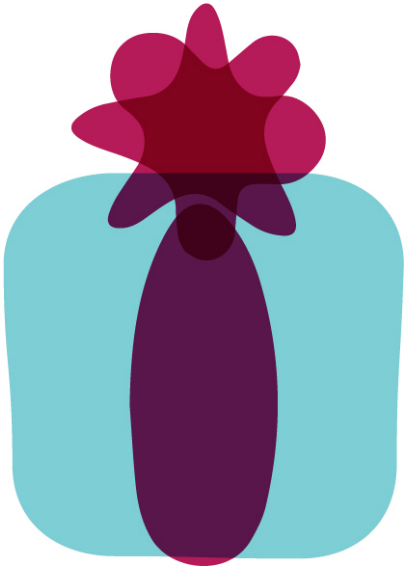
- Have no side effects

- Never rely on outside information

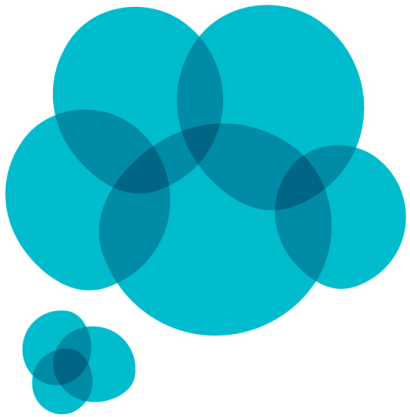




laziness

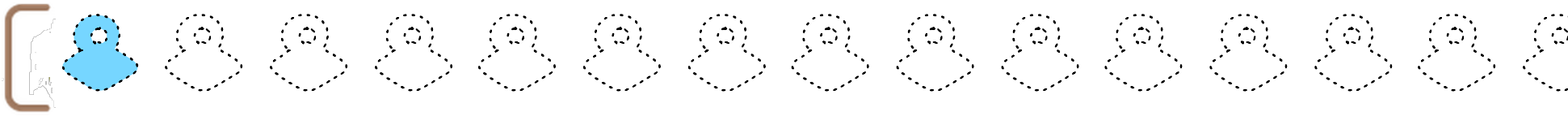


memoization



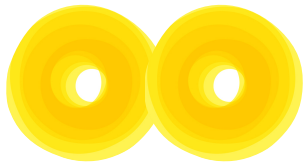
Smarter
(not Harder)

laziness



Lazy data structures defer expression evaluation for as long as possible.

- defer expensive calculations until absolutely needed
- create infinite collections
- functions like map and filter can be made more efficient when lazily evaluated



Laziness in Java (pre-8)

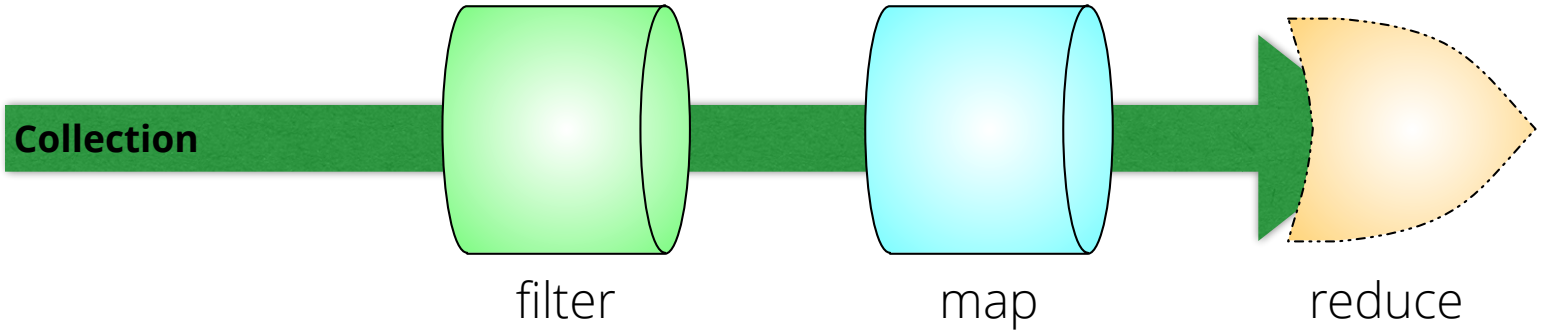
```
import com.googlecode.totallylazy.Predicate;
import com.googlecode.totallylazy.Sequence;

import static com.googlecode.totallylazy.Predicates.is;
import static com.googlecode.totallylazy.numbers.Numbers.*;
import static com.googlecode.totallylazy.predicates.WherePredicate.where;

public class NumberClassifier {
   public static Predicate<Number> isFactor(Number n) {
        return where(remainder(n), is(zero));
    }

    public static Sequence<Number> getFactors(final Number n) {
        return range(1, n).filter(isFactor(n));
    }

    public static Sequence<Number> factors(final Number n) {
        return getFactors(n).memorise();
    }

    public static Number aliquotSum(Number n) {
        return subtract(factors(n).reduce(sum), n);
    }

    public static boolean isPerfect(Number n) {
        return equalTo(n, aliquotSum(n));
    }

    public static boolean isAbundant(Number n) {
        return greaterThan(aliquotSum(n), n);
    }

    public static boolean isDeficient(Number n) {
        return lessThan(aliquotSum(n), n);
    }
}
```

f(x) totallylazy

Another functional library for Java

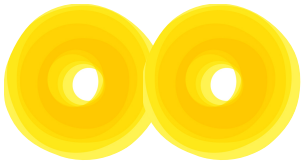
https://code.google.com/p/totallylazy/

Streams



λ functional rather than stateful 

consumed upon use 

 unbound/infinite

as lazy as possible 

Benefits of Laziness

```
def isPalindrome(s) {
  def sl = s.toLowerCase()
  sl == sl.reverse()
}

def findFirstPalindrome(s) {
  s.tokenize(' ').find {isPalindrome(it)}
}

s1 = "The quick brown fox jumped over anna the dog";
println(findFirstPalindrome(s1))
s2 = "Bob went to Harrah and gambled with Otto and Steve"
println(findFirstPalindrome(s2))
```


```
(defn palindrome? [s]
  (let [sl (.toLowerCase s)]
  (= sl (apply str (reverse sl)))))

(defn find-palindromes [s]
  (filter palindrome? (clojure.string/split s #" ")))

(println (find-palindromes "The quick brown fox jumped over anna the dog"))
; (anna)
(println (find-palindromes "Bob went to Harrah and gambled with Otto and Steve"))
;(Bob Harrah Otto)
(println (take 1 (find-palindromes "Bob went to Harrah with Otto and Steve")))
;(Bob)
```

Laziness

- Seq, an abstraction on traditional Lisp lists, forms the core of laziness
- collection functions produce and consume Seqs
- concrete and lazy Seqs interoperate

