ThoughtWorks®

Functional Thinking

NEAL FORD software architect / meme wrangler

ThoughtWorks®

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
blog: memeagora.blogspot.com
twitter: neal4d

NF

tortured

3 ^ metaphors

& a history lesson

assign to x

$$x := x + 1$$

solve for x?

Paradigms

~~Languages~~ are tools.

Learning a new one takes time.

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

Execution in the Kingdom of Nouns

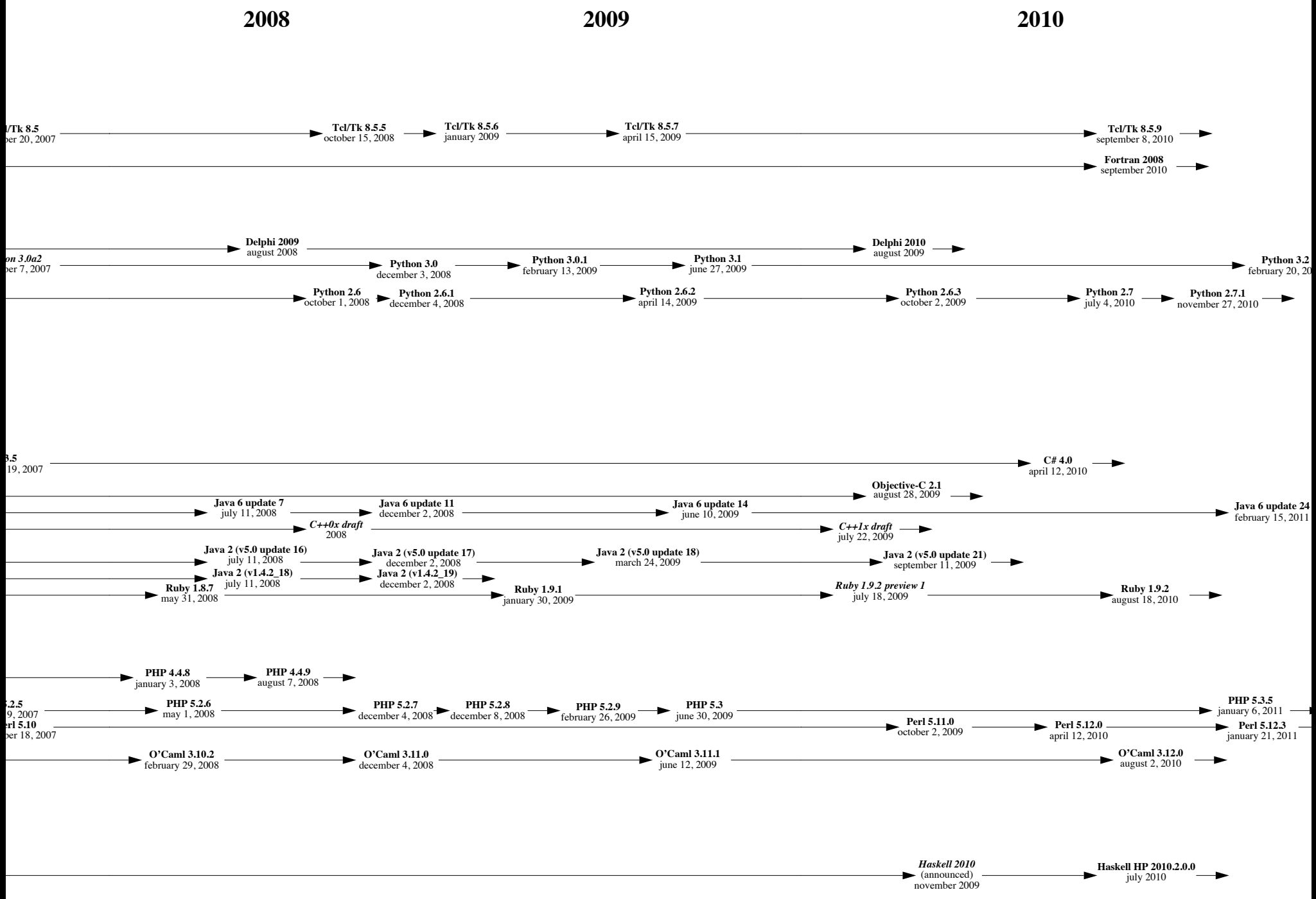
Steve Yegge

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

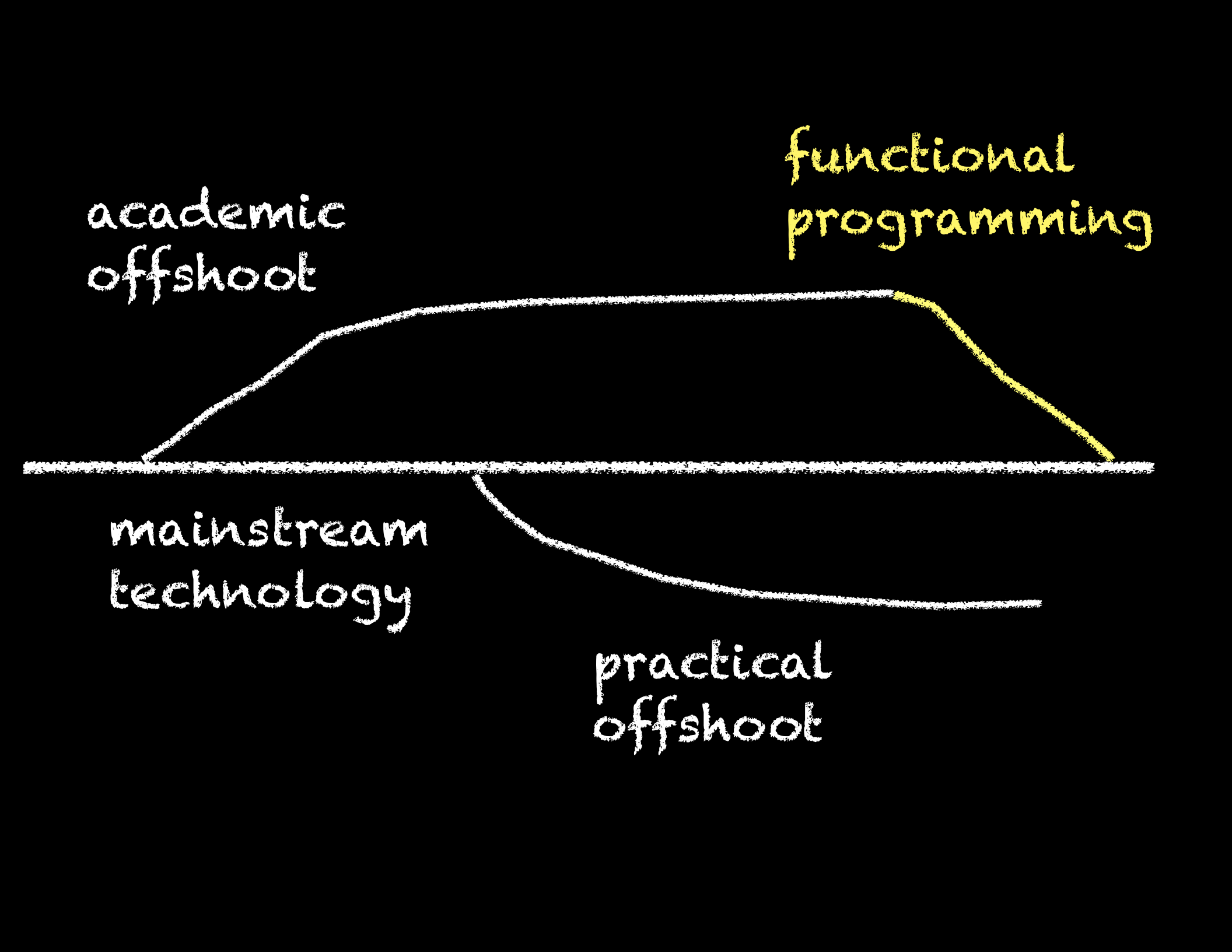

verbs!

