ThoughtWorks®

Functional Thinking

NEAL FORD director / software architect
meme wrangler

ThoughtWorks®

nford@thoughtworks.com
2002 Summit Boulevard, Atlanta, GA 30319
nealford.com
thoughtworks.com
memeagora.blogspot.com
@neal4d

NF

a metaphor
an essay
a history lesson

new language:
easy

new paradigm:
hard

"functional" is more a way of thinking than a tool set

Execution in the Kingdom of Nouns

Steve Yegge

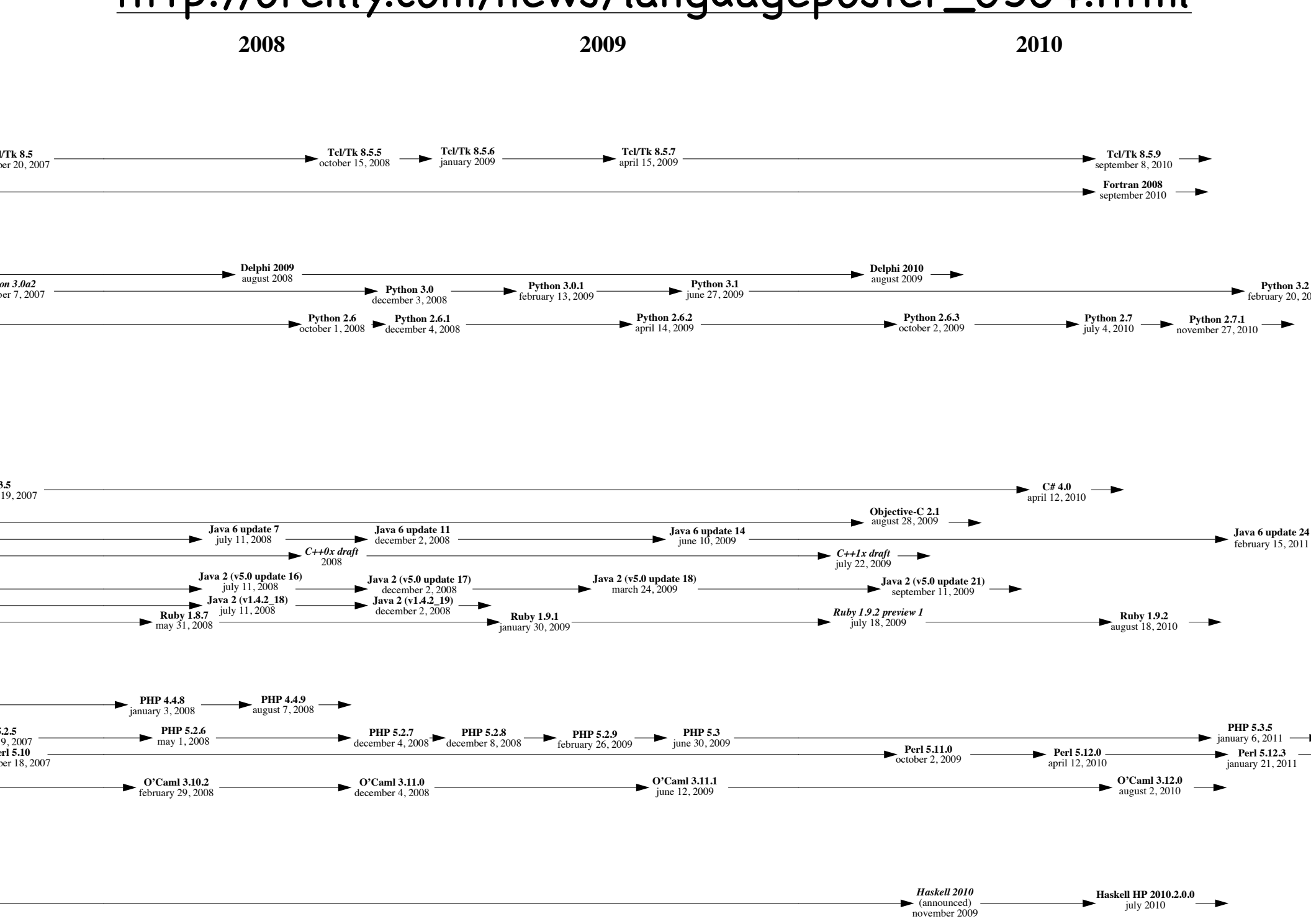
http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

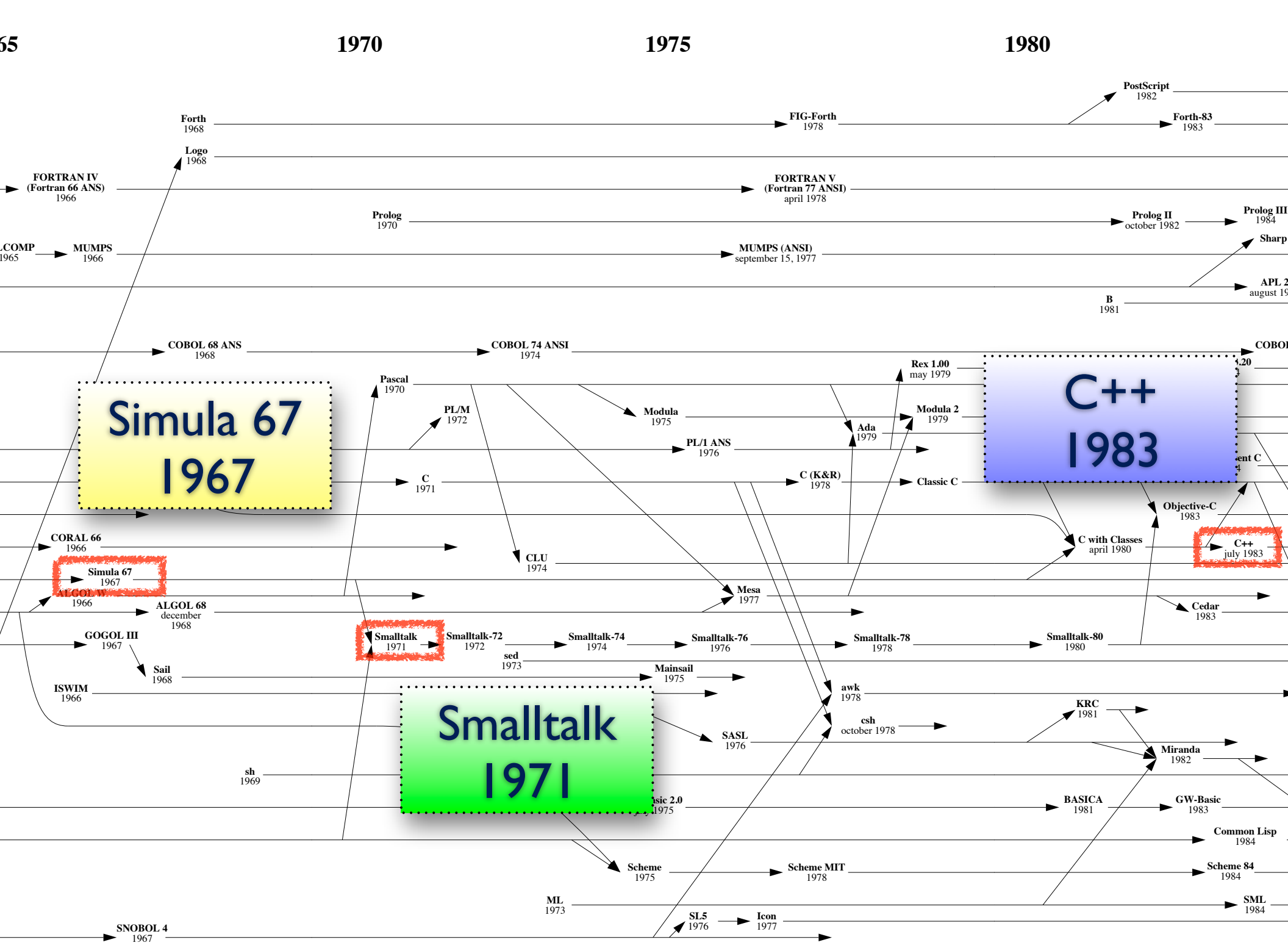

verbs!

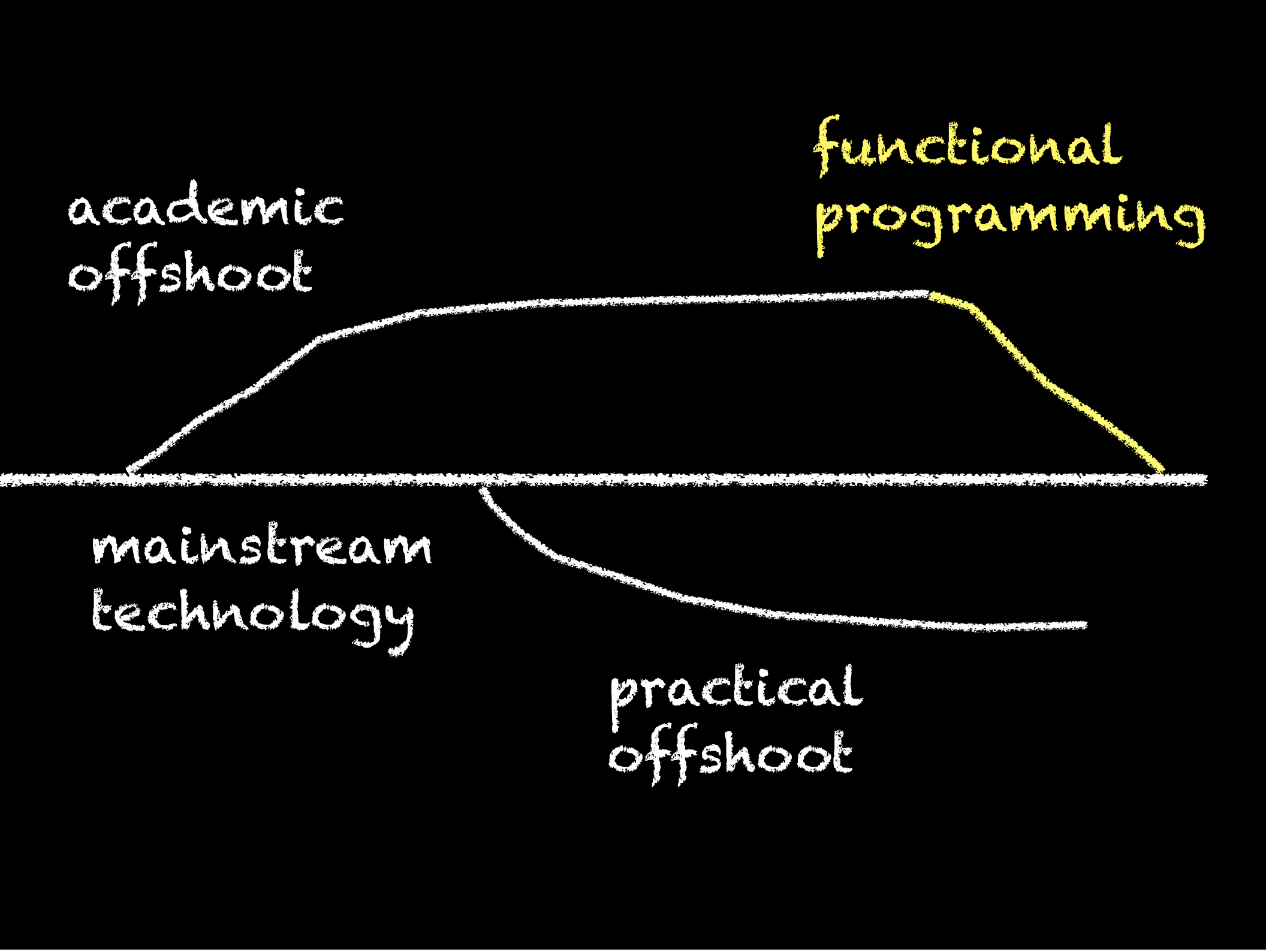
UndoManager.execute()

undo()