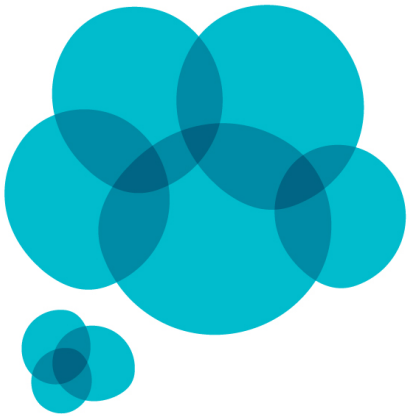
```
(for [x (range 2) y (range 3)] [x y])
=> ([0 0] [0 1] [0 2] [1 0] [1 1] [1 2])


(take 20 (for [x (range 100000000) y (range 1000000)
               :while (< y x)]
           [x y]))
=> ([1 0] [2 0] [2 1] [3 0] [3 1] [3 2]
    [4 0] [4 1] [4 2] [4 3] [5 0] [5 1]
    [5 2] [5 3] [5 4] [6 0] [6 1] [6 2]
    [6 3] [6 4])
```

Scala Laziness

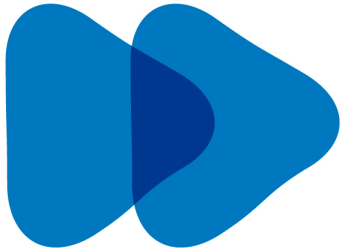
```
def isPalindrome(x: String) = x == x.reverse
def findPalidrome(s: Seq[String]) = s find isPalindrome
```

```
findPalindrome(words take 1000000)
```

```
findPalindrome(words.view take 1000000)
```

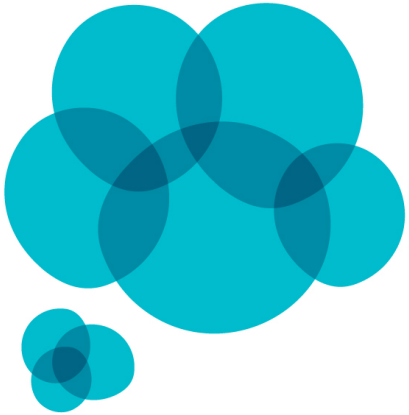


Smarter
(not Harder)

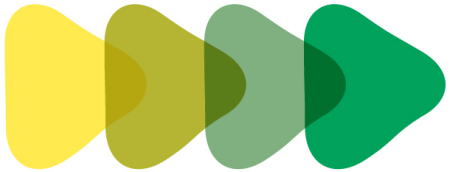


Shift

Multiparadigm

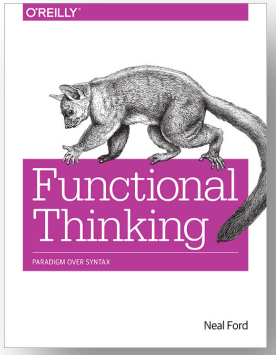


Smarter
(not Harder)