UndoManager.execute()

undo()


2008
2009
2010
Tcl/Tk 8.5.5
october 15, 2008
Tcl/Tk 8.5.6
january 2009
Tcl/Tk 8.5.7
april 15, 2009
Tcl/Tk 8.5.9
september 8, 2010
Fortran 2008
september 2010
Delphi 2009
august 2008
Delphi 2010
august 2009
Python 3.0
december 3, 2008
Python 3.0.1
february 13, 2009
Python 3.1
june 27, 2009
Python 3.2
Python 2.6
october 1, 2008
Python 2.6.1
december 4, 2008
Python 2.6.2
april 14, 2009
Python 2.6.3
october 2, 2009
Python 2.7
july 4, 2010
Python 2.7.1
november 27, 2010
C# 4.0
april 12, 2010
Objective-C 2.1
august 28, 2009
Java 6 update 7
july 11, 2008
Java 6 update 11
december 2, 2008
Java 6 update 14
june 10, 2009
Java 6 update 24
february 15, 2011
C++0x draft
2008
C++1x draft
july 22, 2009
Java 2 (v5.0 update 16)
july 11, 2008
Java 2 (v5.0 update 17)
december 2, 2008
Java 2 (v5.0 update 18)
march 24, 2009
Java 2 (v5.0 update 21)
september 11, 2009
Java 2 (v1.4.2_18)
july 11, 2008
Java 2 (v1.4.2_19)
december 2, 2008
Ruby 1.8.7
may 31, 2008
Ruby 1.9.1
january 30, 2009
Ruby 1.9.2 preview 1
july 18, 2009
Ruby 1.9.2
august 18, 2010
PHP 4.4.8
january 3, 2008
PHP 4.4.9
august 7, 2008
PHP 5.2.6
may 1, 2008
PHP 5.2.7
december 4, 2008
PHP 5.2.8
december 8, 2008
PHP 5.2.9
february 26, 2009
PHP 5.3
june 30, 2009
PHP 5.3.5
january 6, 2011
Perl 5.11.0
october 2, 2009
Perl 5.12.0
april 12, 2010
Perl 5.12.3
january 21, 2011
O'Caml 3.10.2
february 29, 2008
O'Caml 3.11.0
december 4, 2008
O'Caml 3.11.1
june 12, 2009
O'Caml 3.12.0
august 2, 2010
Haskell 2010
(announced)
november 2009
Haskell HP 2010.2.0.0
july 2010

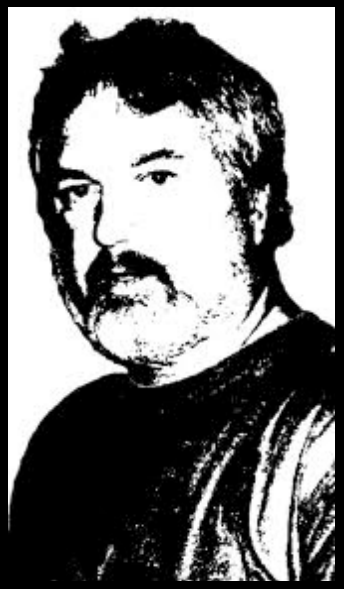

65
1970
1975
1980
Forth
1968
Logo
1968
FORTRAN IV
(Fortran 66 ANS)
1966
LCOMP
1965
MUMPS
1966
Prolog
1970
COBOL 68 ANS
1968
COBOL 74 ANSI
1974
Pascal
1970
PL/M
1972
BCPL
july 1967
B
1969
C
1971
CORAL 66
1966
Simula 67
1967
ALGOL W
1966
ALGOL 68
december
1968
CLU
1974
GOGOL III
1967
Sail
1968
ISWIM
1966
sh
1969
SNOBOL 4
1967
Smalltalk
1971
Smalltalk-72
1972
sed
1973
Smalltalk-74
1974
Smalltalk-76
1976
Mainsail
1975
SASL
1976
MS Basic 2.0
july 1975
Scheme
1975
ML
1973
SL5
1976
Icon
1977
FIG-Forth
1978
FORTRAN V
(Fortran 77 ANSI)
april 1978
MUMPS (ANSI)
september 15, 1977
Modula
1975
PL/1 ANS
1976
Rex 1.00
may 1979
Modula 2
1979
Ada
1979
C (K&R)
1978
Classic C
Mesa
1977
Smalltalk-78
1978
awk
1978
csh
october 1978
Scheme MIT
1978
PostScript
1982
Forth-83
1983
Prolog II
october 1982
Prolog III
1984
Sharp
APL 2
august 19
B
1981
Rex 2.00
1980
Rex 3.00
1982
Rexx 3.20
1984
Pascal AFNOR
1983
Ada 83 ANSI
january 1983
Concurrent C
1984
Objective-C
1983
C with Classes
april 1980
C++
july 1983
Cedar
1983
Smalltalk-80
1980
KRC
1981
Miranda
1982
BASICA
1981
GW-Basic
1983
Common Lisp
1984
Scheme 84
1984
SML
1984

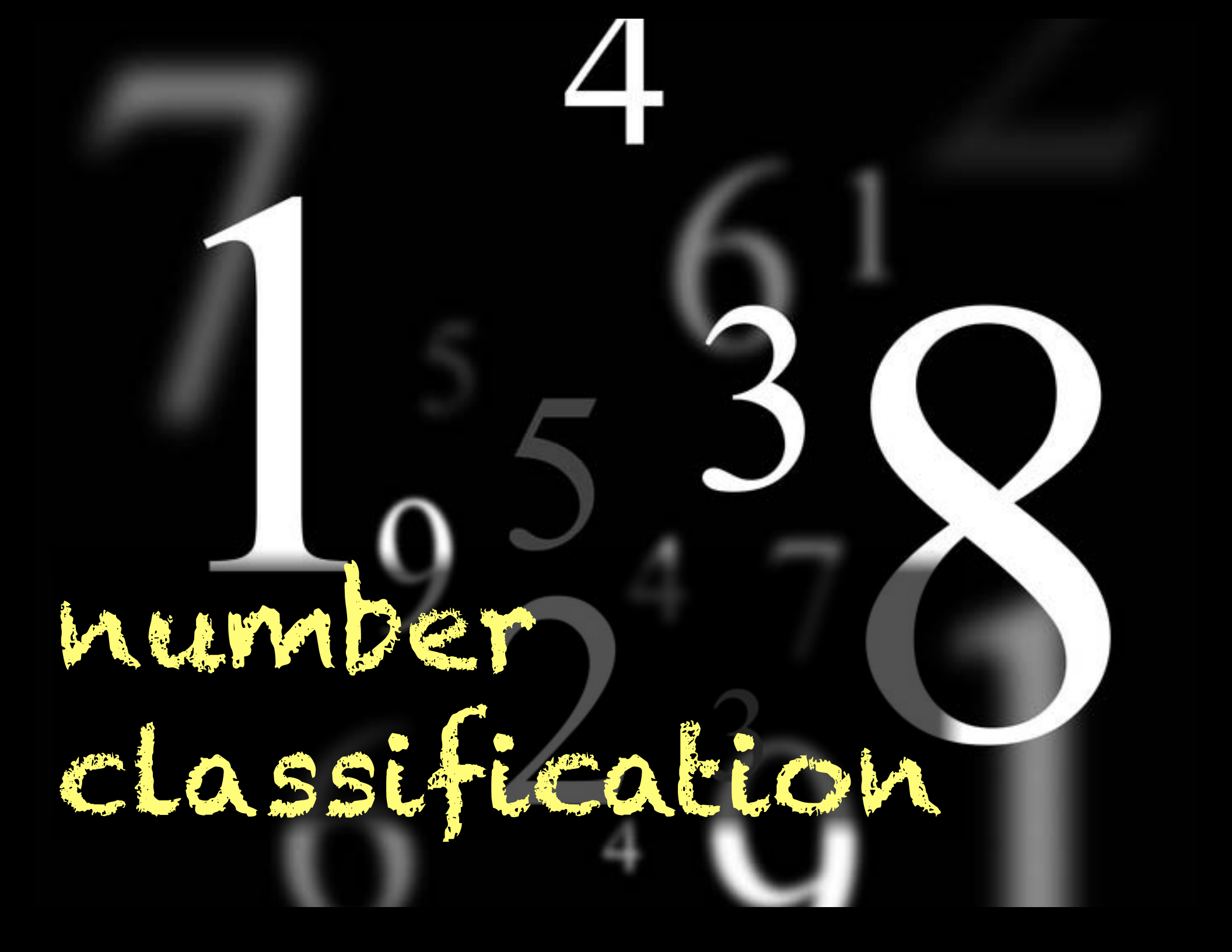

functional
programming
academic
offshoot
mainstream
technology
practical
offshoot

OO makes code understandable by encapsulating moving parts.