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



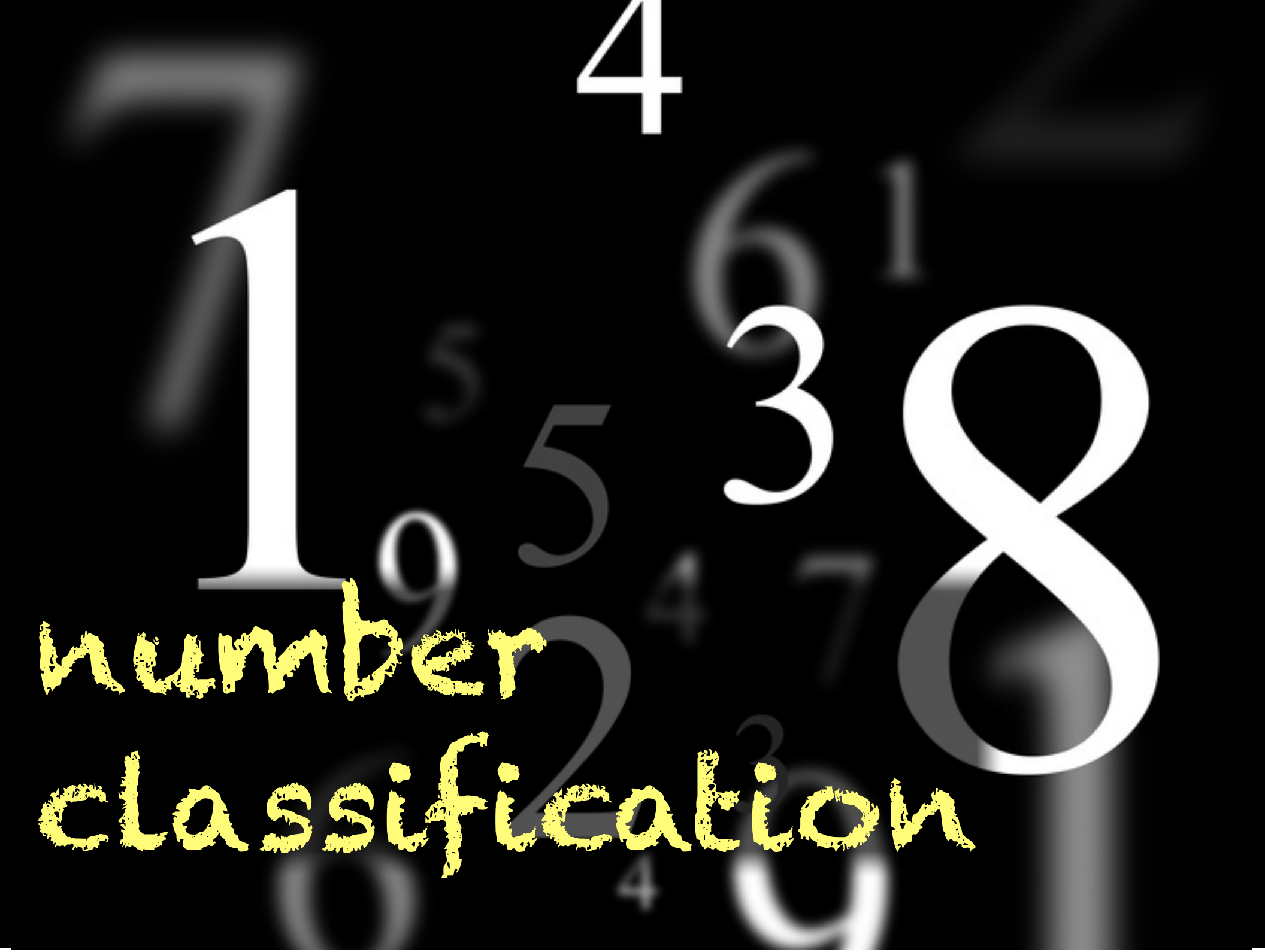

65
1970
1975
1980
Forth
1968
FIG-Forth
1978
PostScript
1982
Forth-83
1983
Logo
1968
FORTRAN IV
(Fortran 66 ANS)
1966
FORTRAN V
(Fortran 77 ANSI)
april 1978
Prolog
1970
Prolog II
october 1982
Prolog III
1984
LCOMP
1965
MUMPS
1966
MUMPS (ANSI)
september 15, 1977
Sharp
APL
B
1981
COBOL 68 ANS
1968
COBOL 74 ANSI
1974
Simula 67
1967
Pascal
1970
PL/M
1972
Modula
1975
Modula 2
1979
Ada
1979
Rex 1.00
may 1979
C++
1983
PL/1 ANS
1976
C
1971
C (K&R)
1978
Classic C
Objective-C
1983
CORAL 66
1966
C with Classes
april 1980
C++
july 1983
Simula 67
1967
CLU
1974
ALGOL W
1966
Mesa
1977
ALGOL 68
december
1968
Cedar
1983
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


academic
offshoot
functional
programming
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"


number
classification

perfect

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

(sum of the factors of a #) - # = #

(sum of the factors of a #) = 2#

6: 1 + 2 + 3 + 6 = 12 (2x6)

28: 1 + 2 + 4 + 7 + 14 + 28 = 56 (2x28)

496: . . .

classification

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

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

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

imperative

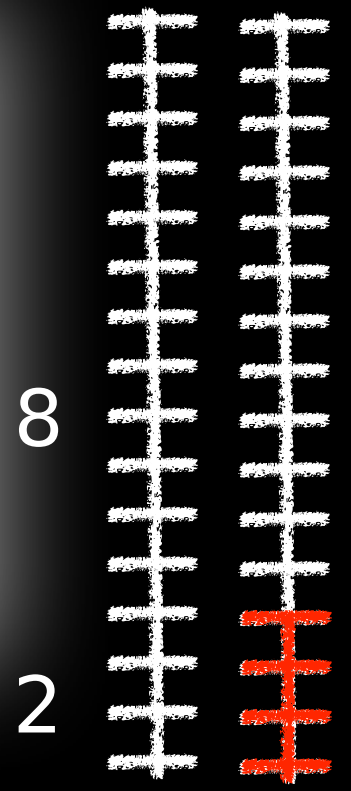
imperative classifier

```
package com.nealford.conf.tdd.perfectnumbers;

import java.util.Set;
import java.util.HashSet;
import static java.lang.Math.sqrt;

public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {
        if (number < 1)
            throw new InvalidNumberException(
            "Can't classify negative numbers");
        _number = number;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
    }

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

    public Set<Integer> getFactors() {
        return _factors;
    }

    private void calculateFactors() {
        for (int i = 1; i <= sqrt(_number) + 1; i++)
            if (isFactor(i))
                addFactor(i);
    }

    private void addFactor(int factor) {
        _factors.add(factor);
        _factors.add(_number / factor);
    }

    private int sumOfFactors() {
        calculateFactors();
        int sum = 0;
        for (int i : _factors)
            sum += i;
        return sum;
    }

    public boolean isPerfect() {
        return sumOfFactors() - _number == _number;
    }

    public boolean isAbundant() {
        return sumOfFactors() - _number > _number;
    }

    public boolean isDeficient() {
        return sumOfFactors() - _number < _number;
    }

    public static boolean isPerfect(int number) {
        return new Classifier6(number).isPerfect();
    }
}
```

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {
        if (number < 1)
            throw new InvalidNumberException(
            "Can't classify negative numbers");
        _number = number;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
    }

    public boolean isPerfect() {…

    public boolean isAbundant() {…

    public boolean isDeficient() {…

    public static boolean isPerfect(int number) {…
}
```

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {…

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

    public Set<Integer> getFactors() {
        return _factors;
    }

    public boolean isPerfect() {…

    public boolean isAbundant() {…

    public boolean isDeficient() {…

    public static boolean isPerfect(int number) {…
}
```

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {…

    private boolean isFactor(int factor) {…

    public Set<Integer> getFactors() {…

    private void calculateFactors() {
        for (int i = 2; i < sqrt(_number) + 1; i++)
            if (isFactor(i))
                addFactor(i);
    }

    private void addFactor(int factor) {
        _factors.add(factor);
        _factors.add(_number / factor);
    }
}
```



```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {…

    private boolean isFactor(int factor) {…

    public Set<Integer> getFactors() {…

    private void calculateFactors() {…

    private void addFactor(int factor) {…

    private int sumOfFactors() {
        calculateFactors();
        int sum = 0;
        for (int i : _factors)
            sum += i;
        return sum;
    }
}
```

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {…

    private boolean isFactor(int factor) {…

    public Set<Integer> getFactors() {…

    private void calculateFactors() {…

    privat
    privat
    public
    public
    public
    public
}
```

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

public boolean isAbundant() {
    return sumOfFactors() - _number > _number;
}

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

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {…

    private boolean isFactor(int factor) {…

    public Set<Integer> getFactors() {…

    private void calculateFactors() {…

    private void addFactor(int factor) {…

    private int sumOfFactors() {…

    public boolean isPerfect() {…

    public boolean isAbundant() {…

    public boolean isDeficient() {…

    public static boolean isPerfect(int number) {…
}
```

internal state

cohesive

composed

testable

refactorable

(slightly more) functional

```
public class NumberClassifier {

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

    public int sum(Set<Integer> factors) {…

    public boolean isPerfect(int number) {…

    public boolean isAbundant(int number) {…

    public boolean isDeficient(int number) {…
}
```

(slightly) more functional classifier


```
package com.nealford.conf.ft.numberclassifier;

import static java.lang.Math.sqrt;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class NumberClassifier {

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

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

    static public boolean isPerfect(int number) {
        return sum(factors(number)) - number == numbe
    }

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

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

```
public class NumberClassifier {

    public boolean isFactor(int number, int potential_factor) {…

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

```
public class NumberClassifier {

    public boolean isFactor(int number, int potential_factor) {…

    public Set<Integer> factors(int number) {…

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

```
public class NumberClassifier {

    public boolean isFactor(int number, int potential_factor) {…

    public Set<Integer> factors(int number) {…

    pub
    pub
    pub
    pub
}
```

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

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

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

no internal state

```
public class NumberClassifier {

    static public boolean isFactor(int number, int potential_factor) {…

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

    static public int sum(Set<Integer> factors) {…

    static public boolean isPerfect(int number) {…

    static public boolean isAbundant(int number) {…

    static public boolean isDeficient(int number) {…
}
```

less need for scoping refactorable testable

"functional" is more a way of thinking than a tool set

1st class/
higher order
functions

pure
functions

concepts

strict evaluation

recursion

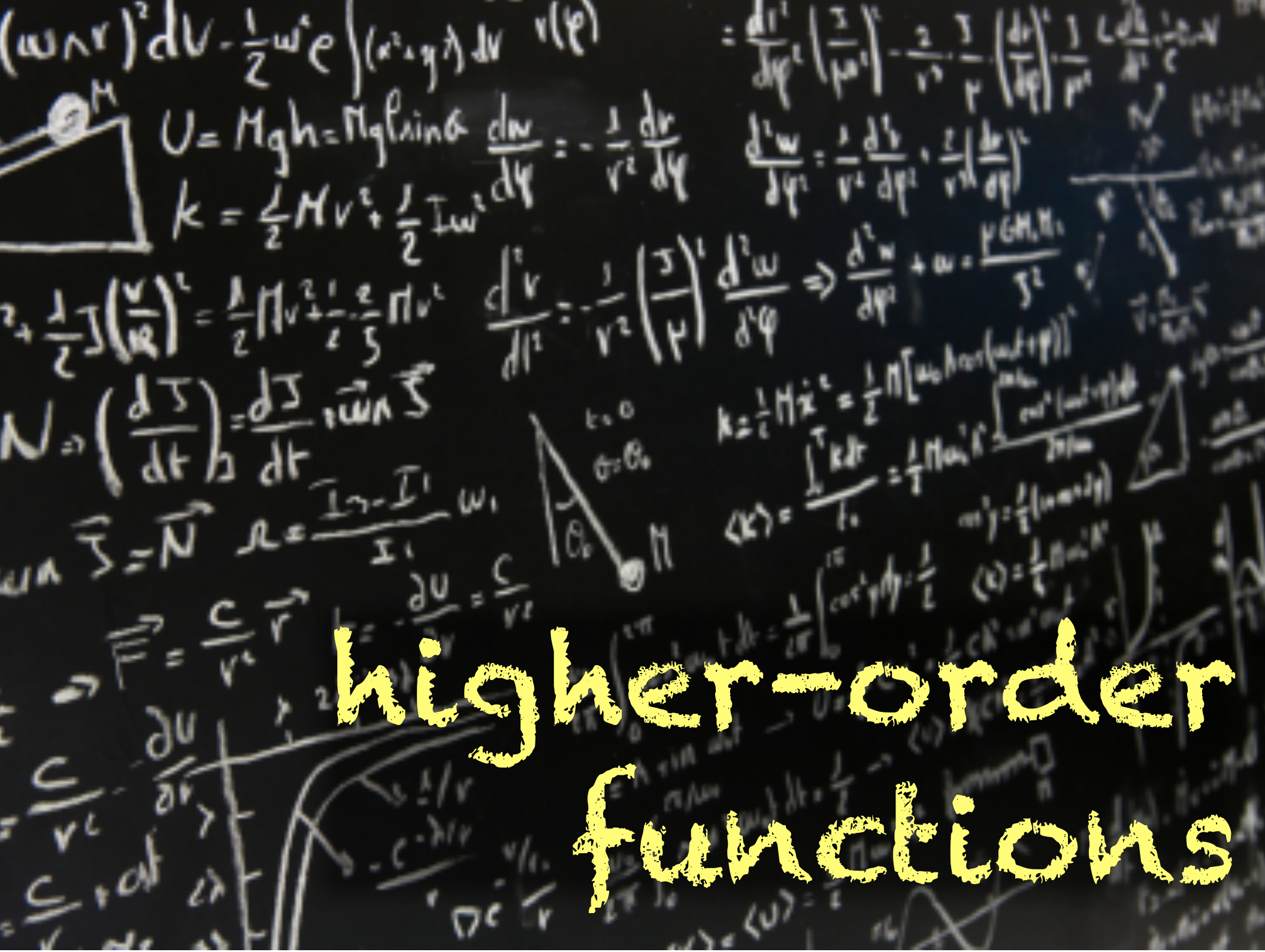
1st class /
higher-order
functions

pure
functions

concepts

strict evaluation

recursion


higher-order
functions

higher-order functions

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

extracting reusable code:

- extract via Template Method DP
- encapsulate verb via Command DP

```
public void addOrderFrom(ShoppingCart cart, String userName,
                         Order order) throws SQLException {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (SQLException sqlx) {
        rollbackTransaction();
        throw sqlx;
    } finally {
        cleanUp();
    }
}
```

```
public void addOrderFrom(ShoppingCart cart, String userName,
                         Order order) throws Exception {
    setupDataInfrastructure();
    try {
        add(order, userKeyBasedOn(userName));
        addLineItemsFrom(cart, order.getOrderKey());
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}
```

```
public void wrapInTransaction(Command c) throws Exception {
    setupDataInfrastructure();
    try {
        c.execute();
        completeTransaction();
    } catch (Exception condition) {
        rollbackTransaction();
        throw condition;
    } finally {
        cleanUp();
    }
}

public void addOrderFrom(final ShoppingCart cart,
    final String userName, final Order order) {
    wrapInTransaction(new Command() {
        public void execute() {
            add(order, userKeyBasedOn(userName));
            addLineItemsFrom(cart, order.getOrderKey());
        }
    });
}
```

same “unit of work” code

```
public class OrderDbClosure {
    def wrapInTransaction(command) {
      setupDataInfrastructure()
      try {
        command()
        completeTransaction()
      } catch (RuntimeException ex) {
        rollbackTransaction()
        throw ex
      } finally {
        cleanUp()
      }
    }

    def addOrderFrom(cart, userName, order) {
      wrapInTransaction {
        add order, userKeyBasedOn(userName)
        addLineItemsFrom cart, order.orderKey
      }
    }
}
```


```groovy
def wrapInTransaction(command) {
  setupDataInfrastructure()
  try {
    command()
    completeTransaction()
  } catch (Exception ex) {
    rollbackTransaction()
    throw ex
  } finally {
    cleanUp()
  }
}

def addOrderFrom(cart, userName, order) {
  wrapInTransaction {
    add order, userKeyBasedOn(userName)
    addLineItemsFrom cart, order.getOrderKey()
  }
}
```

```
def addOrderFrom(cart, userName, order) {
  wrapInTransaction {
    add order, userKeyBasedOn(userName)
    addLineItemsFrom cart, order.getOrderKey()
  }
}
```