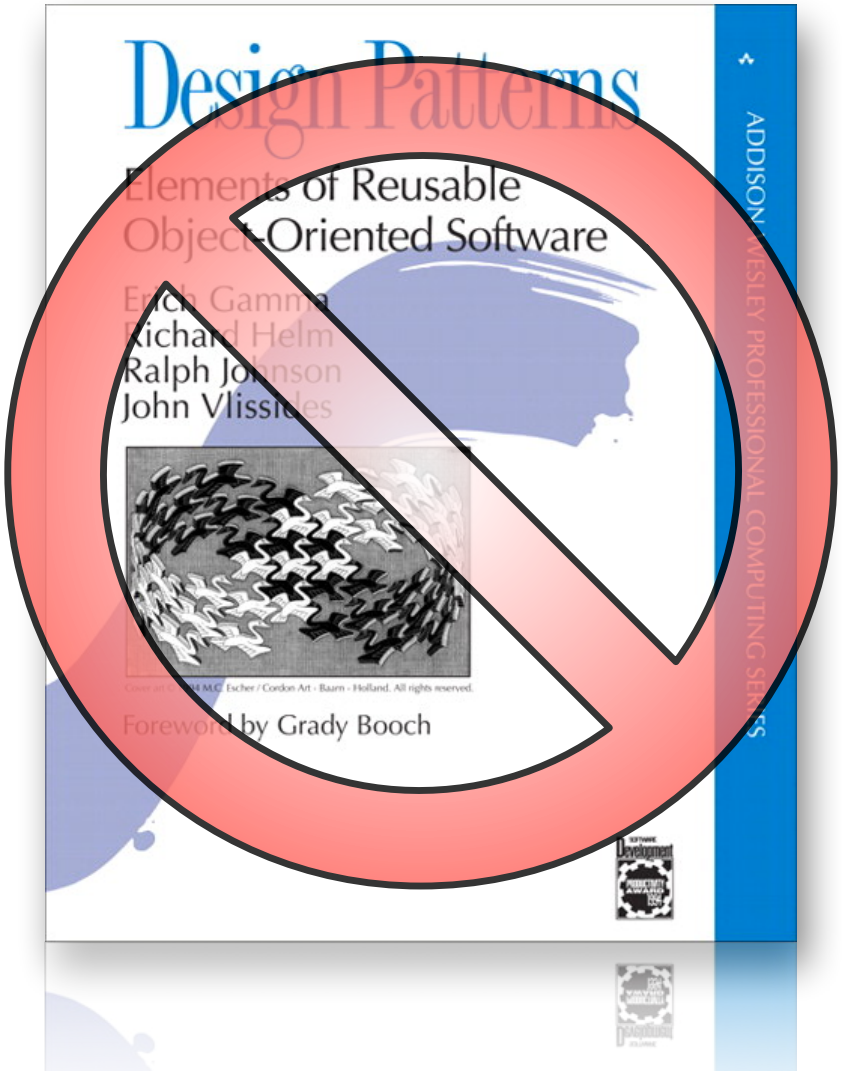


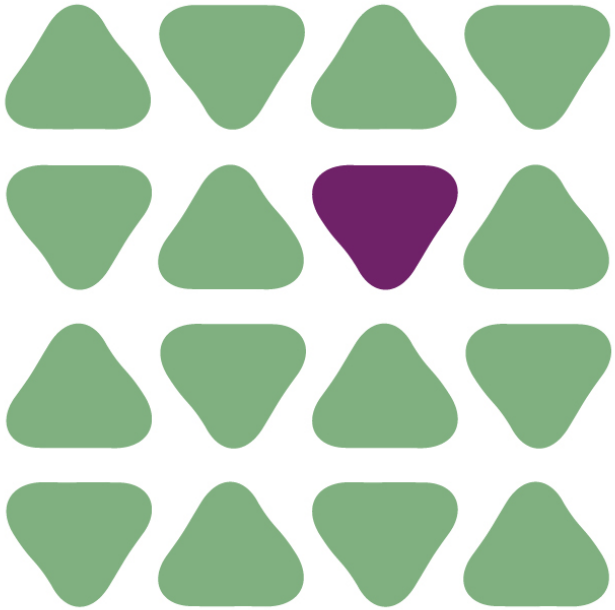
Advance

AGENDA



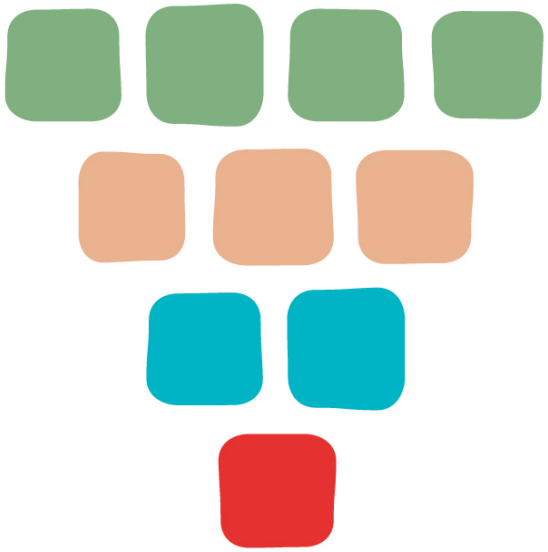
pattern: a named, cataloged solution to a common problem





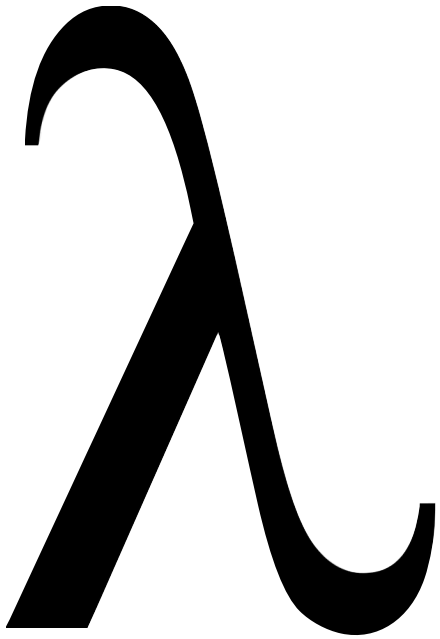
Design Patterns in Functional Languages

- The pattern is absorbed by the language.
  - Command design pattern
- The pattern solution still exists in the functional paradigm, but the implementation details differ.
  - Flyweight design pattern
- The solution is implemented using capabilities other languages or paradigms lack
  - Laziness



Reuse:

Structural versus Functional



Number Classifier

```
public class ClassifierAlpha {
    private int number;

    public ClassifierAlpha(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    static public int sum(Set<Integer> factors) {
        Iterator it = factors.iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

    public boolean isPerfect() {
        return sum(factors()) - number == number;
    }

    public boolean isAbundant() {
        return sum(factors()) - number > number;
    }

    public boolean isDeficient() {
        return sum(factors()) - number < number;
    }
}
```

Prime Numbers

```
public class PrimeAlpha {
    private int number;

    public PrimeAlpha(int number) {
        this.number = number;
    }

    public boolean isPrime() {
        Set<Integer> primeSet = new HashSet<Integer>() {{
            add(1); add(number);}};
        return number > 1 &&
                factors().equals(primeSet);
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

}
```

```java
public class ClassifierAlpha {
    private int number;

    public ClassifierAlpha(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    static public int sum(Set<Integer> factors) {
        Iterator it = factors.iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

    public boolean isPerfect() {
        return sum(factors()) - number == number;
    }

    public boolean isAbundant() {
        return sum(factors()) - number > number;
    }

    public boolean isDeficient() {
        return sum(factors()) - number < number;
    }
}
```

==

```java
public class PrimeAlpha {
    private int number;

    public PrimeAlpha(int number) {
        this.number = number;
    }

    public boolean isPrime() {
        Set<Integer> primeSet = new HashSet<Integer>() {{
            add(1); add(number);}};
        return number > 1 &&
                factors().equals(primeSet);
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

}
```

Refactor to Eliminate Duplication

Factors

```
public class FactorsBeta {
    protected int number;

    public FactorsBeta(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> getFactors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

Refactored Children

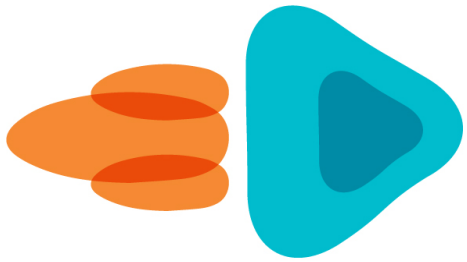
```java
public class ClassifierBeta extends FactorsBeta {

    public ClassifierBeta(int number) {
        super(number);
    }

    public int sum() {
        Iterator it = getFactors().iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

    public boolean isPerfect() {
        return sum() - number == number;
    }

    public boolean isAbundant() {
        return sum() - number > number;
    }

    public boolean isDeficient() {
        return sum() - number < number;
    }
}
```

```java
public class FactorsBeta {
    protected int number;

    public FactorsBeta(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> getFactors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

```java
public class PrimeBeta extends FactorsBeta {
    public PrimeBeta(int number) {
        super(number);
    }

    public boolean isPrime() {
        Set<Integer> primeSet = new HashSet<Integer>() {{
            add(1); add(number);}};
        return getFactors().equals(primeSet);
    }

}
```

coupling

```java
public class NumberClassifier {

    public static boolean isFactor(final int candidate, final int number) {
        return number % candidate == 0;
    }

    public static Set<Integer> factors(final int number) {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < number; i++)
            if (isFactor(i, number))
                factors.add(i);
        return factors;
    }

    public static int aliquotSum(final Collection<Integer> factors) {
        int sum = 0;
        int targetNumber = Collections.max(factors);
        for (int n : factors) {
            sum += n;
        }
        return sum - targetNumber;
    }

    public static boolean isPerfect(final int number) {
        return aliquotSum(factors(number)) == number;
    }

    public static boolean isAbundant(final int number) {
        return aliquotSum(factors(number)) > number;
    }

    public static boolean isDeficient(final int number) {
        return aliquotSum(factors(number)) < number;
    }
}
```