FP makes code understandable by minimizing moving parts.

Michael Feathers, author of "Working with Legacy Code"




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

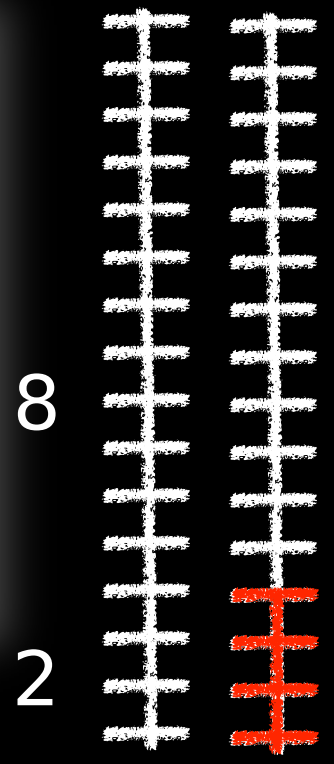
$\Sigma(f(\#)) < 2\#$ deficient

imperative

```
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) {…
}
```

```java
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) {…

    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 functions

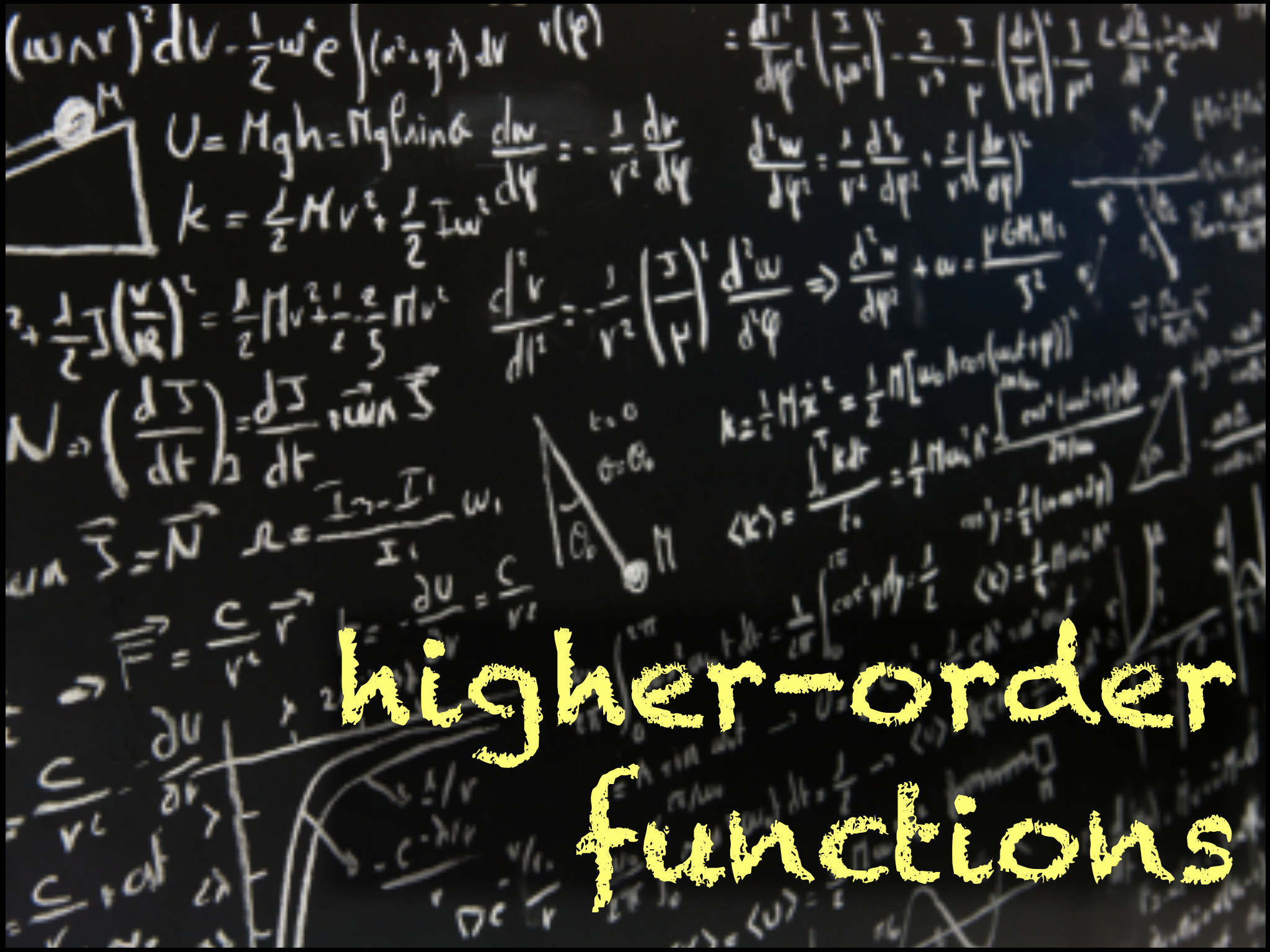
pure functions

concepts

strict evaluation

higher-order functions

recursion


higher-order
functions

higher-order functions

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

```
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());
        }
    });
}
```