What's so special about...

closures

minimal code to demonstrate closures

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

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

c2 = makeCounter()

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

```groovy
def makeCounter() {
    def very_local_variable = 0
    return { very_local_variable += 1 }
}

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

c2 = makeCounter()

println "C1 = ${c1()}, C2 = ${c2()}"
```

```
closures » groovy MakeCounter.groovy
C1 = 4, C2 = 1
```

closest Java equivalent to closure code

```
class Counter {
    public int varField;

    Counter(int var) {
        varField = var;
    }

    public static Counter makeCounter() {
        return new Counter(0);
    }

    public int execute() {
        return ++varField;
    }
}
```

```
public class Counter {
    public int varField;

    public Counter(int var) {
        varField = var;
    }

    public static Counter makeCounter() {
        return new Counter(0);
    }

    public int execute() {
        return ++varField;
    }
}
```

Let the Language manage state

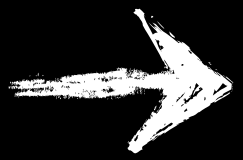
Languages handle

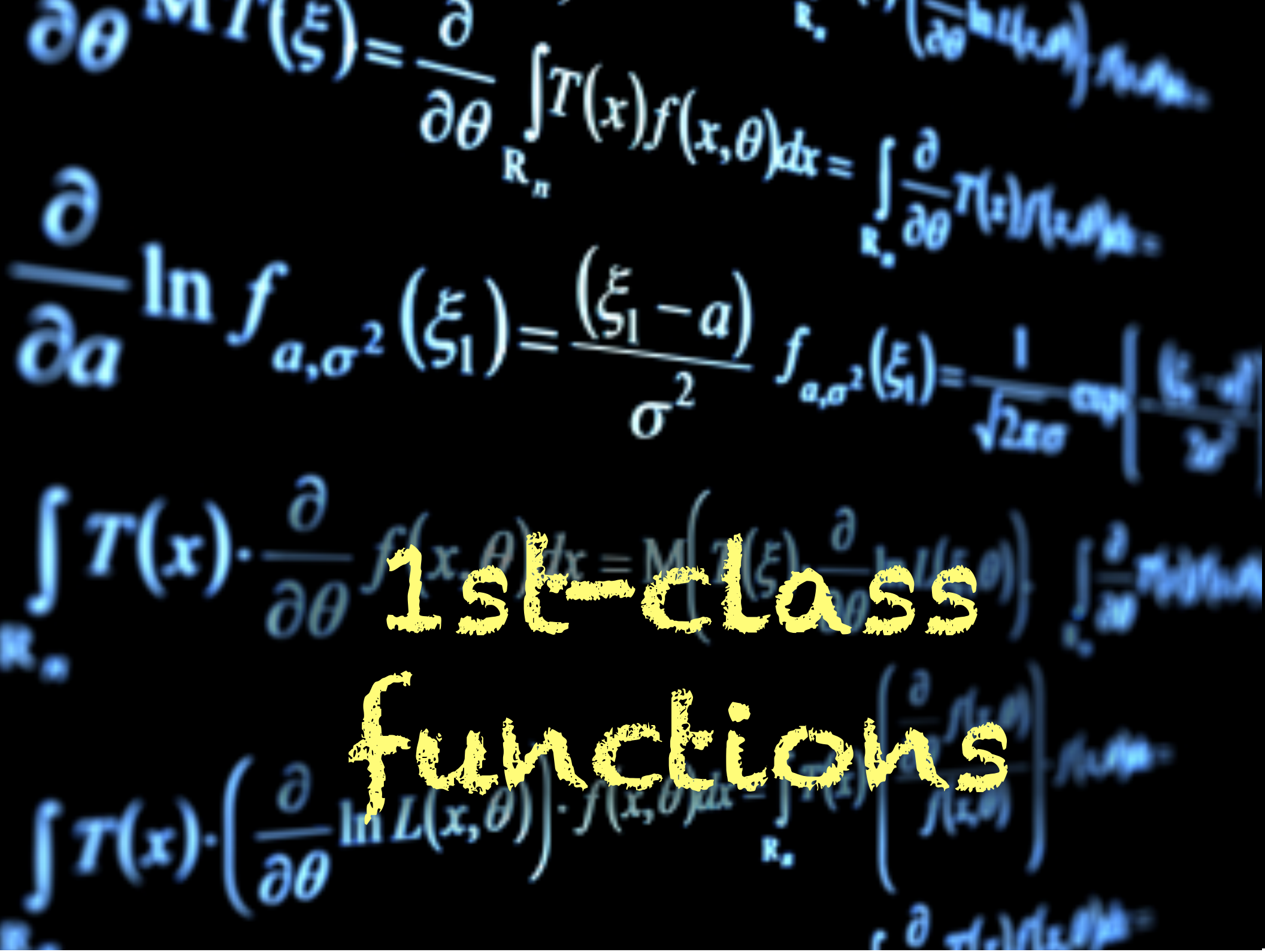
memory allocation

garbage collection

concurrency

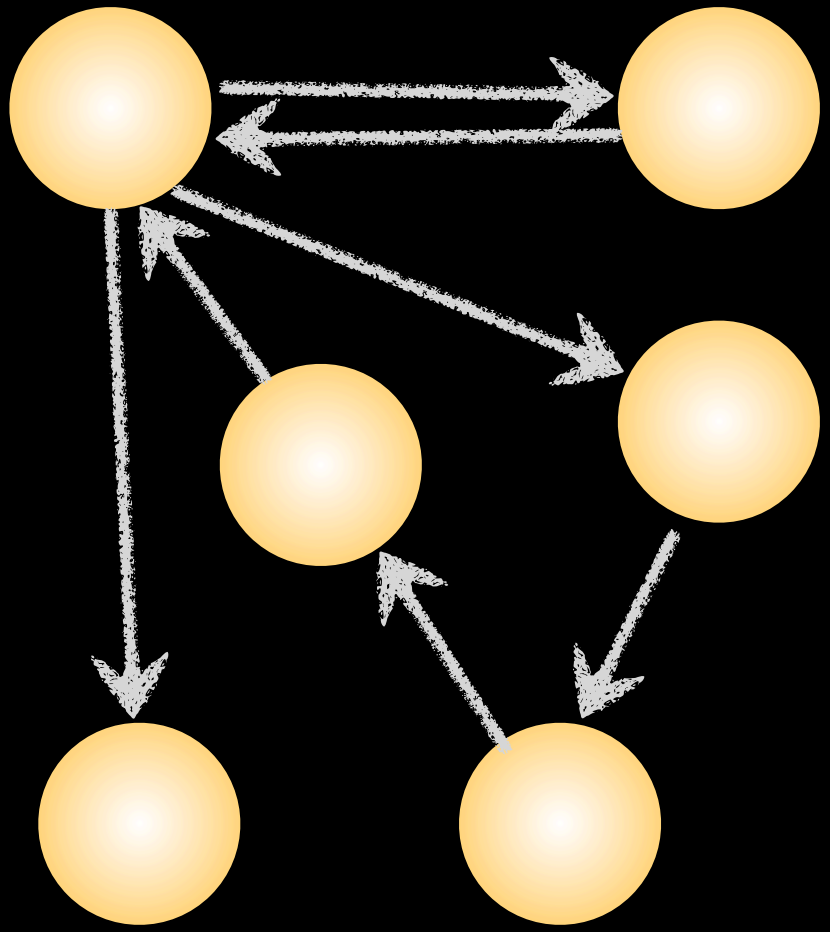
state

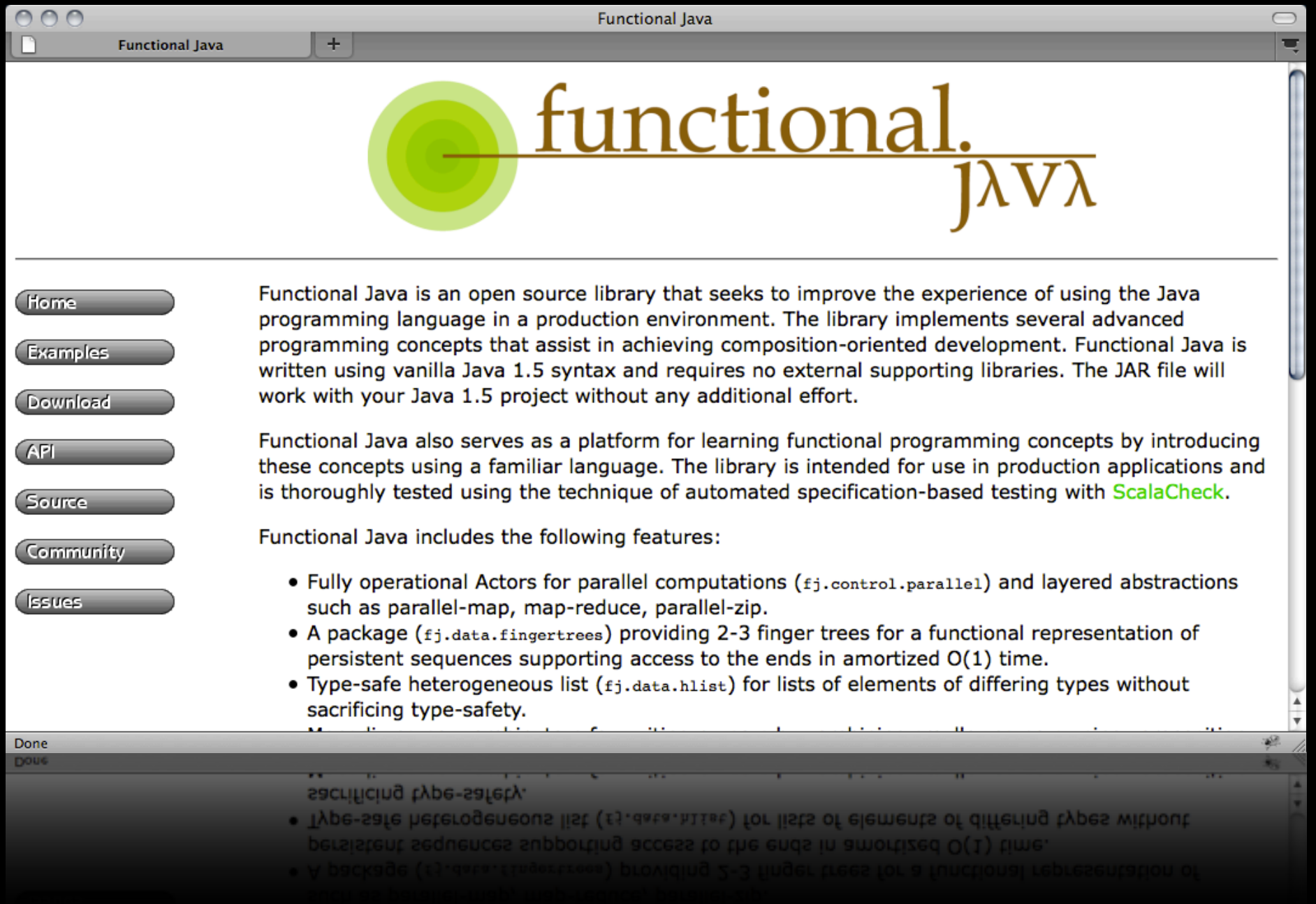
tests  specification-based testing frameworks


1st-class
functions

1st-class functions

functions can appear anywhere other language constructs can appear




Functional Java
Functional Java
functional.
java
Home
Examples
Download
API
Source
Community
Issues
Functional Java is an open source library that seeks to improve the experience of using the Java programming language in a production environment. The library implements several advanced programming concepts that assist in achieving composition-oriented development. Functional Java is written using vanilla Java 1.5 syntax and requires no external supporting libraries. The JAR file will work with your Java 1.5 project without any additional effort.
Functional Java also serves as a platform for learning functional programming concepts by introducing these concepts using a familiar language. The library is intended for use in production applications and is thoroughly tested using the technique of automated specification-based testing with ScalaCheck.
Functional Java includes the following features:
Fully operational Actors for parallel computations (fj.control.parallel) and layered abstractions such as parallel-map, map-reduce, parallel-zip.
A package (fj.data.fingertrees) providing 2-3 finger trees for a functional representation of persistent sequences supporting access to the ends in amortized O(1) time.
Type-safe heterogeneous list (fj.data.hlist) for lists of elements of differing types without sacrificing type-safety.
Done

NumberClassifier using Functional Java

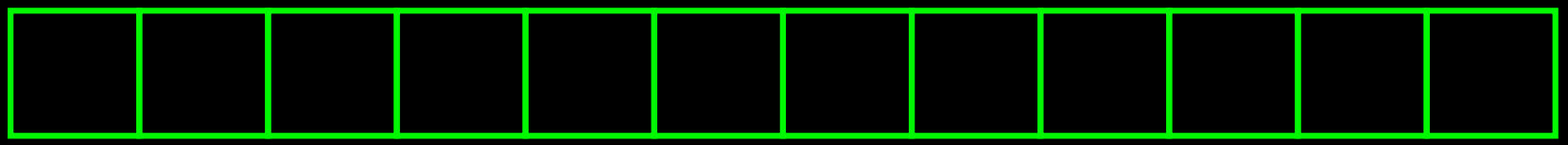
```
package com.nealford.conf.ft.numberclassifier;

import fj.F;
import fj.data.List;
import static fj.data.List.range;
import static fj.function.Integers.add;

import static java.lang.Math.round;
import static java.lang.Math.sqrt;

public class FNumberClassifier {

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

    public List<Integer> factorsOf(final int number) {
        return range(1, number+1)
                .filter(new F<Integer, Boolean>() {
                    public Boolean f(final Integer i) {
                        return isFactor(number, i);
                    }
                });
    }

    public int sum(List<Integer> factors) {
        return factors.foldLeft(fj.function.Integers.add, 0);
    }

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

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

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

