Functional Thinking
tortured

3^metaphors
assign to x

x := x + 1

solve for x?
Paradigms 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

verbs!
“OOP makes working with state easier. FP makes eliminating state easier.”
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 

\[ \sum(f(#)) - # = # \]

(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

\[ \sum(f(#)) = 2\# \]  
perfect

\[ \sum(f(#)) > 2\# \]  
abundant

\[ \sum(f(#)) < 2\# \]  
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) {...
}
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() {...

    private ...

    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) {...
}
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) {
    ...
}
public class NumberClassifier {

    public boolean isFactor(int number, int potential_factor) {
        // Implementation...
    }

    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) {
        // Implementation
    }

    public Set<Integer> factors(int number) {
        // Implementation
    }

    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;
    }
}
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
concepts

1st class functions

pure functions

strict evaluation

high-order functions

recursion
high-order functions
high-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());
        }
    });
}
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()
    }
}
UndoManager.execute()
def addOrderFrom(cart, userName, order) {
    wrapInTransaction {
        add order, userKeyBasedOn(userName)
        addLineItemsFrom cart, order.getOrderKey()
    }
}

What's so special about...
closures
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
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 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 (`functional-actors`) and layered abstractions such as parallel-map, map-reduce, parallel-zip.
- A package (`functional.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 (`functional.data.hlist`) for lists of elements of differing types without sacrificing type-safety.
- Many more libraries of functionality to be integrated with the core framework.
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 number % i == 0;
            }
        });
    }

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

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

public boolean isPerfect(int number) {
    return sum(factorsOf(number)) - number == 0;
}

public boolean isAbundant(int number) {
    return sum(factorsOf(number)) - number > 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 number % i == 0;
        }
    });
}
public boolean isFactor(int number, int potential_factor) {
    return number % potential_factor == 0;
}

general List<Integer> factors0f(final int number) {
    return range(1, number+1).filter(new F<Integer, Boolean>() {
        public Boolean f(final Integer i) {
            return number % i == 0;
        }
    });
}

general int sum(List<Integer> factors) {
    return factors.fold(0, function. integer. add);
list comprehension

```
(defn factors [number]
  (set (for [n (range 1 (inc number))]
        :when (is-factor? n number) n)))
```

return the list as a set

for each n in range from 1 to (number + 1)

filter list by criteria my is-factor? function

return the numbers that match
(defn is-factor? [factor number]
  (= 0 (rem number factor)))

(defn factors [number]
  (set (for [n (range 1 (inc number))
     :when (is-factor? n number)] n)))

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

(defn perfect? [number]
  (= number (- (sum-factors number) number))))
composition
academia alert!
currying

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

then: $\text{curry}(f): X \rightarrow (Y \rightarrow Z)$

currying takes a function with a particular number of parameters and returns a function with some of the parameter values fixed, creating a new function
def product = { x, y ->
    return x * y
}

return a version that always multiplies by 4

def quadrator = product.curry(4)

==

def quadrator_ = { 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)}"
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)))
}
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
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

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)))
}

http://www.scala-lang.org/node/135
think about results, not steps

what about things you want to control?

performance?

new, different tools
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 factors

```java
public List<Integer> factorsOfOptimzied(final int number) {
    List<Integer> factors = range(1, (int) round(sqrt(number))+1))
        .filter(new F<Integer, Boolean>() {
            public Boolean f(final Integer i) {
                return number % i == 0;
            }
        });
    return factors.append(factors.map(new F<Integer, Integer>() {
        public Integer f(final Integer i) {
            return number / i;
        }
    }));
}
```

think about results, not steps
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


sound familiar?
languages handle

garbage collection

concurrency

state

tests

iteration

...
strict evaluation
academia alert!
strict evaluation

`divByZero`

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

non-strict evaluation

all elements pre-evaluated

elements evaluated as needed
(use '[clojure.contrib.lazy-seqs :only (primes)])

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

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

(new, different tools)
concurrency

\[ \oint_{D} dA = \int_{V} \rho dV = Q \]

\[ \oint_{E} dl = -\frac{d}{dt} \int_{A} B dA \]

\[ \oint_{B} dA = 0 \]

\[ \oint_{H} dl = \int_{A} F dA + \frac{d}{dt} \int_{A} D dA \]
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
<table>
<thead>
<tr>
<th>term</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>immutable data in a persistent data structure</td>
</tr>
<tr>
<td>identity</td>
<td>series of causally related values over time</td>
</tr>
<tr>
<td>state</td>
<td>identity at a point in time</td>
</tr>
<tr>
<td>time</td>
<td>relative: before/simultaneous/after ordering of causal values</td>
</tr>
</tbody>
</table>
\[ \delta_{ijt} = \left( \frac{v_{ijt}}{\sum_{i=1}^{n} v_{ijt}} \right) \times 100 - \left( \frac{r_{ijt}}{\sum_{i=1}^{n} r_{ijt}} \right) \times 100 \]

\[ \sigma(\delta_{ij}) \]
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
}
immutability over state transitions

results over steps
composition over structure
declarative over imperative
paradigm over tool
summary

\[ E = mc^2 \]
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
please fill out the session evaluations