```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

```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
```

```java
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

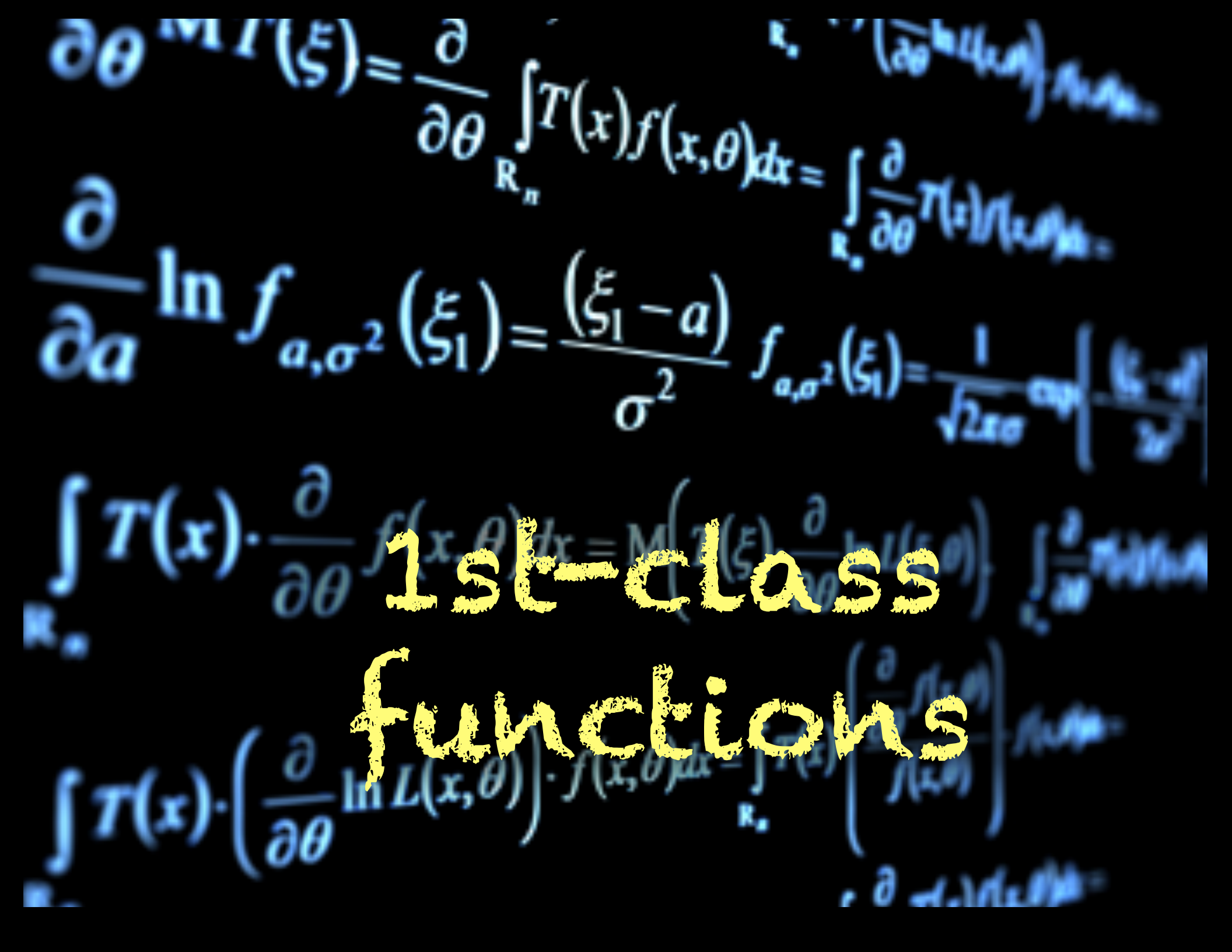
memory allocation

garbage collection

concurrency

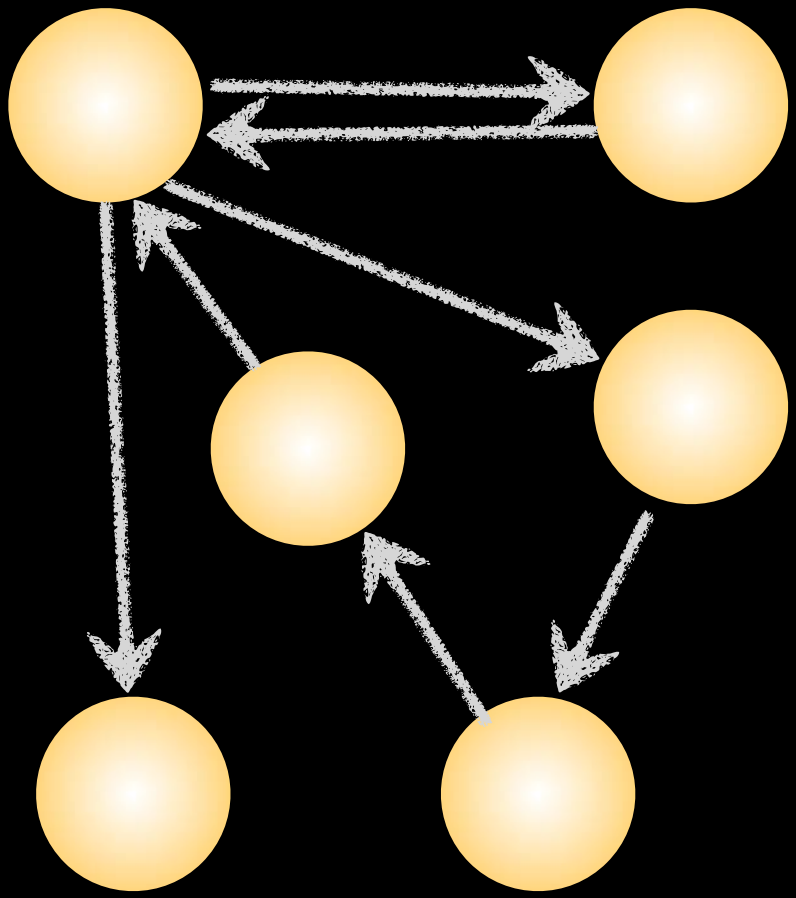
state

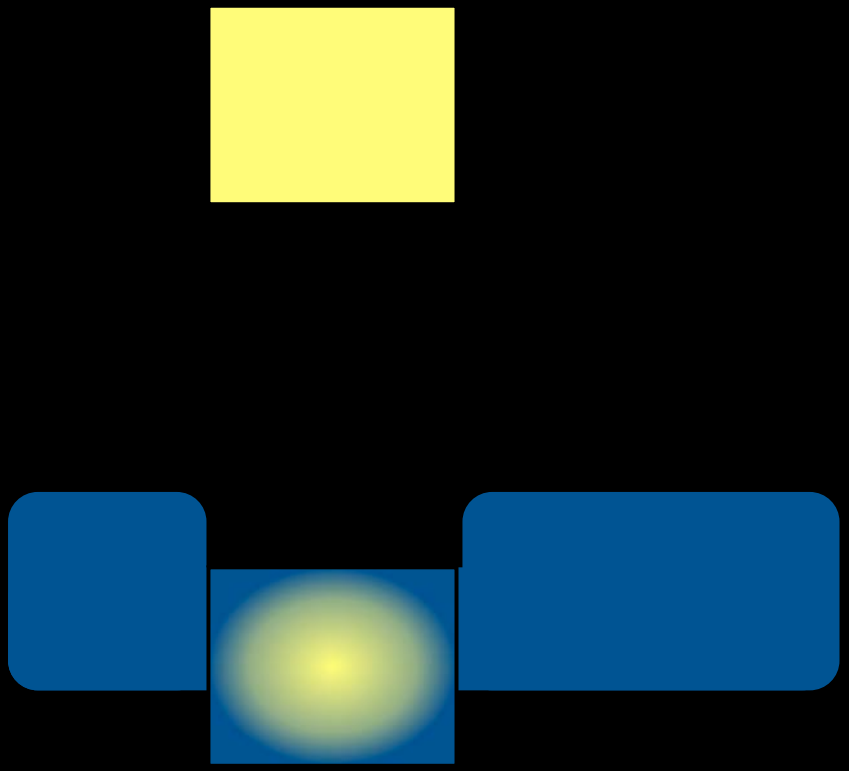
tests  specification-based testing frameworks

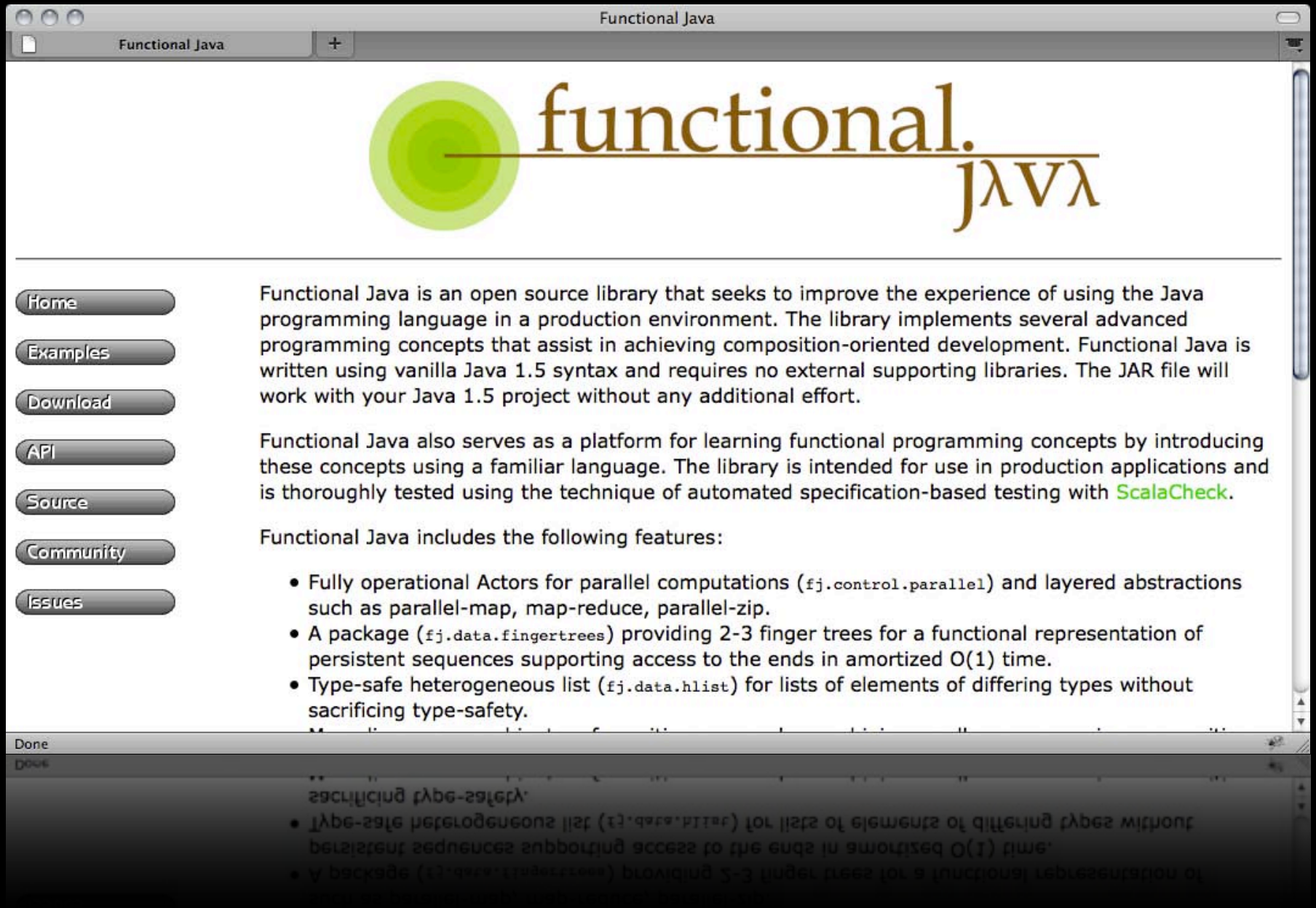

1st-class
functions

1st-class functions

functions can appear anywhere other language constructs can appear



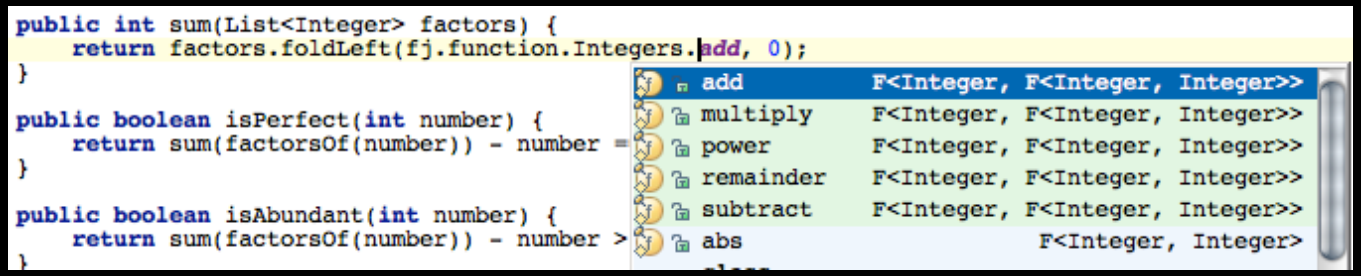



Functional Java
Functional Java
functional.
javλ
Home
Examples
Download
API
Source
Community
Issues
Functional Java is an open source library that seeks to improve the experience of using the Java programming language in a production environment. The library implements several advanced programming concepts 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