```
public class FNumberClassifier {

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

    public List<Integer> factorsOf(final int number) {
        return range(1, number+1).filter(new F<Integer, Boolean>() {
                    public Boolean f(final Integer i) {
                        return isFactor(number, i);
                    }
                });
    }

    public int sum(List<Integer> factors) {
        return factors.foldLeft(fj.function.Integers.add, 0);
    }

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

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

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

```
  public int sum(List<Integer> factors) {
      return factors.foldLeft(add, 0);
  }
```

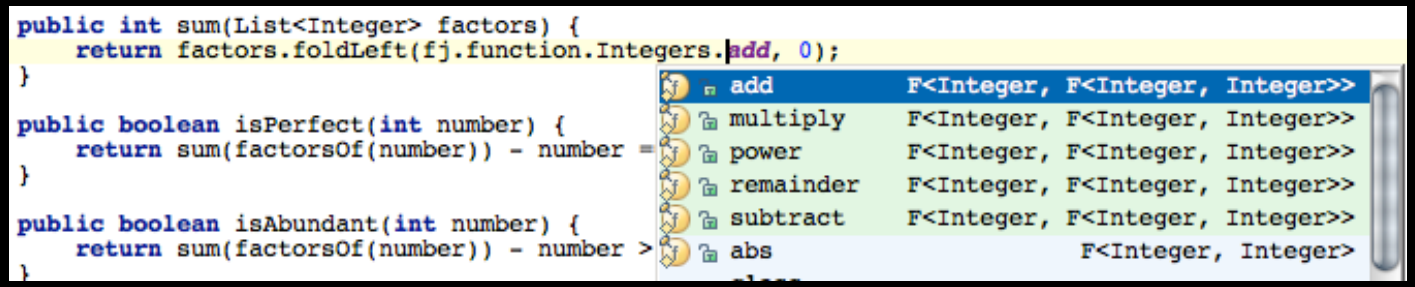
```
public int sum(List<Integer> factors) {
    return factors.foldLeft(fj.function.Integers.add, 0);
}
```

fold



```
public int sum(List<Integer> factors) {
    return factors.foldLeft(add, 0);
}
```

```
public int sum(List<Integer> factors) {
    return factors.foldLeft(fj.function.Integers.add, 0);
}
```




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

think about results, not steps

```
public List<Integer> factorsOf(final int number) {
    return range(1, number + 1)
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
}
```

12

1 2 3 4 5 6 7 8 9 10 11 12

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

public List<Integer> factorsOf(final int number) {
    return range(1, number + 1)
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
}

public int sum(List<Integer> factors) {
    return factors.foldLeft(0, function.integerAdd);
}
```

think about results,
not steps


academia
alert!

currying

given: $f : (X \times Y) \to Z$

then: $\mathrm{curry}(f) : X \to (Y \to Z)$

currying transforms a multi-argument function so that it can be called as a chain of single-argument functions

partial application

partial application fixes a number of arguments to a function, producing another function of smaller arity

currying & partial application

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

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

println "4x4: ${quadrate.call(4)}"
println "5x8: ${octate(5)}"
```

```groovy
def quadrate_ = { y ->
  return 4 * y
}
```

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

println "increment 7: ${incrementer(7)}"
```

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

println "composition of curried functions yields ${thirtyTwoer(2)}"
```


```groovy
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)

println "The volume of the 2x3x4 rectangle is ${volume(2, 3, 4)}"
println "The area of the 3x4 square is ${area(3, 4)}"
println "The length of the 6 line is ${lengthPA(6)}"
println "The length of the 6 line via curried function is ${lengthC(6)}"
```

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

return a version that always multiplies by 4

```groovy
def quadrate = product.curry(4)
```

==

```groovy
def quadrate_ = { y ->
  return 4 * y
}
```

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

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

println "4x4: ${quadrate.call(4)}"
println "5x8: ${octate(5)}"
```

currying
vs
partial application

```
def volume = {h, w, l -> h * w * l}
```

partial application

currying
vs
partial application

partial application

```
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)
```

currying

function reuse

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

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

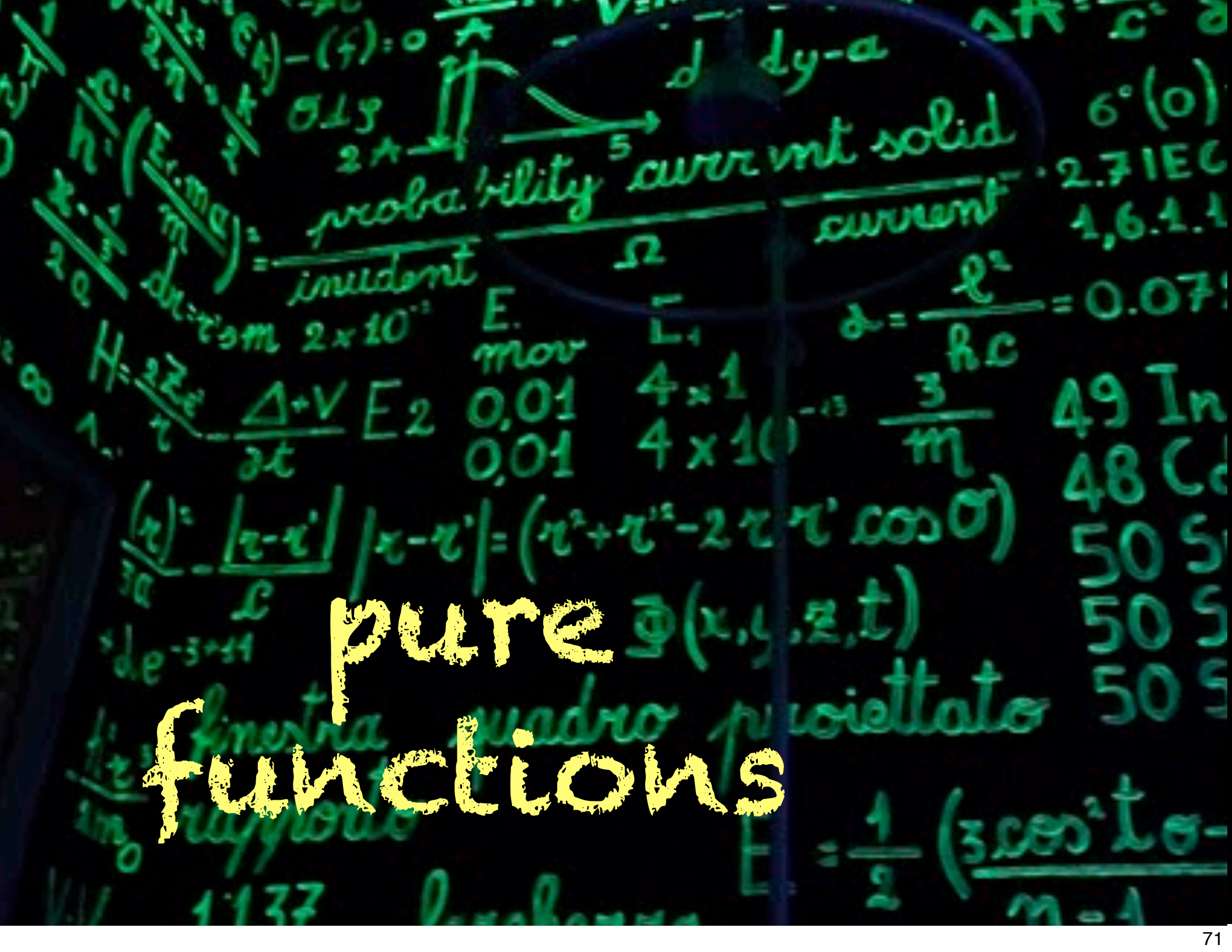
new, different tools

currying (& recursion)

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```

currying

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```


pure
functions

pure functions

no memory or i/o side effects

purity

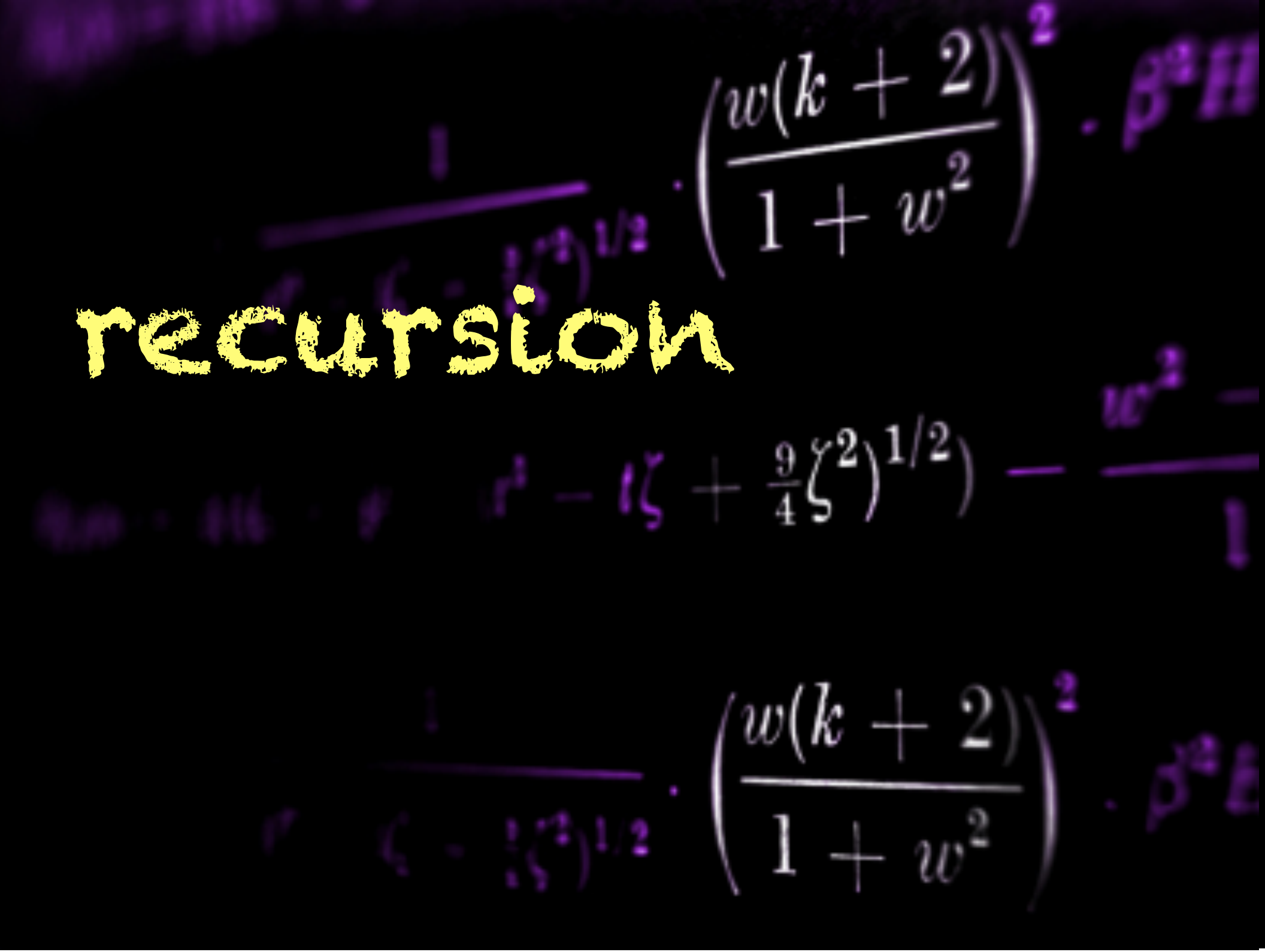
if the result isn't used, it can be removed

a particular invocation with a set of parameters returns a constant value

enables memoization

execution order can change

parallel execution


recursion

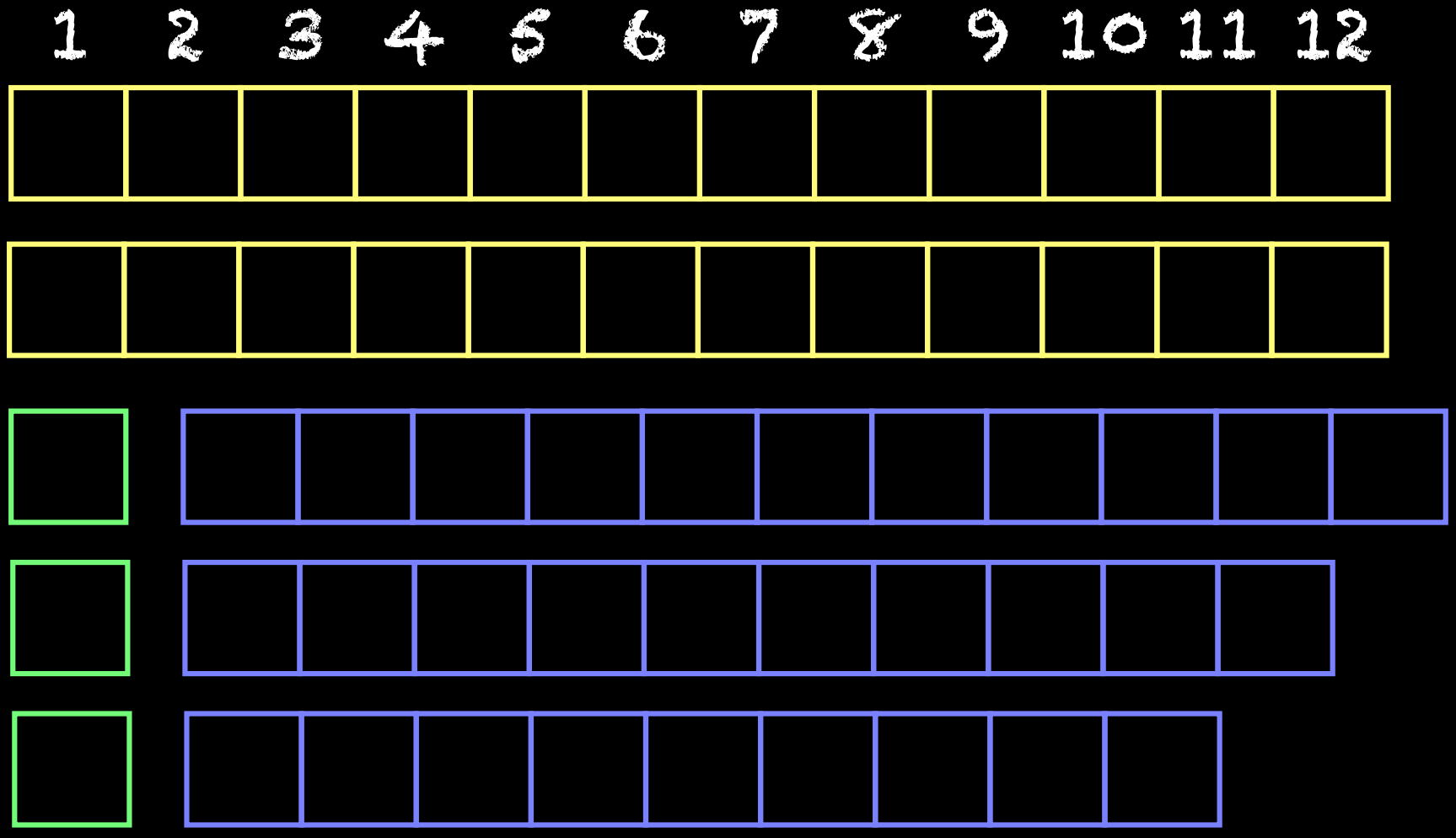
iteration & recursion