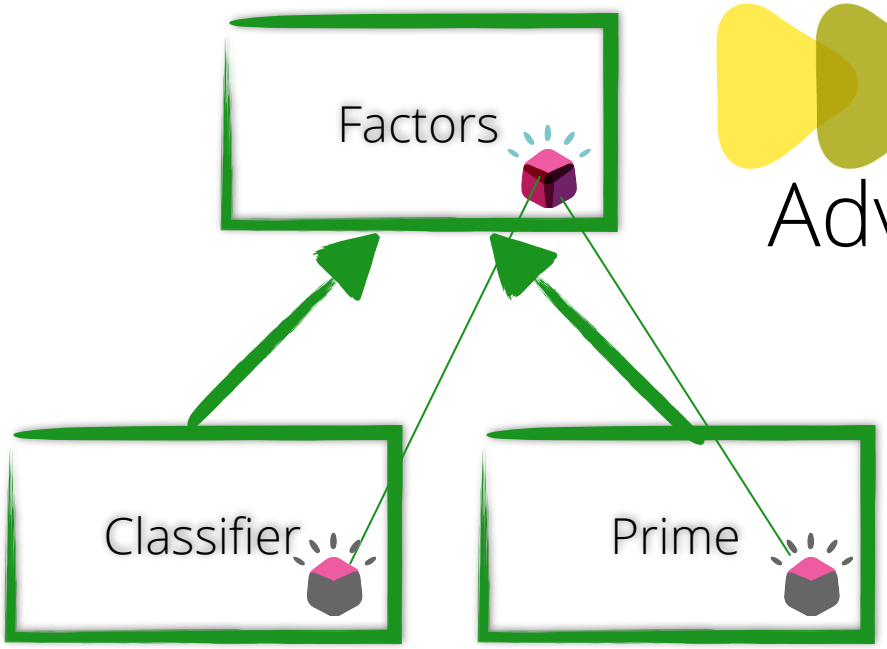


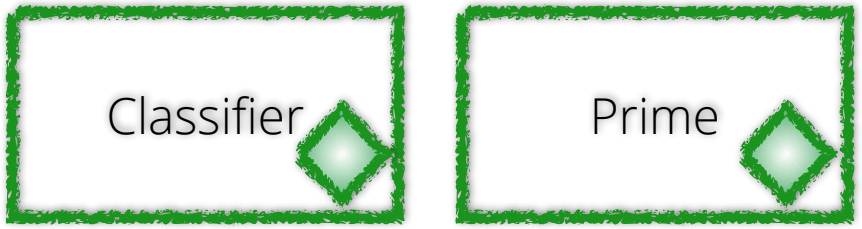
```
public class Factors {
    static public boolean isFactor(int number, int potential_factor) {
        return number % potential_factor == 0;
    }

    static public Set<Integer> of(int number) {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(number, i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

}
```

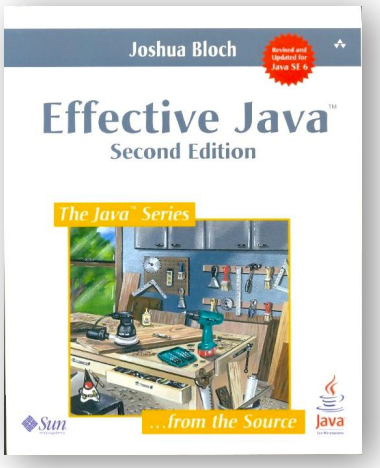
```
public class FPrime {

    public static boolean isPrime(int number) {
        Set<Integer> factors = Factors.of(number);
        return number > 1 &&
                factors.size() == 2 &&
                factors.contains(1) &&
                factors.contains(number);
    }
}
```

```
public class FClassifier {

    public static int sumOfFactors(int number) {
        Iterator<Integer> it = Factors.of(number).iterator();
        int sum = 0;
        while (it.hasNext())
            sum += it.next();
        return sum;
    }

    public static boolean isPerfect(int number) {
        return sumOfFactors(number) - number == number;
    }

    public static boolean isAbundant(int number) {
        return sumOfFactors(number) - number > number;
    }

    public static boolean isDeficient(int number) {
        return sumOfFactors(number) - number < number;
    }
}
```

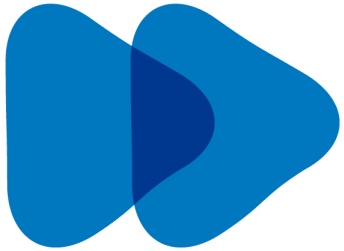
Coupling versus Composition





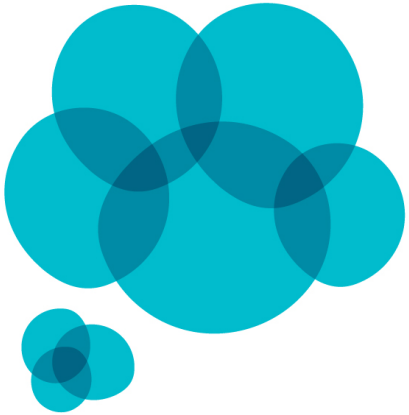


> ...favor composition over inheritance.

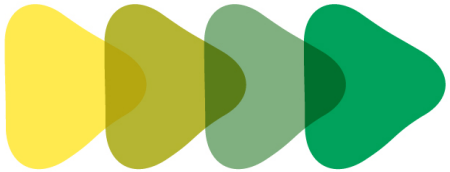


Shift

Multiparadigm

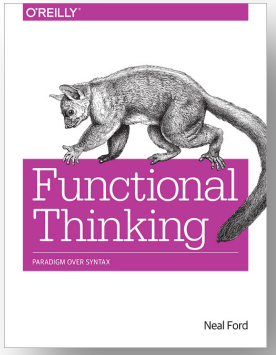


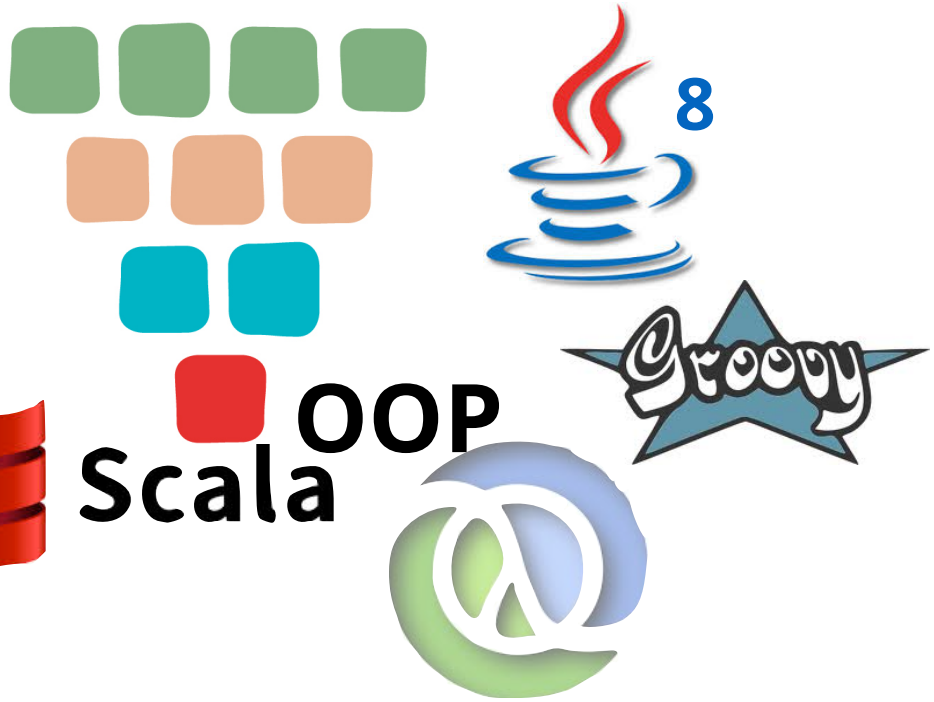
Smarter
(not Harder)