```java
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);
}
```



think about results, not steps

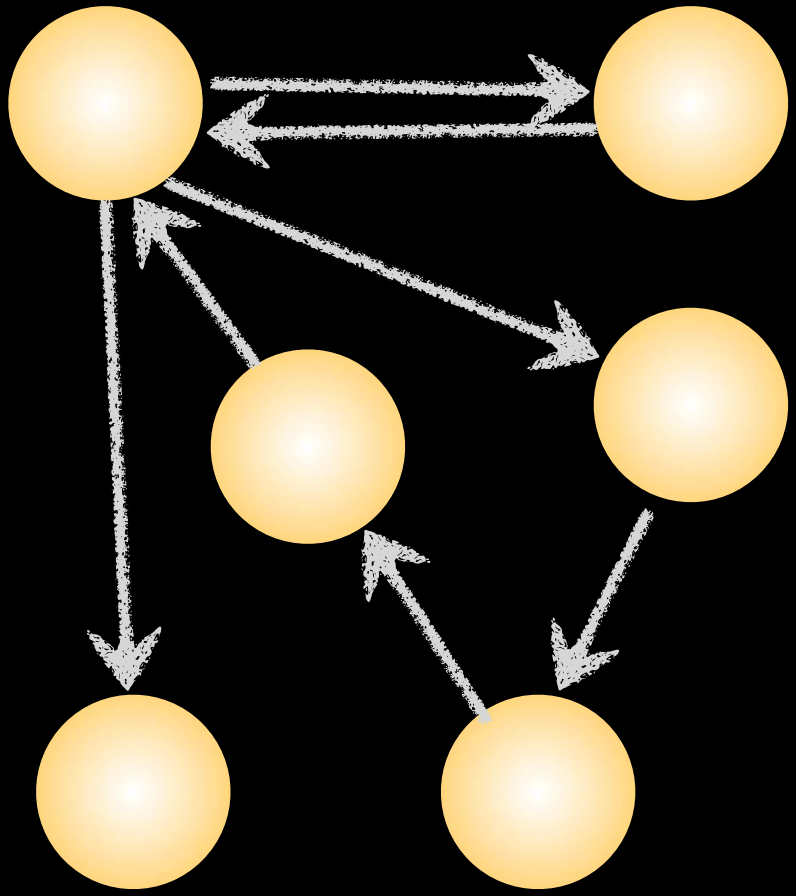
```
public List<Integer> factorsOf(final int number) {
    return range(1, number + 1)
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(number, i);
                }
            });
}
```

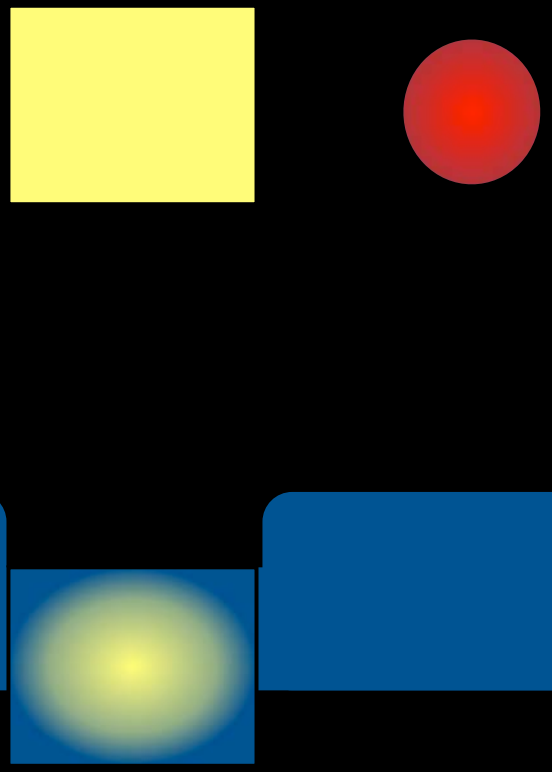
12

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





composition


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