```groovy
def perfectNumbers = [6, 28, 496, 8128]

def iterateList(listOfNums) {
  listOfNums.each { n ->
    println "${n}"
  }
}

iterateList(perfectNumbers)

def recurseList(listOfNums) {
  if (listOfNums.size == 0) return;
  println "${listOfNums.head()}"
  recurseList(listOfNums.tail())
}

recurseList(perfectNumbers)
```

iterate a List

```
def perfectNumbers = [6, 28, 496, 8128]

def iterateList(listOfNums) {
  listOfNums.each { n ->
    println "${n}"
  }
}

iterateList(perfectNumbers)
```

List perspective



recurse a list

```
def recurseList(listOfNums) {
  if (listOfNums.size == 0) return;
  println "${listOfNums.head()}"
  recurseList(listOfNums.tail())
}

recurseList(perfectNumbers)
```

iterative & recursive filtering


```groovy
def filter(list, criteria) {
  def new_list = []
  list.each { i ->
    if (criteria(i))
      new_list << i
  }
  return new_list
}

modBy2 = { n -> n % 2 == 0}

l = filter(1..20, modBy2)
```

```groovy
def filter(list, p) {
  if (list.size() == 0) return list
  if (p(list.head()))
    [] + list.head() + filter(list.tail(), p)
  else filter(list.tail(), p)
}

filter(1..20, {n-> n % 2 == 0}).each {i-> println "${i}"}
```

iterative filtering

```
def filter(list, criteria) {
  def new_list = []
  list.each { i ->
    if (criteria(i))
      new_list << i
  }
  return new_list
}

modBy2 = { n -> n % 2 == 0}

l = filter(1..20, modBy2)
```

recursive filtering

```
def filter(list, p) {
  if (list.size() == 0) return list
  if (p(list.head()))
    return [] + list.head() + filter(list.tail(), p)
  else return filter(list.tail(), p)
}
```

functional vs imperative

```
def filter(list, p) {
  if (list.size() == 0) return list
  if (p(list.head()))
    return [] + list.head() + filter(list.tail(), p)
  else return filter(list.tail(), p)
}
```

```
def filter(list, criteria) {
  def new_list = []
  list.each { i ->
    if (criteria(i))
      new_list << i
  }
  return new_list
}
```

who's minding the state?

recursive filtering

```
object CurryTest extends Application {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))
  println(filter(nums, dividesBy(3)))
}
```

think about results, not steps

http://www.scala-lang.org/node/135

think about results, not steps

what about things you want to control?

performance?

new, different tools

imperative number classifier

```
public class Classifier6 {
    private Set<Integer> _factors;
    private int _number;

    public Classifier6(int number) {…

    private boolean isFactor(int factor) {…

    public Set<Integer> getFactors() {…

    private void calculateFactors() {
        for (int i = 2; i < sqrt(_number) + 1; i++)
            if (isFactor(i))
                addFactor(i);
    }

    private void addFactor(int factor) {
        _factors.add(factor);
        _factors.add(_number / factor);
    }
}
```

optimized!

optimized functional classifier

```
public List<Integer> factorsOfOptimized(final int number) {
    final List<Integer> factors = range(1, (int) round(sqrt(number) + 1))
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
    return factors.append(factors.map(new F<Integer, Integer>() {
        public Integer f(final Integer i) {
            return number / i;
        }
    }))
    .nub();
}
```

optimized factors

```
public List<Integer> factorsOfOptimized(final int number) {
    final List<Integer> factors = range(1, (int) round(sqrt(number) + 1))
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
    return factors.append(factors.map(new F<Integer, Integer>() {
        public Integer f(final Integer i) {
            return number / i;
        }
    }))
            .nub();
}
```

think about results, not steps

special thanks to Michal Karwanski for optimizations to my optimizations

post-imperative

Google challenged college grads to write code for 100 CPU computers...

...they failed

http://broadcast.oreilly.com/2008/11/warning-x-x-1-may-be-hazardous.html

ingrained imperativity

learn MapReduce

http://code.google.com/edu/submissions/mapreduce-minilecture/listing.html

sound familiar?

Languages handle

garbage collection

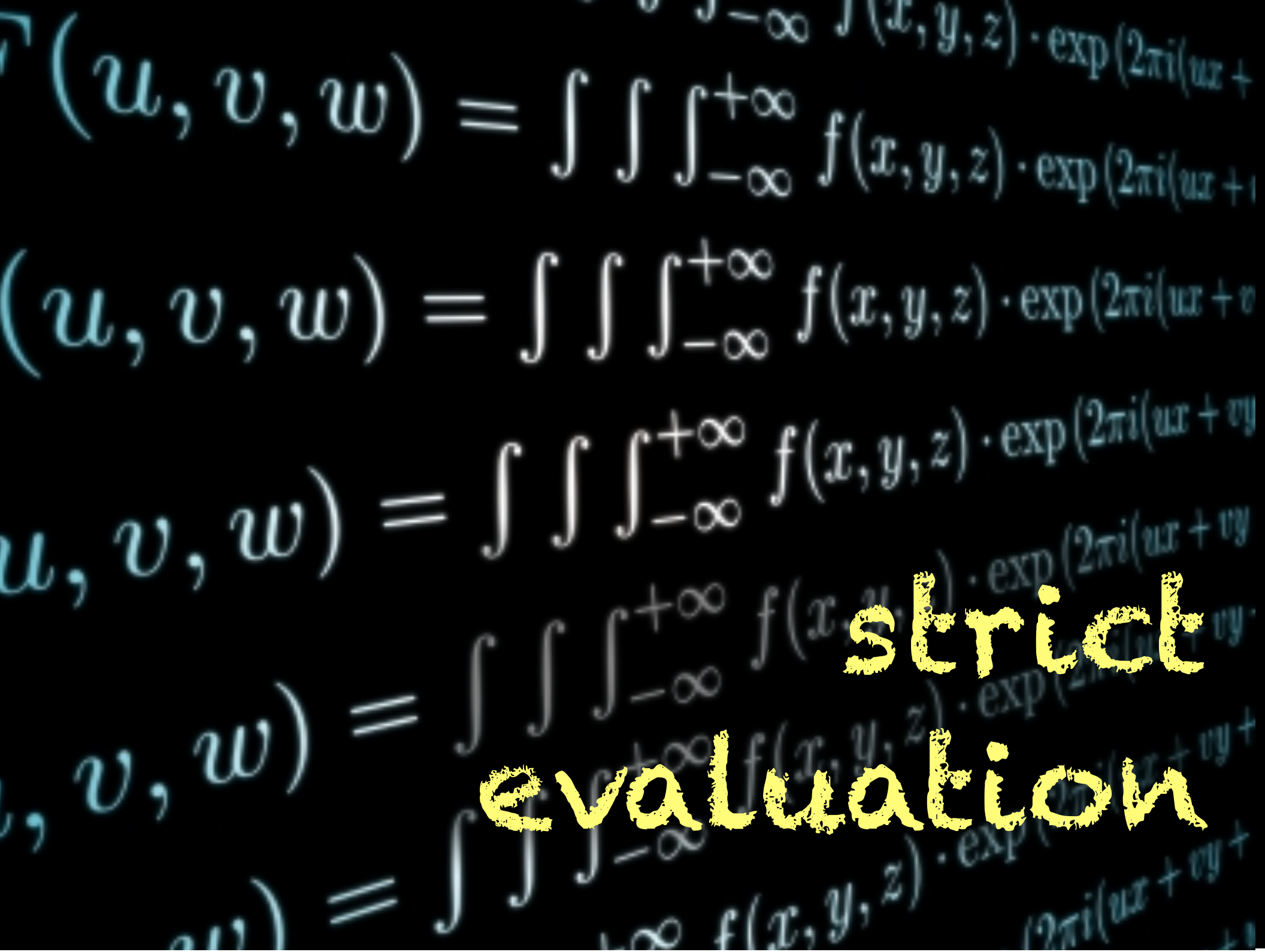
concurrency

state

tests

iteration

. . .


strict
evaluation


academia
alert!

strict evaluation

all elements pre-evaluated

divByZero

```
print length([2+1, 3*2, 1/0, 5-4])
```

=4

non-strict evaluation

elements evaluated as needed


Laziness

Lazy List

```
class LazyList {
    private head, tail

    LazyList(head, tail) {
        this.head = head;
        this.tail = tail
    }

    def LazyList getTail() { tail ? tail() : null }

    def List getHead(n) {
        def valuesFromHead = [];
        def current = this
        n.times {
            valuesFromHead << current.head
            current = current.tail
        }
        valuesFromHead
    }

    def LazyList filter(Closure p) {
        if (p(head))
            p.owner.prepend(head, { getTail().filter(p) })
        else
            getTail().filter(p)
    }
}
```

Laziness

```
class LazyList {
    private head, tail

    LazyList(head, tail) {
        this.head = head;
        this.tail = tail
    }

    def List getHead(n) {
        def valuesFromHead = [];
        def current = this
        n.times {
            valuesFromHead << current.head
            current = current.tail
        }
        valuesFromHead
    }

    def LazyList filter(Closure p) {
        if (p(head))
            p.owner.prepend(head, { getTail().filter(p) })
        else
            getTail().filter(p)
    }
}
```

Lazy list in 


```groovy
def prepend(val, closure) { new LazyList(val, closure) }

def integers(n) { prepend(n, { integers(n + 1) }) }

@Test
public void lazy_list_acts_like_a_list() {
    def naturalNumbers = integers(1)
    assertEquals('1 2 3 4 5 6 7 8 9 10',
        naturalNumbers.getHead(10).join(' '))
    def evenNumbers = naturalNumbers.filter { it % 2 == 0 }
    assertEquals('2 4 6 8 10 12 14 16 18 20',
        evenNumbers.getHead(10).join(' '))
}
```

Laziness

```
def prepend(val, closure) { new LazyList(val, closure) }

def integers(n) { prepend(n, { integers(n + 1) }) }

@Test
public void lazy_list_acts_like_a_list() {
    def naturalNumbers = integers(1)
    assertEquals('1 2 3 4 5 6 7 8 9 10',
        naturalNumbers.getHead(10).join(' '))
    def evenNumbers = naturalNumbers.filter { it % 2 == 0 }
    assertEquals('2 4 6 8 10 12 14 16 18 20',
        evenNumbers.getHead(10).join(' '))
}
```

nomenclature


```
foldLeft()      inject()

filter()        findAll()
```

Tersest NumberClassifier


```
class NumberClassifier {
    static def factorsOf(number) {
        (1..number).findAll { i -> number % i == 0 }
    }

    static def isPerfect(number) {
        factorsOf(number).inject(0, {i, j -> i + j}) == 2 * number
    }

    static def nextPerfectNumberFrom(n) {
        while (! isPerfect(++n)) ;
        n
    }
}
```

perfect #'s

```
class NumberClassifier {
    static def factorsOf(number) {
        (1..number).findAll { i -> number % i == 0 }
    }

    static def isPerfect(number) {
        factorsOf(number).inject(0, {i, j -> i + j}) == 2 * number
    }

    static def nextPerfectNumberFrom(n) {
        while (! isPerfect(++n)) ;
        n
    }
}
```

∞ perfect # sequence

```
import static com.nealford.ft.allaboutlists.NumberClassifier.nextPerfectNumberFrom

def prepend(val, closure) { new LazyList(val, closure) }

def perfectNumbers(n) { prepend(n,
    { perfectNumbers(nextPerfectNumberFrom(n)) }) };

@Test
public void infinite_perfect_number_sequence() {
    def perfectNumbers =
        perfectNumbers(nextPerfectNumberFrom(1))
    assertEquals([6, 28, 496], perfectNumbers.getHead(3))
}
```

∞ sequence

```
import static com.nealford.ft.allaboutlists.NumberClassifier.nextPerfectNumberFrom

def prepend(val, closure) { new LazyList(val, closure) }

def perfectNumbers(n) { prepend(n,
    { perfectNumbers(nextPerfectNumberFrom(n)) }) };