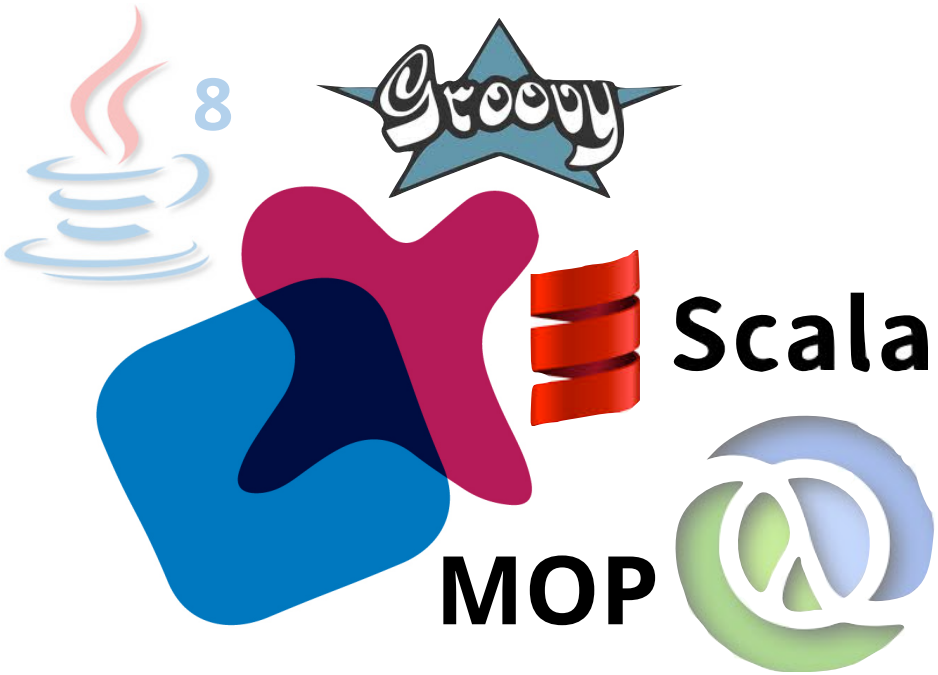
Advance

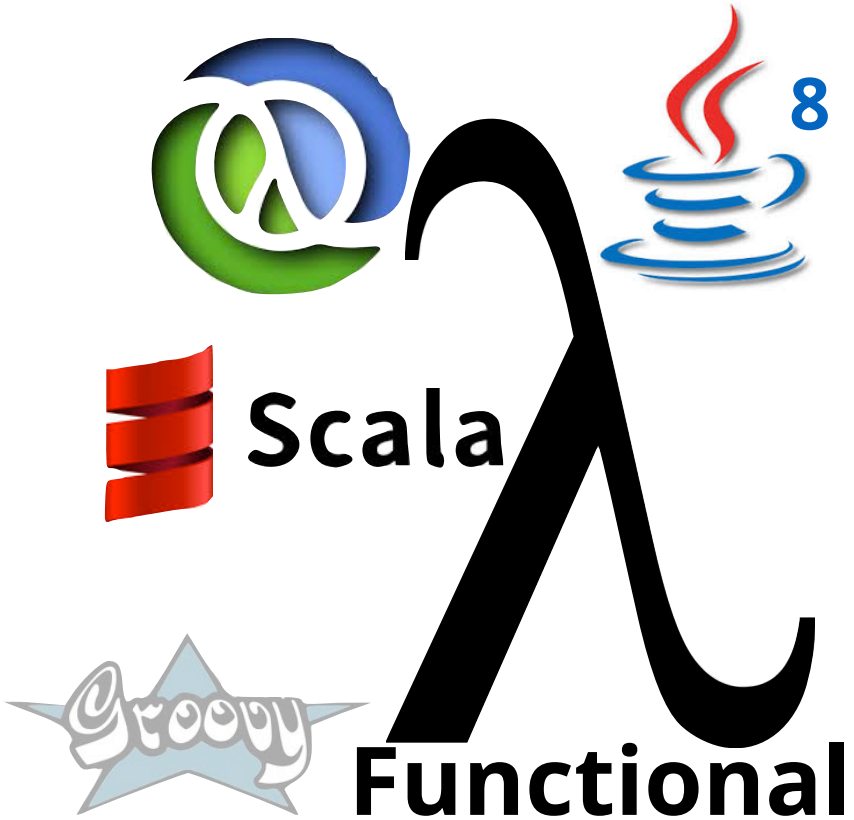
AGENDA



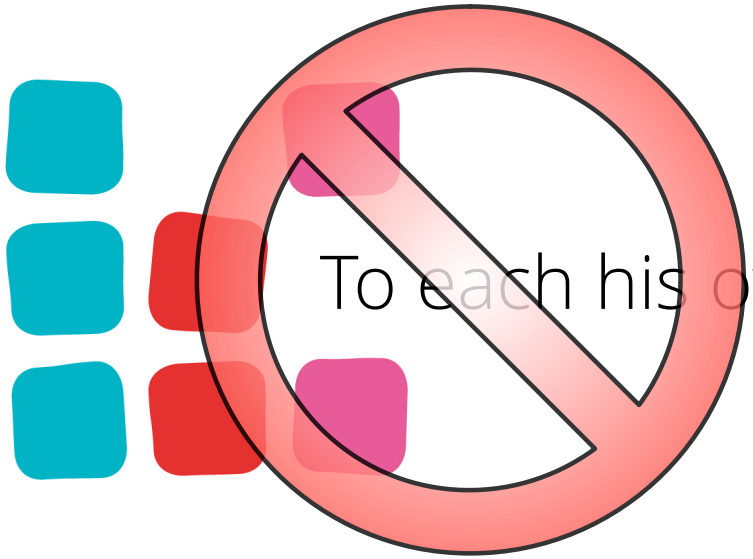


Multiparadigm





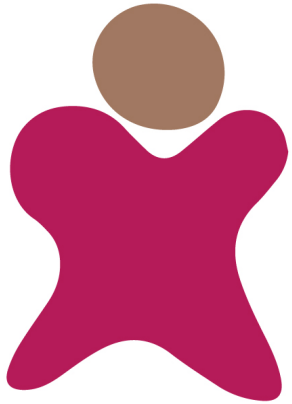
Consequences of Multi-paradigm Languages



To each his own paradigm!