```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)
```

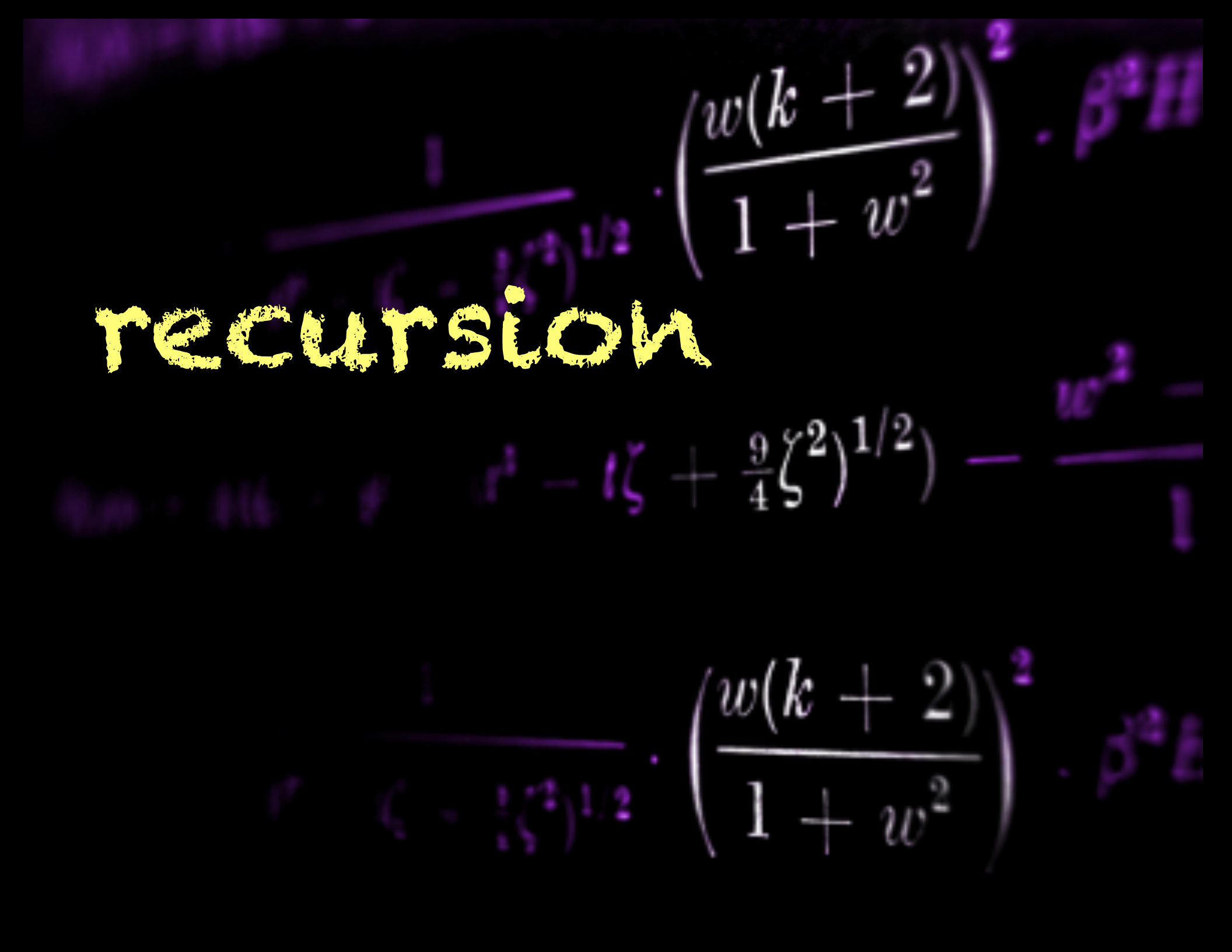
```
def composite = { f, g, x -> return f(g(x))}
def thirtyTwoer = composite.curry(quadrate, octate)
```

new, different tools

currying

```
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)))
}
```

recursion



iterate a List

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

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

iterateList(perfectNumbers)
```

recurse a list

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

recurseList(perfectNumbers)
```

iterative 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)
```

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)
}
```

who's minding the state?

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

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 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

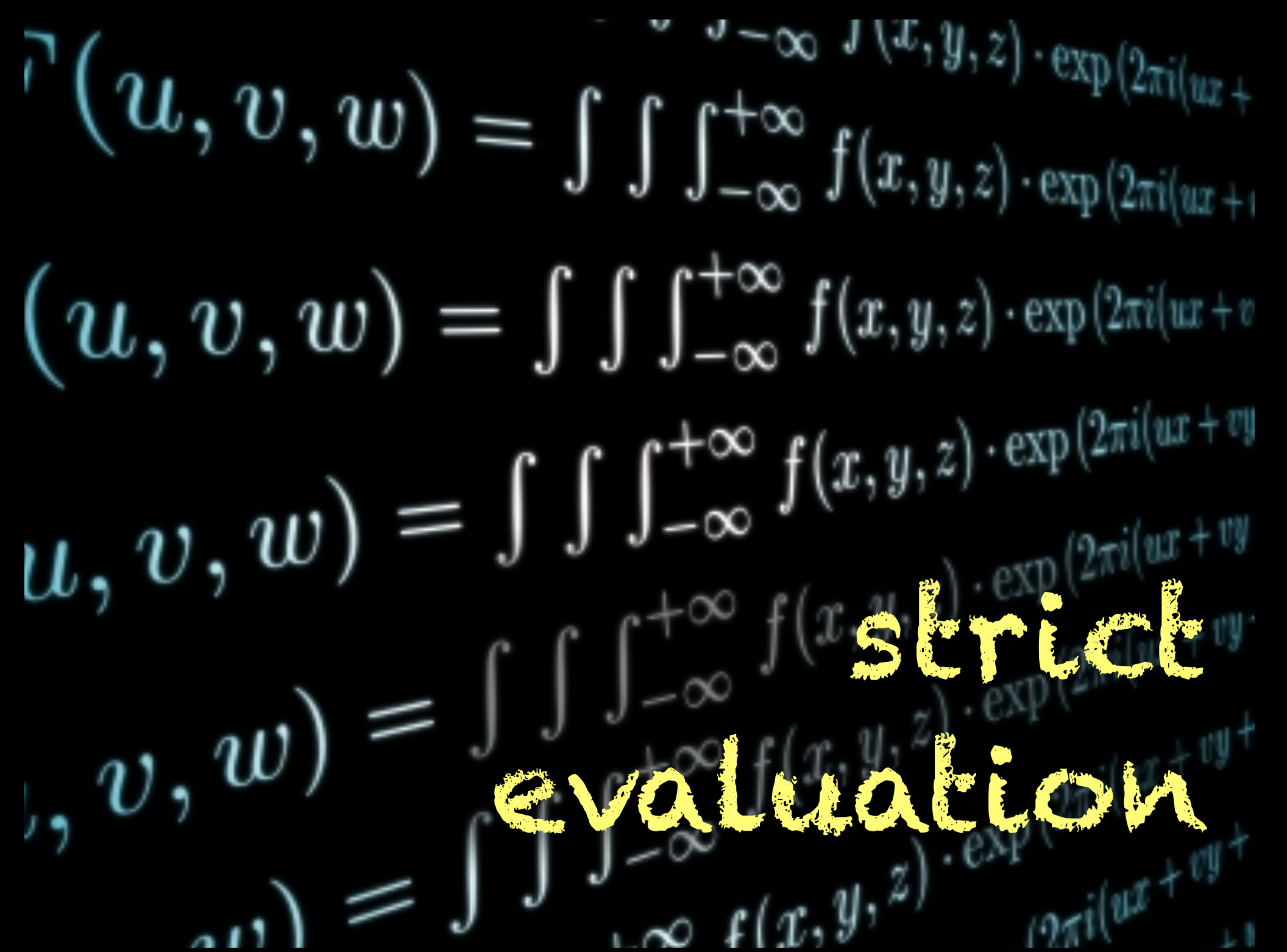
concurrency

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