@Test
public void infinite_perfect_number_sequence() {
    def perfectNumbers =
        perfectNumbers(nextPerfectNumberFrom(1))
    assertEquals([6, 28, 496], perfectNumbers.getHead(3))
}
```

new, different tools

concurrency

$$\oint D\,dA = \int_V \rho\,dV = Q$$

$$\oint E\,dl = -\frac{d}{dt}\int_A B\,dA$$

$$\oint B\,dA = 0$$

$$\oint H\,dl = \int_A J\,dA + \frac{d}{dt}\int_A D\,dA$$

http://www.infoq.com/presentations/Simple-Made-Easy

Simple Made Easy

Rich Hickey

variables

assume 1 thread of control, 1 timeline

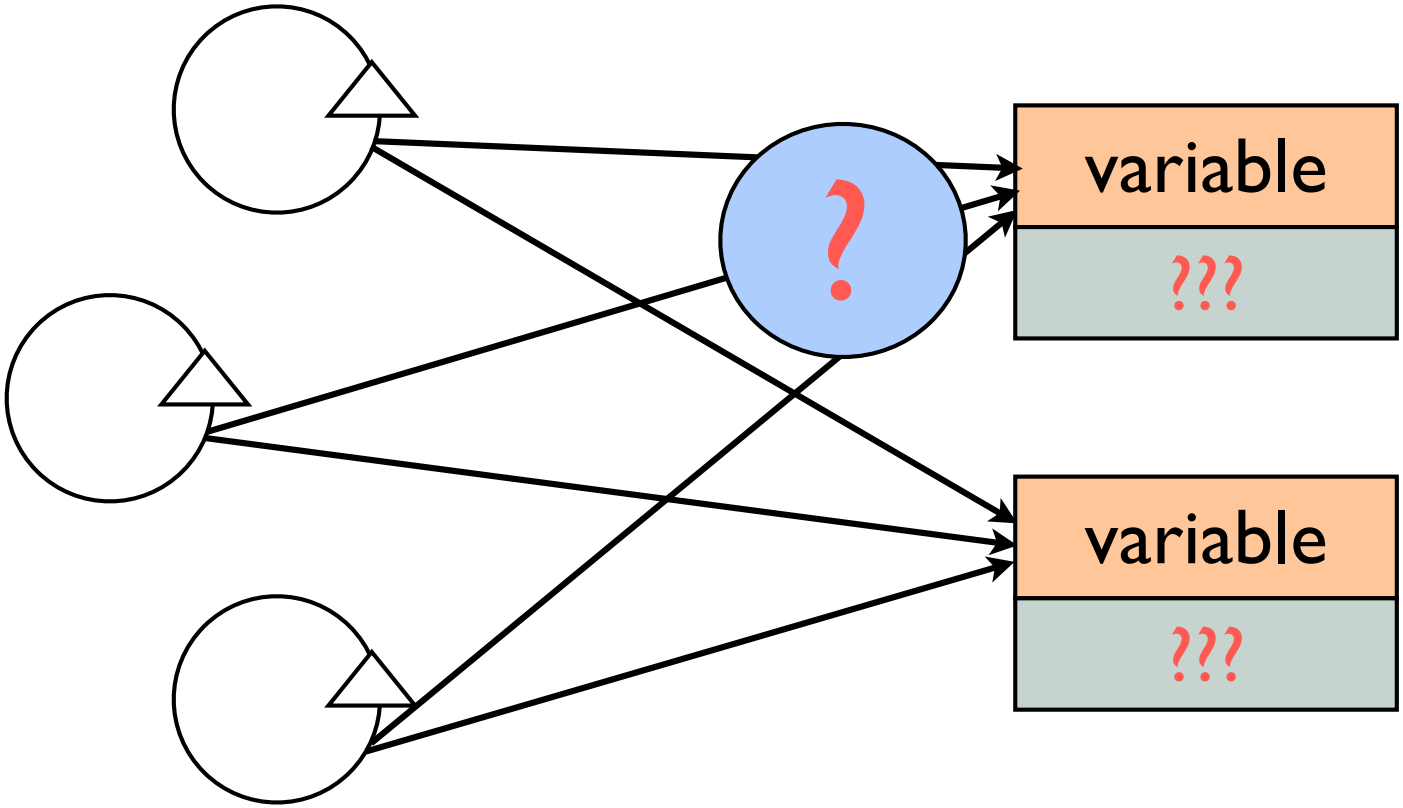
not atomic

non-composable

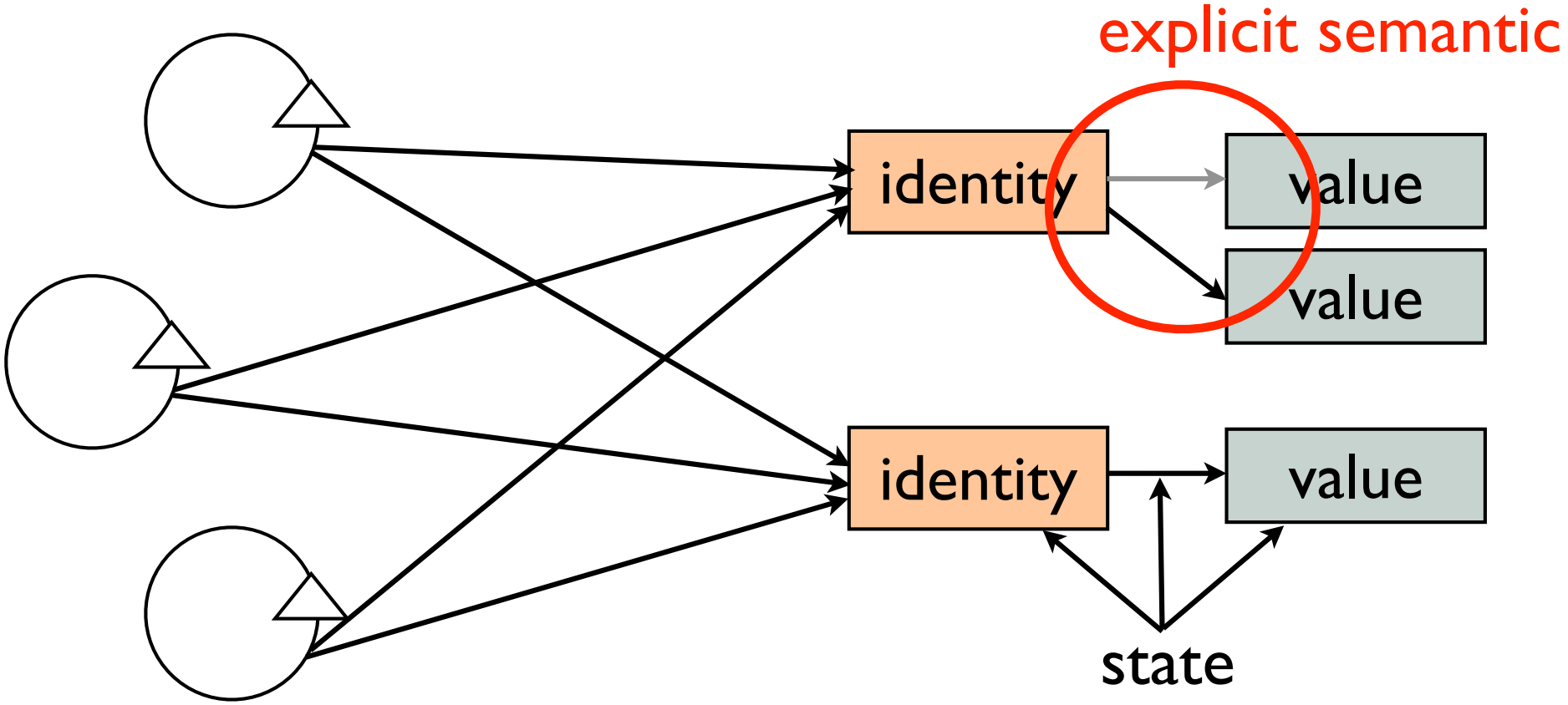
subtle visibility rules

with concurrency: **lock & pray**

Life w/ variables



identity

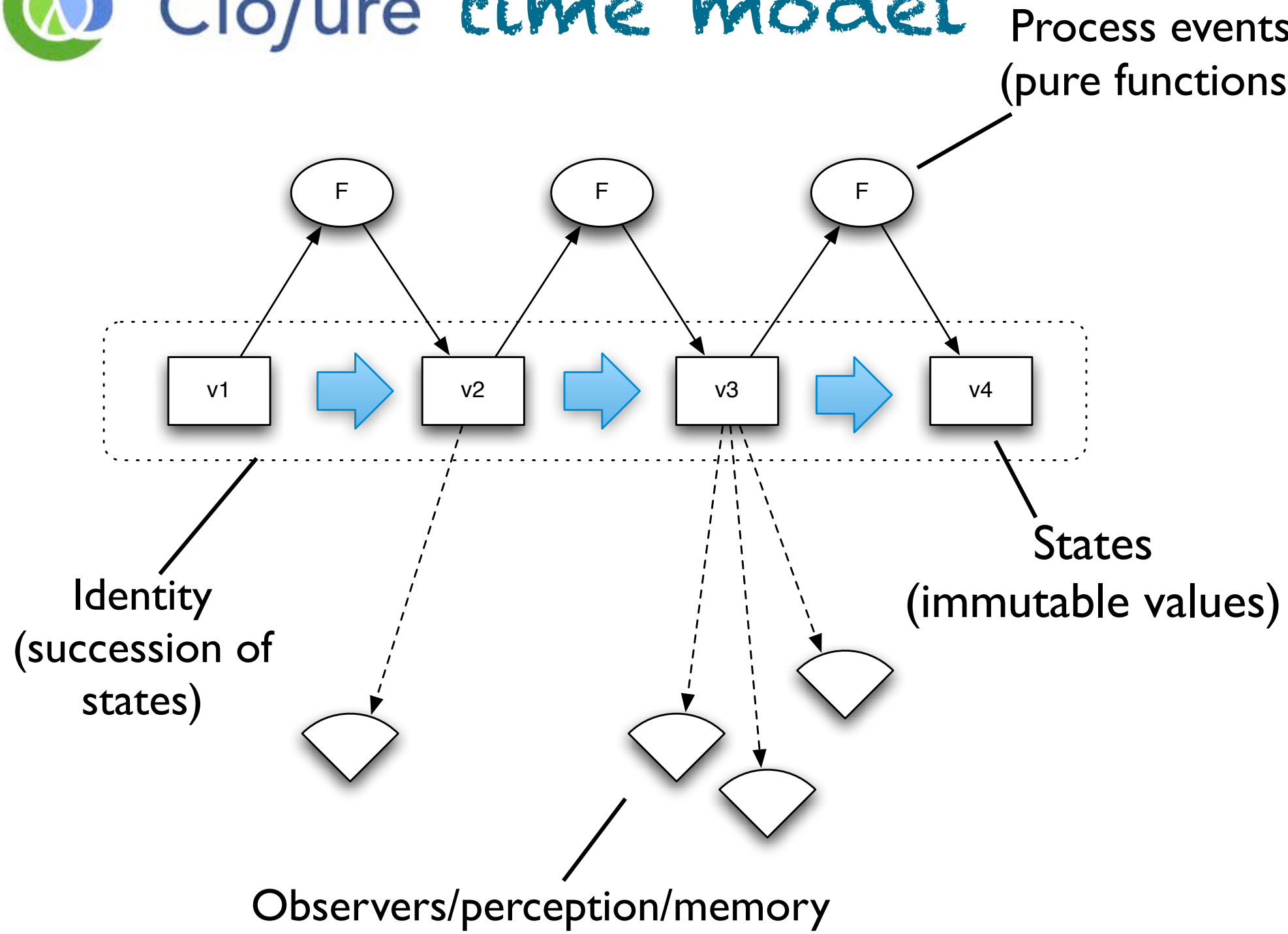

explicit semantic
identity
value
value
identity
value
state

identity, state, & time

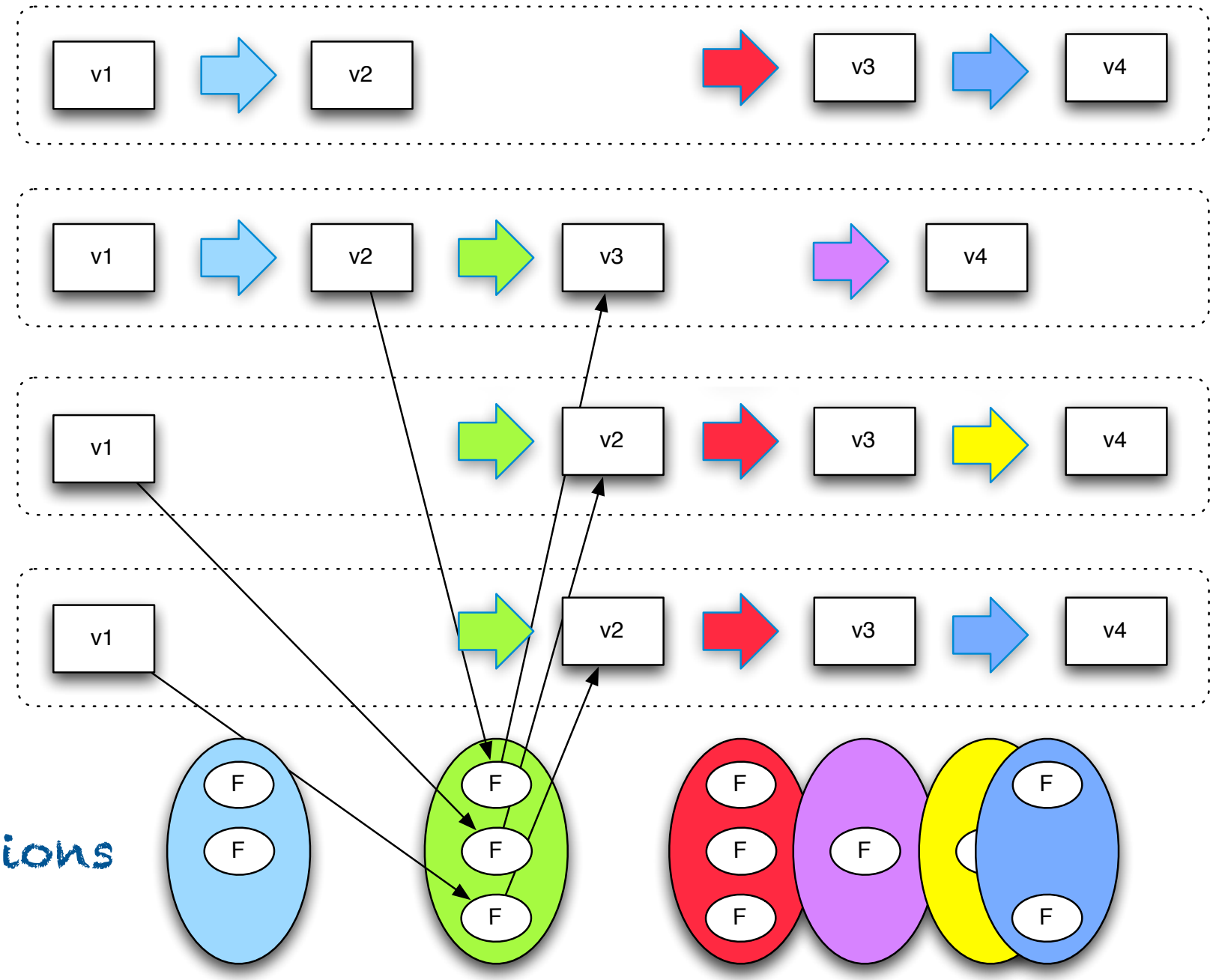
term	meaning
value	immutable data in a persistent data structure
identity	series of causally related values over time
state	identity at a point in time
time	relative: before/ simultaneous/after ordering of causal values

Clojure

Clojure time model



Clojure's Software Transactional Memory



transactions

$$\delta_{ijt} = \frac{(v_{ijt} / \sum_{i=1}^{n} v_{ijt}) \times 100 - (r_{ijt} / \sum_{i=1}^{n} r_{ijt}) \times 100}{\sigma(\delta_{ij})}$$

actors

PerfectNumber