Paradigmatic discipline

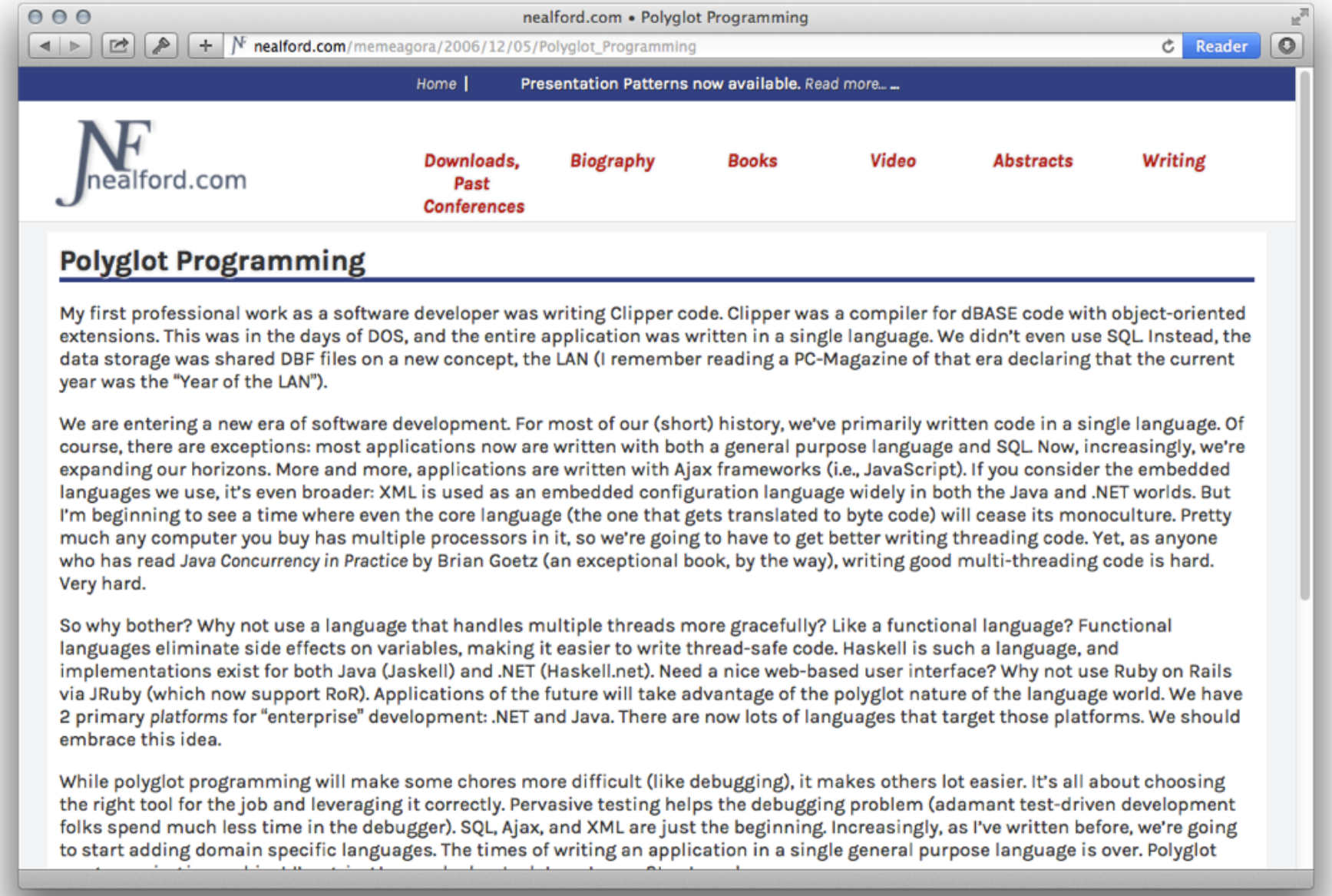
- code reviews
- pair programming

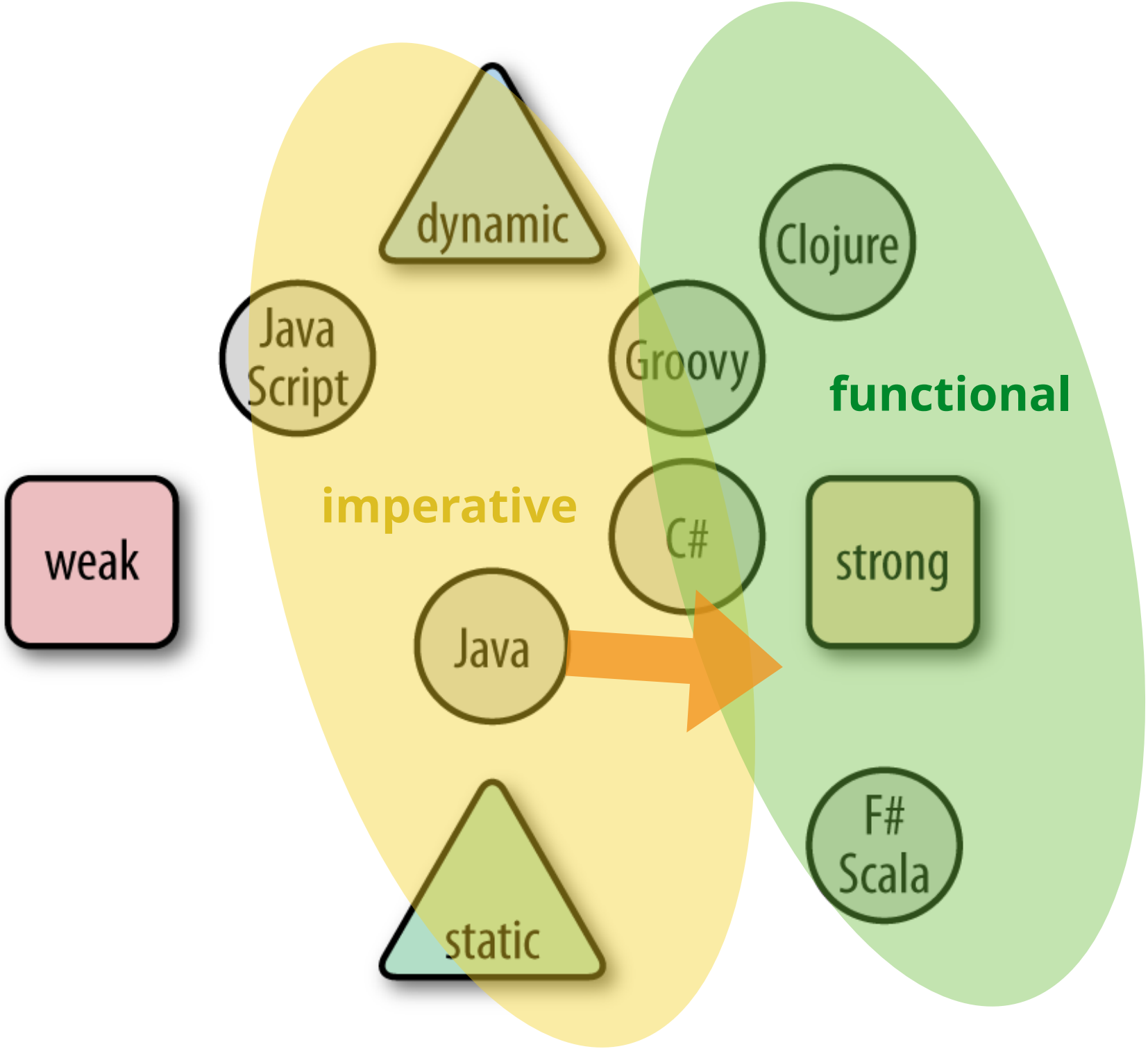


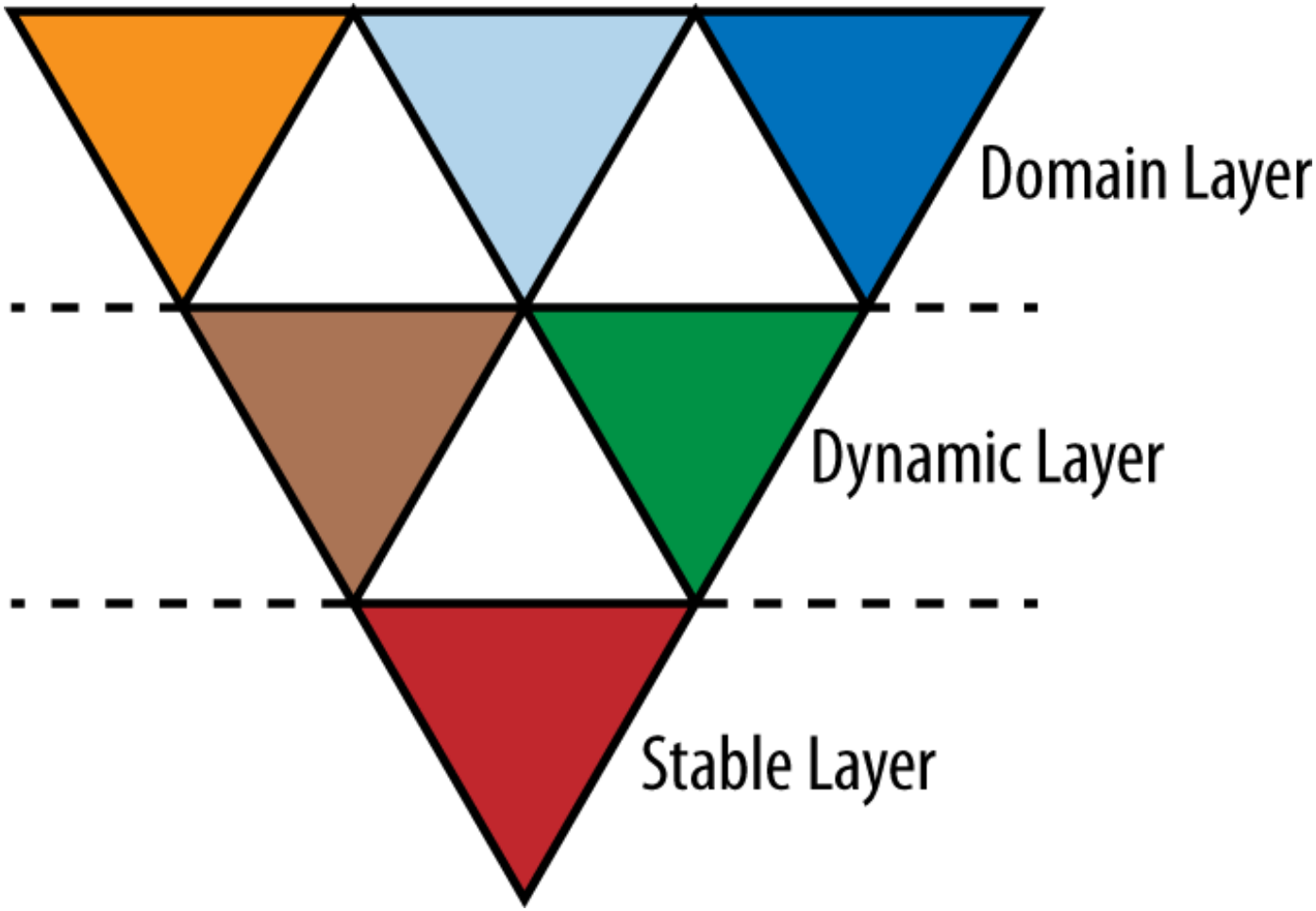
consumer-driven contracts

Polyglot Programming

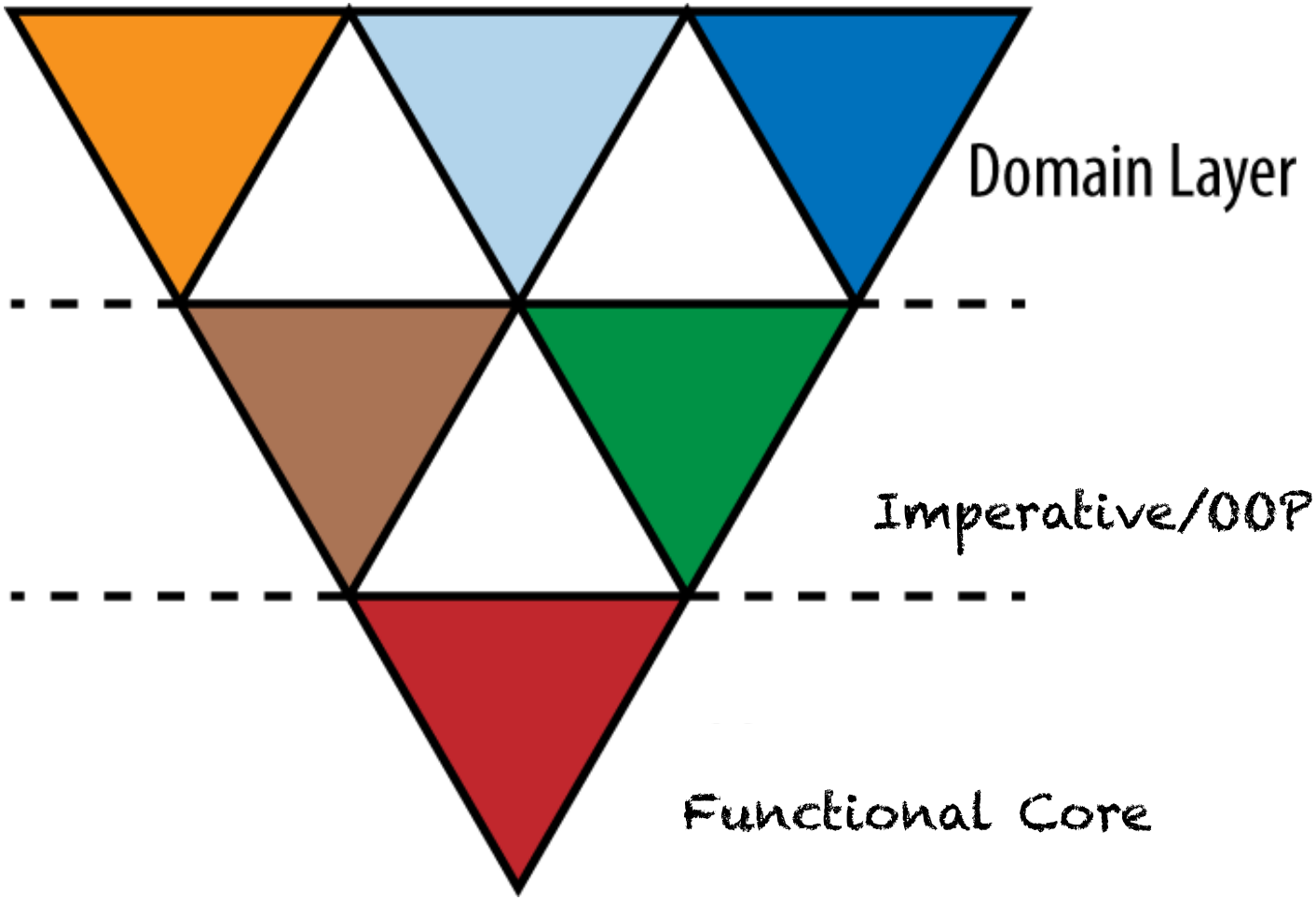
nealford.com/memeagora/2006/12/05/Polyglot_Programming

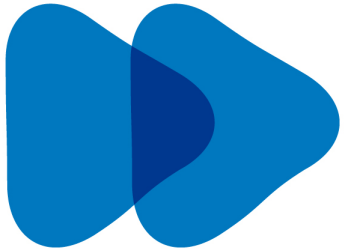



dynamic
Clojure
Java Script
Groovy
functional
imperative
weak
C#
strong
Java
F# Scala