```clojure
(use '[clojure.contrib.lazy-seqs :only (primes)])

(def ordinals-and-primes
     (map vector (iterate inc 1) primes))

(take 5 (drop 1000 ordinals-and-primes))
```

```
([1001 7927] [1002 7933] [1003 7937] [1004 7949] [1005 7951])
```

new, different tools

concurrency

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

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

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

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

functions:

depend only on their arguments

given the same arguments, return the same values

no effect on the world

no notion of time

most programs are processes

expect change over time

affect the world

wait for external events

produce different answers at different times

what can we add to functional programming to deal with processes?

variables

assume 1 thread of control, 1 timeline

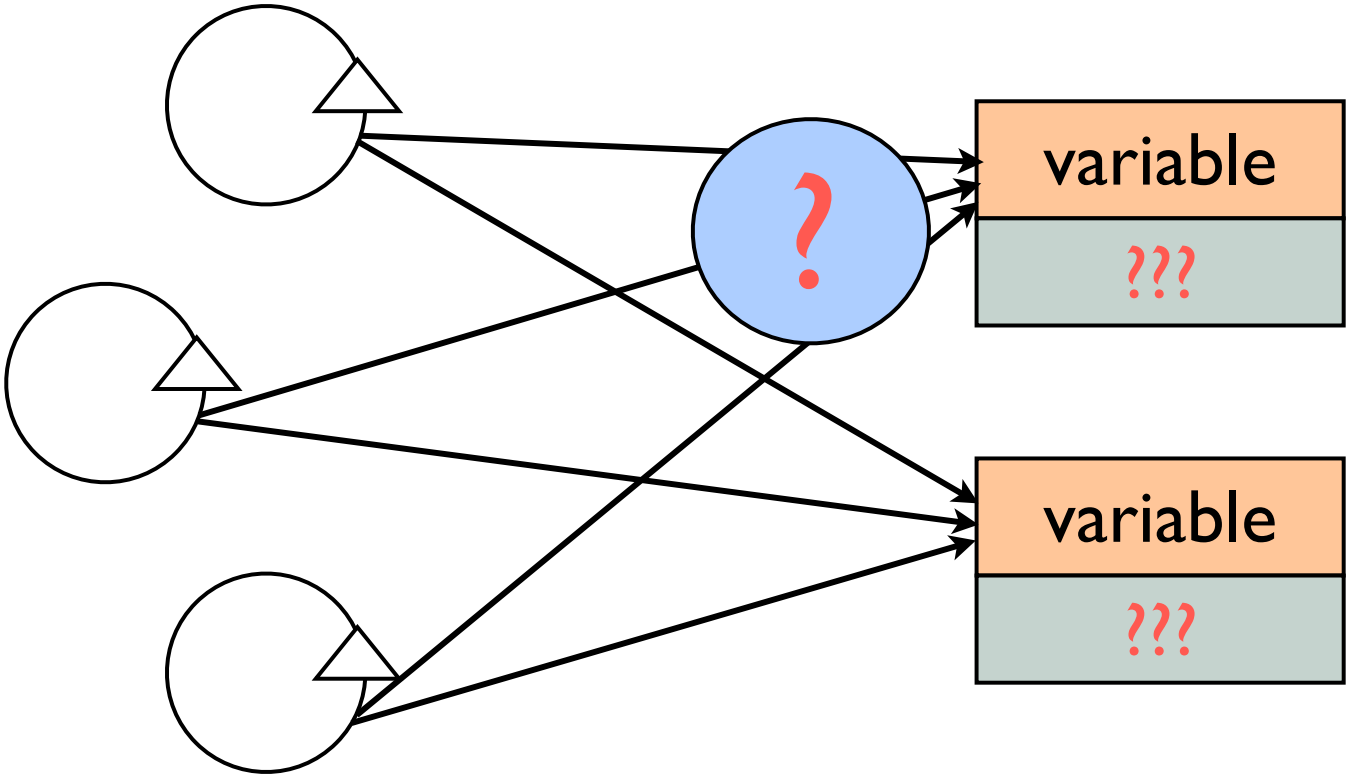
not atomic

non-composable

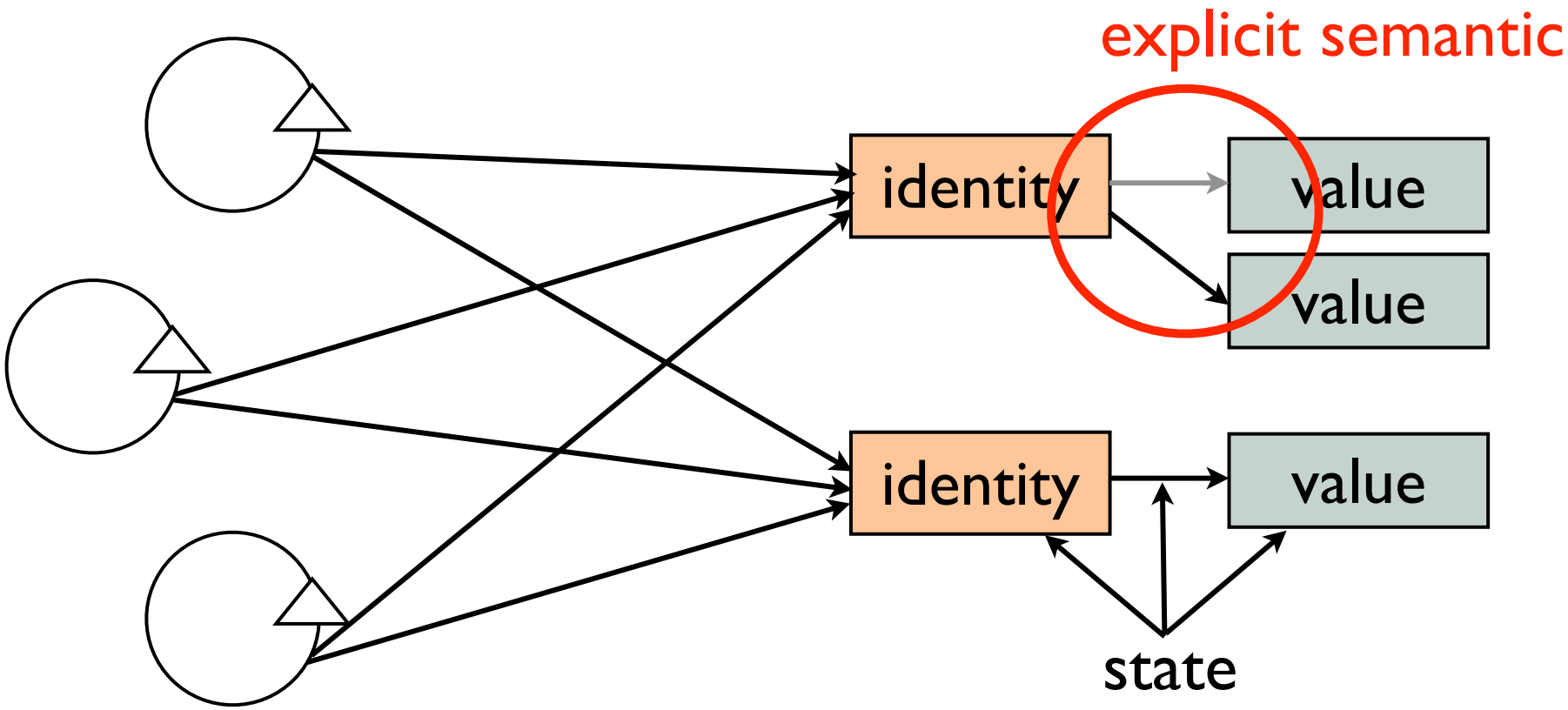
subtle visibility rules

with concurrency: **lock & pray**

Life w/ variables



identity

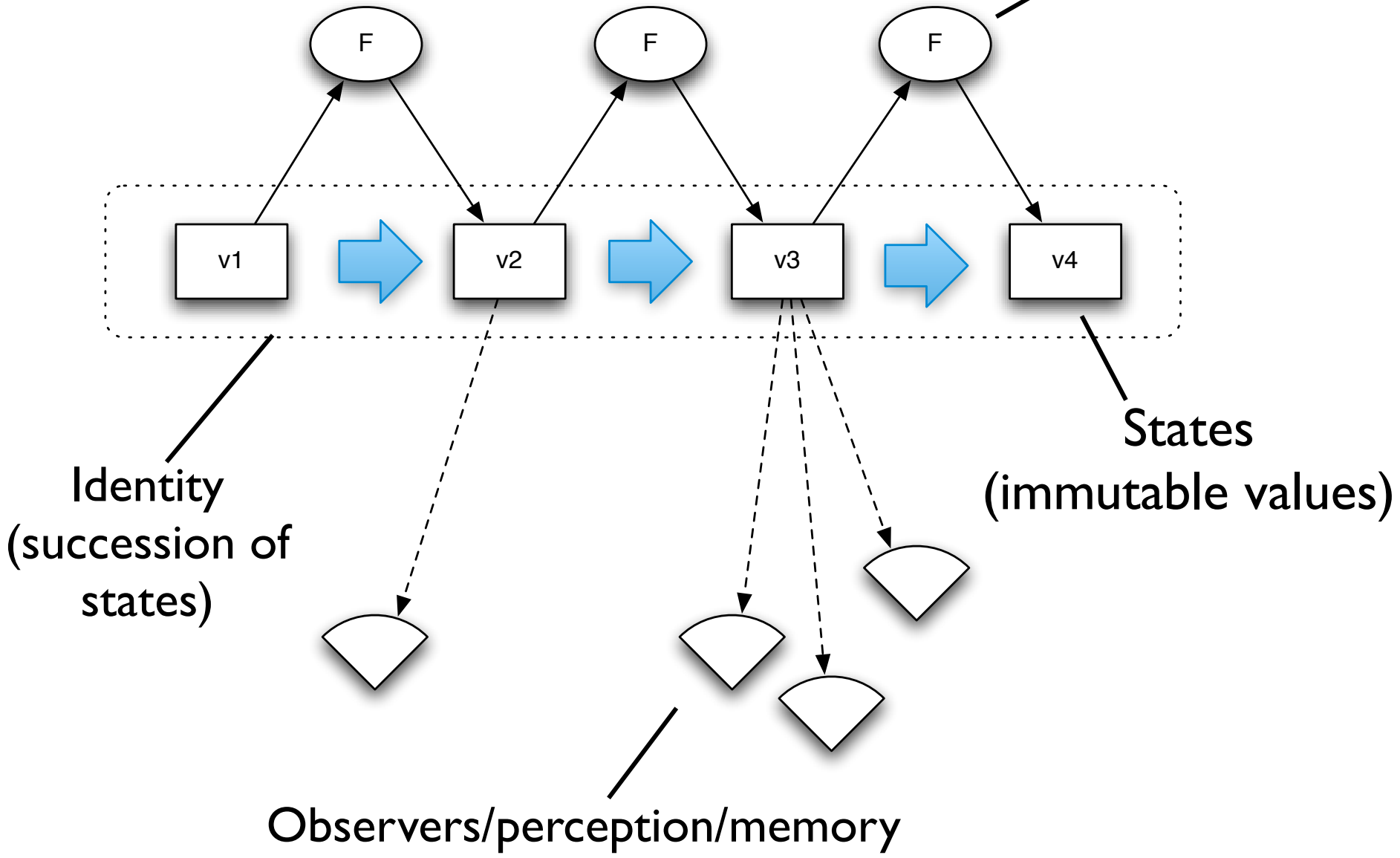

explicit semantic
identity
value
value
identity
value
state

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



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

actors