```
import scala.actors.Actor._

object NumberClassifier extends Application {
  def isPerfect(candidate: Int) =
  {
    val RANGE = 1000000
    val numberOfPartitions = (candidate.toDouble / RANGE).ceil.toInt

    val caller = self

    for (i <- 0 until numberOfPartitions) {
      val lower = i * RANGE + 1;
      val upper = candidate min (i + 1) * RANGE

      actor {
        var partialSum = 0
        for(j <- lower to upper)
          if (candidate % j == 0) partialSum += j

        caller ! partialSum
      }
    }

    var responseExpected = numberOfPartitions
    var sum = 0
    while(responseExpected > 0) {
      receive {
        case partialSum : Int =>
          responseExpected -= 1
          sum += partialSum
      }
    }

    sum == 2 * candidate
  }

  println("6 is perfect? " + isPerfect(6))
  println("496 is perfect? " + isPerfect(496))
  println("33550336 is perfect? " + isPerfect(33550336))
  println("33550337 is perfect? " + isPerfect(33550337))
}
```

```
def isPerfect(candidate: Int) =
{
  val RANGE = 1000000
  val numberOfPartitions = (candidate.toDouble / RANGE).ceil.toInt

  val caller = self

  for (i <- 0 until numberOfPartitions) {
    val lower = i * RANGE + 1;
    val upper = candidate min (i + 1) * RANGE

    actor {
      var partialSum = 0
      for(j <- lower to upper)
        if (candidate % j == 0) partialSum += j

      caller ! partialSum
    }
  }

  var responseExpected = numberOfPartitions
  var sum = 0
  while(responseExpected > 0) {
    receive {
      case partialSum : Int =>
        responseExpected -= 1
        sum += partialSum
    }
  }

  sum == 2 * candidate
}
```

new, different tools


thinking
functionally

immutability over state transitions

http://www.ibm.com/developerworks/java/library/j-jtp02183/index.html

immutable ...

simple to construct & test

automatically thread safe

do not need a copy constructor

does not need an implementation of clone

make good `Map` keys & `Set` elements

no need for defensive copying

has "failure atomicity"

immutable !

ensure the class cannot be overridden

make fields final

all state set in constructor

no mutating methods

defensively copy mutable object fields

Let the Language manage state as much as possible