static


Domain Layer
Dynamic Layer
Stable Layer

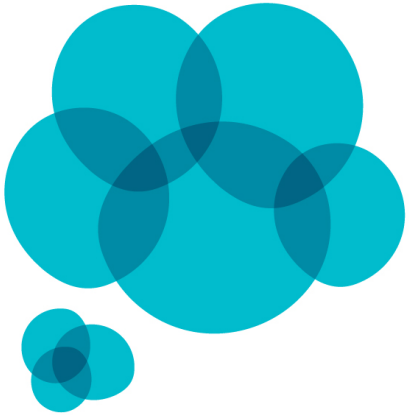
My Updated Pyramid



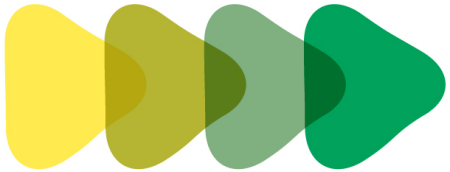


Shift

Multiparadigm

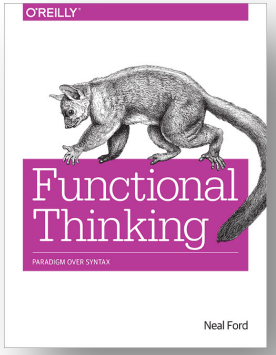


Smarter
(not Harder)



Advance

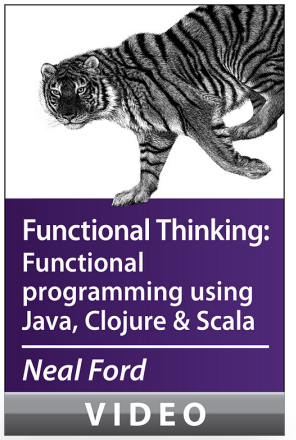
SUMMARY



ThoughtWorks®

NEAL FORD

Director / Software Architect / Meme Wrangler



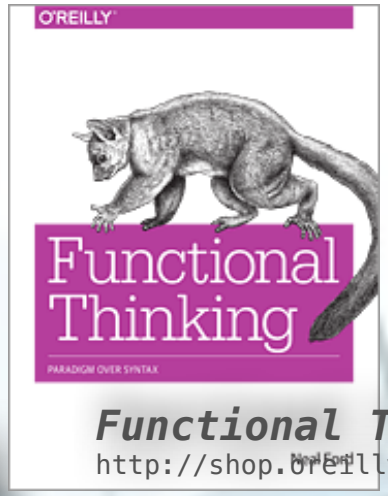
bit.ly/nf_ftvideo

@neal4d
nealford.com

Functional Thinking

Clojure (inside out)

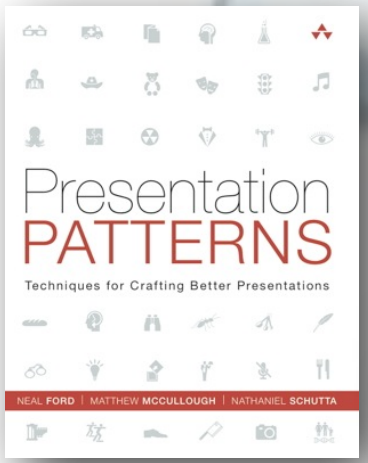
Stuart Halloway, Neal Ford
bit.ly/clojureinsideout



Functional Thinking
http://shop.oreilly.com/product/0636920029687.do

Understanding the Basics

Neal Ford, Mark Richards
http://oreil.ly/1kM7IuV



Presentation Patterns

Neal Ford, Matthew McCullough, Nathaniel Schutta
http://presentationpatterns.com

Taking a Deeper Dive

Neal Ford, Mark Richards
http://oreil.ly/RfKUqh