```scala
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

```java
public final class Address1 {
    private final String name;
    private final List<String> streets;
    private final String city;
    private final String state;
    private final String zip;

    public Address1(String name, List<String> streets, String city, String state, String zip) {
        this.name = name;
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getName() {
        return name;
    }

    public List<String> getStreets() {
        return Collections.unmodifiableList(streets);
    }

    public String getCity() {
        return city;
    }

    public String getState() {
        return state;
    }

    public String getZip() {
        return zip;
    }
}
```

immutable !

ensure the class cannot be overridden

make fields final *

* not necessarily private

all state set in constructor

no mutating methods

defensively copy mutable object fields

```java
public final class Address {
    private final List<String> streets;
    public final String city;
    public final String state;
    public final String zip;

    public Address(List<String> streets, String city, String state, String zip) {
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public final List<String> getStreets() {
        return Collections.unmodifiableList(streets);
    }
}
```

```java
@Test
public void address_access_to_fields_but_enforces_immutability() {
    Address a = new Address(streets("201 E Randolph St",
            "Ste 25"), "Chicago", "IL", "60601");
    assertEquals("Chicago", a.city);
    assertEquals("IL", a.state);
    assertEquals("60601", a.zip);
    assertEquals("201 E Randolph St", a.getStreets().get(0));
    assertEquals("Ste 25", a.getStreets().get(1));
    // compiler disallows
    //a.city = "New York";
    a.getStreets().clear();
}
```

```java
@Test
public void address_access_to_fields_but_enforces_immutability() {
    Address a = new Address(streets("201 E Randolph St",
            "Ste 25"), "Chicago", "IL", "60601");
    assertEquals("Chicago", a.city);
    assertEquals("IL", a.state);
    assertEquals("60601", a.zip);
    assertEquals("201 E Randolph St", a.getStreets().get(0));
    assertEquals("Ste 25", a.getStreets().get(1));
    // compiler disallows
    //a.city = "New York";
    a.getStreets().clear();
}
```

violates Uniform Access Principle

Uniform Access Principle

"All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation."

Bertrand Meyer,
creator of Eiffel programming language

```java
public final class Address {
    private final List<String> streets;
    public final String city;
    public final String state;
    public final String zip;

    public Address(List<String> streets, String city, String state, String zip) {
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public final List<String> getStreets() {
        return Collections.unmodifiableList(streets);
    }
}
```

```
class Address {
    def public final List<String> streets;
    def public final city;
    def public final state;
    def public final zip;

    def Address(streets, city, state, zip) {
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    def getStreets() {
        Collections.unmodifiableList(streets);
    }
}
```

```
class AddressTest {
    def expectedAddr = new Address(
            ["201 E Randolph St", "25th Floor"], "Chicago", "IL", "60601")

    @Test (expected = ReadOnlyPropertyException.class)
    void address_primitives_immutability() {
        assertEquals "Chicago", expectedAddr.city
        expectedAddr.city = "New York"
    }

    @Test (expected = ReadOnlyPropertyException.class)
    void address_list_references() {
        expectedAddr.streets = new ArrayList<String>();
    }

    @Test (expected=UnsupportedOperationException.class)
    void address_list_contents() {
        assertEquals "201 E Randolph St", expectedAddr.streets[0]
        assertEquals "25th Floor", expectedAddr.streets[1]
        expectedAddr.streets[0] = "404 W Randoph St"
    }
}
```

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

declarative
over
imperative

functional.
JAVA

paradigm over tool

Clojure

summary

functional thinking

new ways of thinking about design

new tools for extension, reuse, etc.

immediately beneficial beginning steps

following the general trend in language design

enables entirely new capabilities

?'s

please fill out the session evaluations

NEAL FORD software architect / meme wrangler

ThoughtWorks®

nford@thoughtworks.com
3003 Summit Boulevard, Atlanta, GA 30319
www.nealford.com
www.thoughtworks.com
blog: memeagora.blogspot.com
twitter: neal4d