```groovy
@Immutable
class Client {
    String name, city, state, zip
    List<String> streets
}
```

@Immutable

properties have private, final backing fields

updates result in `ReadOnlyPropertyException`

map & tuple style constructors generated

default `equals`, `hashCode`, `toString` methods

defensive copies of mutable references

Arrays & cloneables use `clone()`

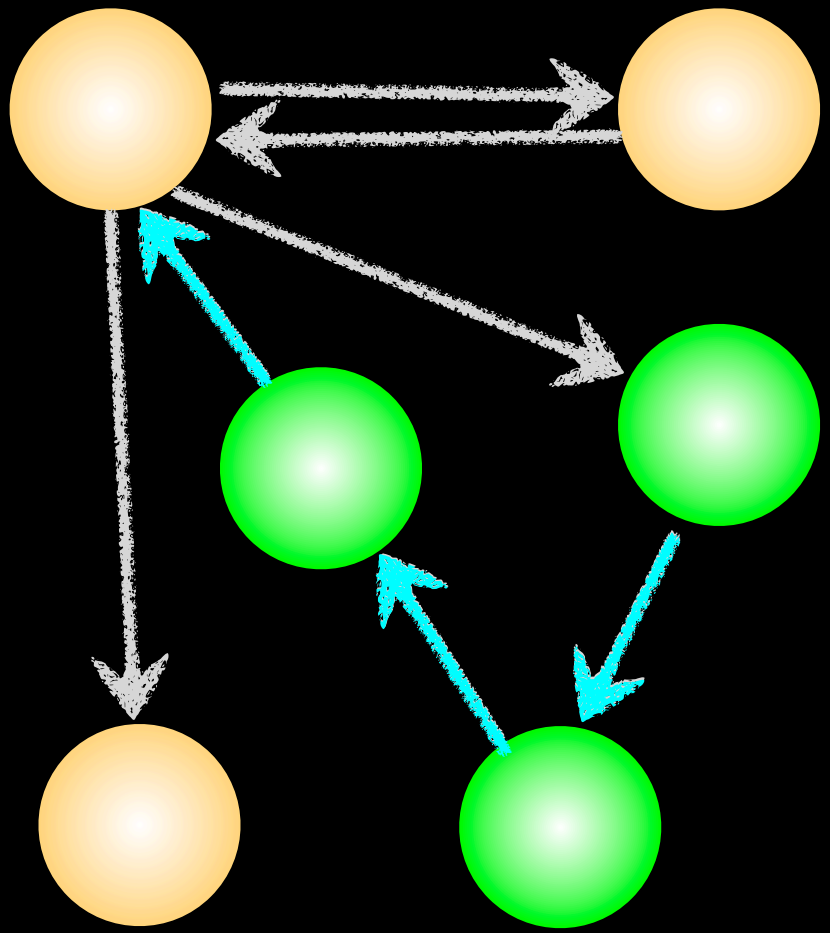
collections wrapped in immutable wrappers

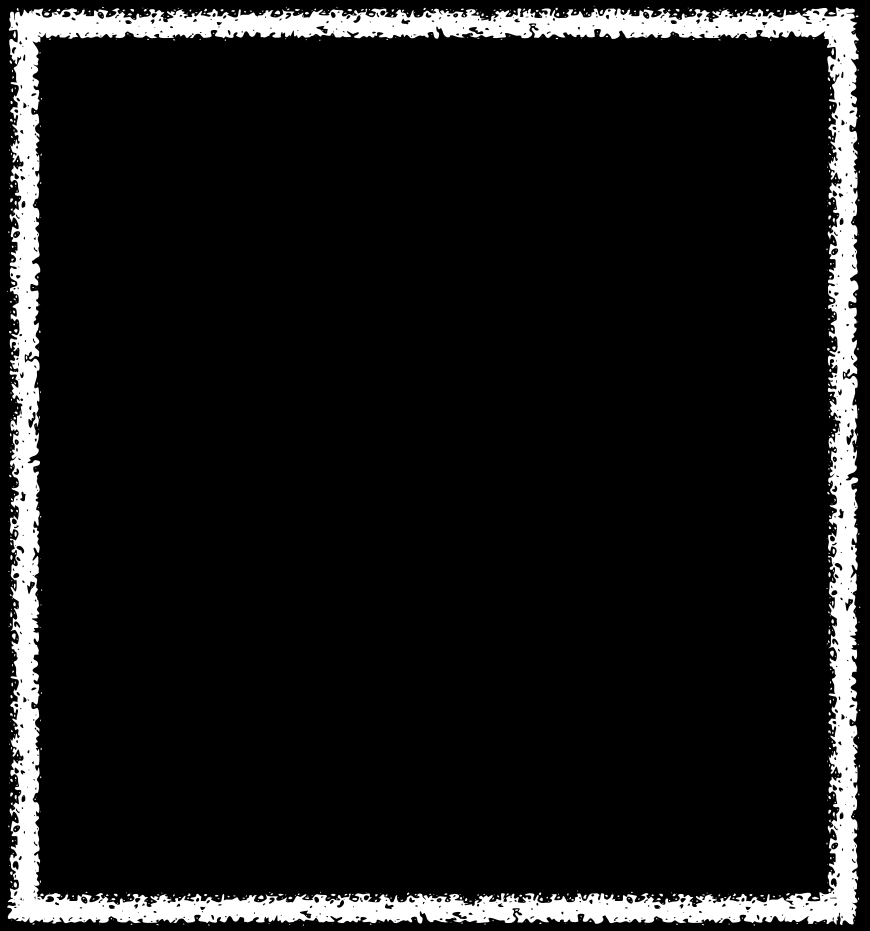
updates result in `UnsupportedOperationException`

results
over
steps

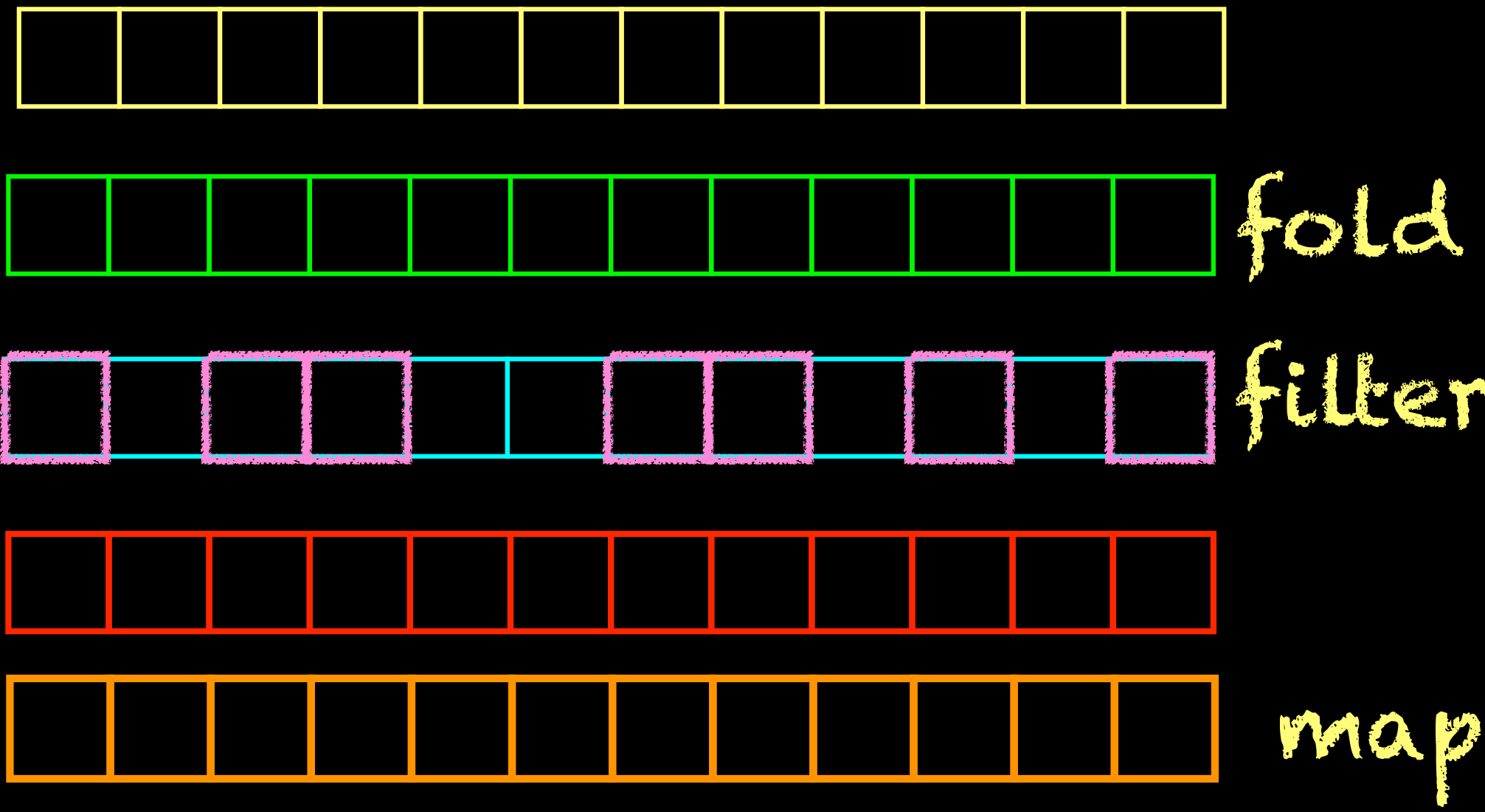
composition
over
structure

structural reuse

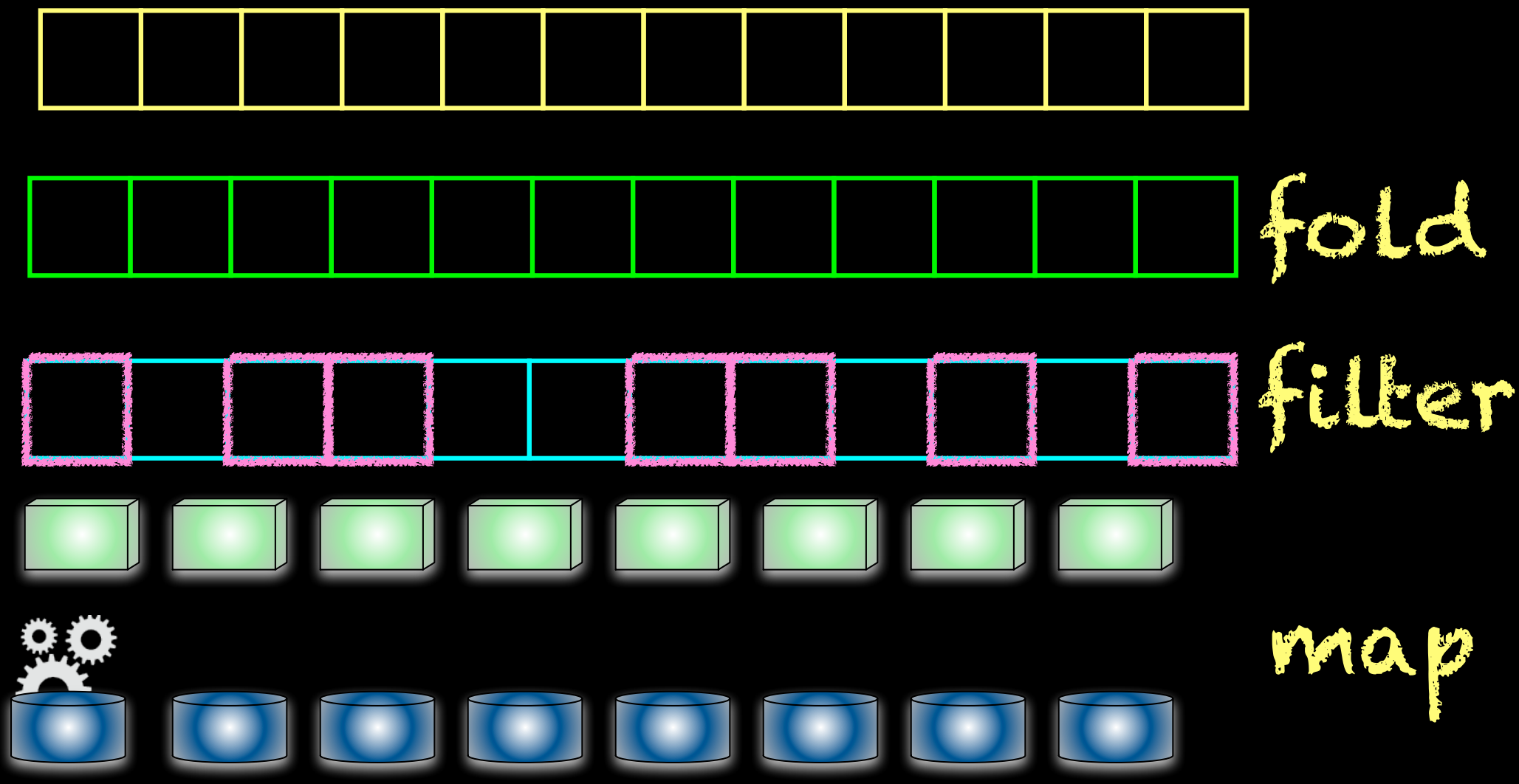


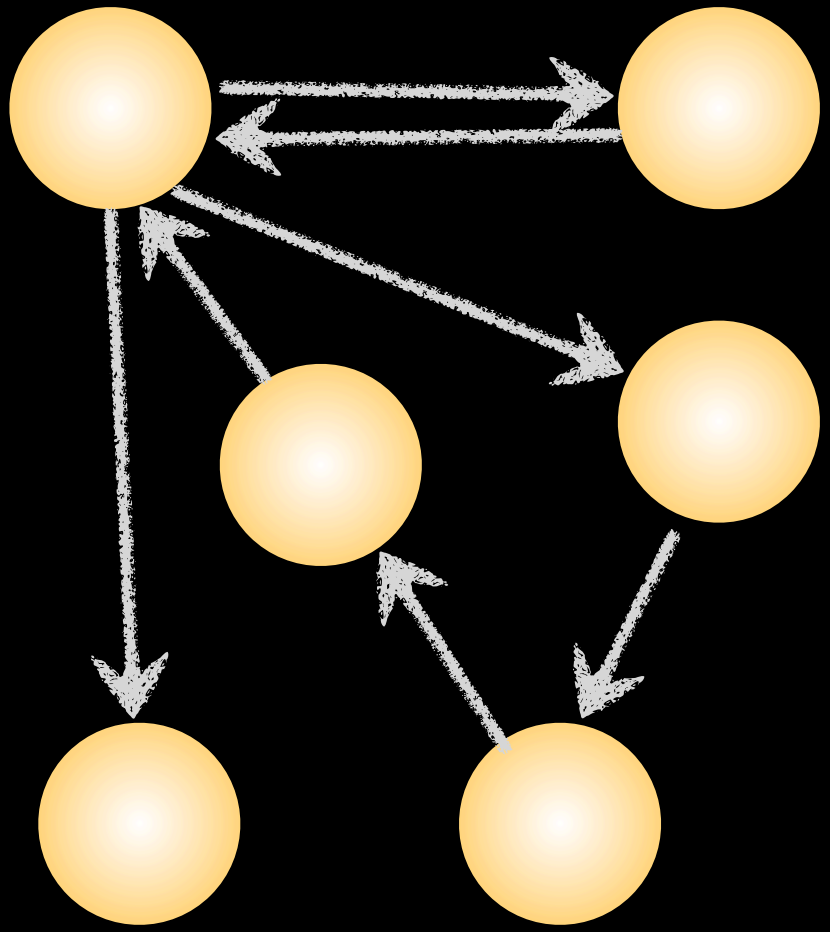


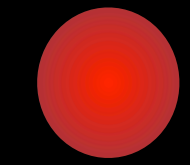
functional reuse



functional reuse

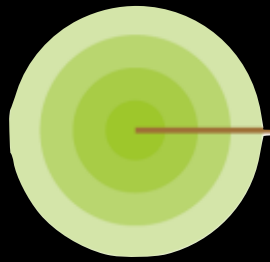






composition

declarative
over
imperative

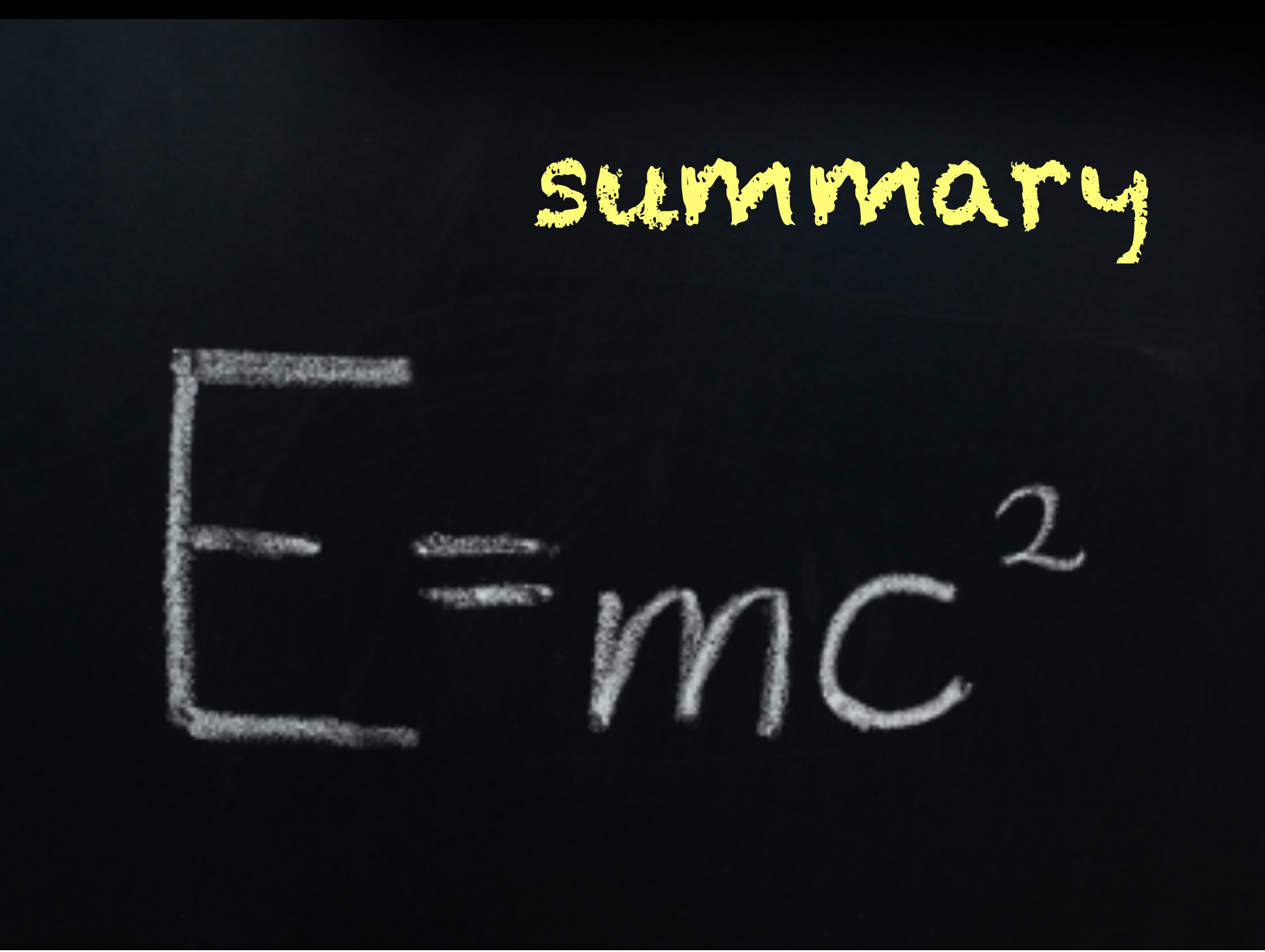


functional.
JAVA

paradigm over tool

Clojure

summary



functional thinking

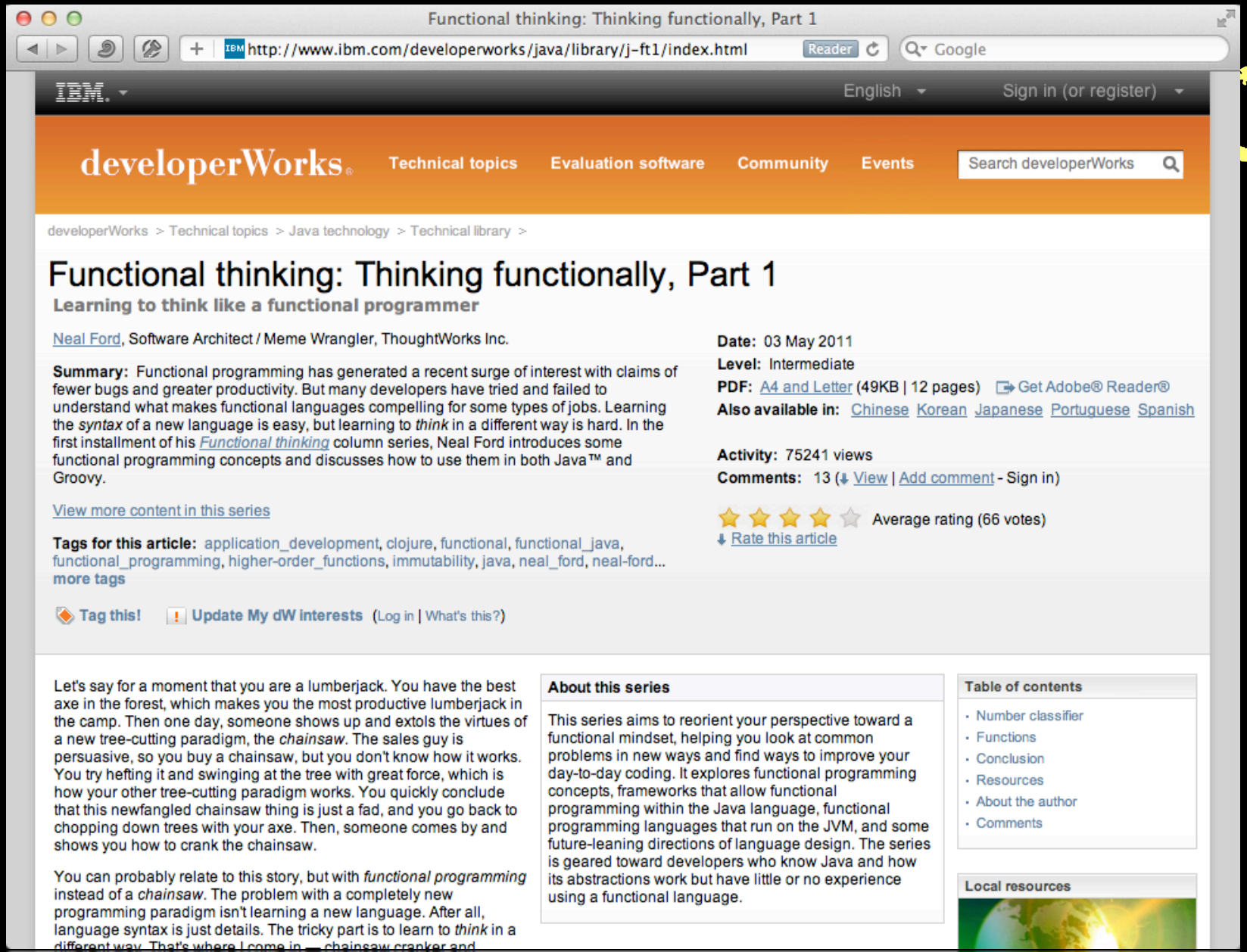
new ways of thinking

cede control to languages & runtimes

immediately beneficial beginning steps

following the general trend in language design

enables entirely new capabilities



http://ibm.co/nf-ft

?'s

please fill out the session evaluations

NEAL FORD director / software architect
meme wrangler

ThoughtWorks®

nford@thoughtworks.com
2002 Summit Boulevard, Atlanta, GA 30319
nealford.com
thoughtworks.com
memeagora.blogspot.com
@neal4d