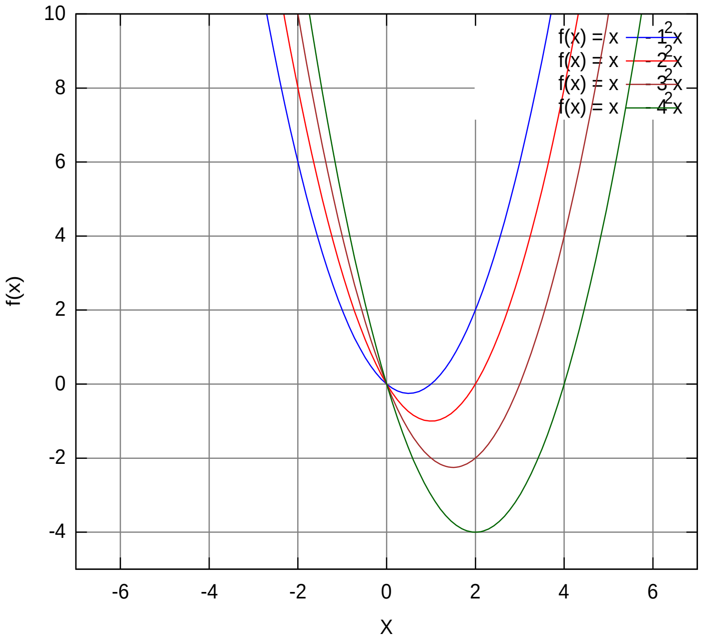
Why we cannot ignore functional programming



Massimiliano Dessì & Mario Fusco

@desmax74 @mariofusco

speakers

Max

JugSardegna, SpringFramework UG, Google Technology UG Sardegna,

Senior Software Engineer in Energeya

Author of Spring 2.5 AOP

Mario

Creator of lambdaj and hammurabi projects

Senior Software Engineer in

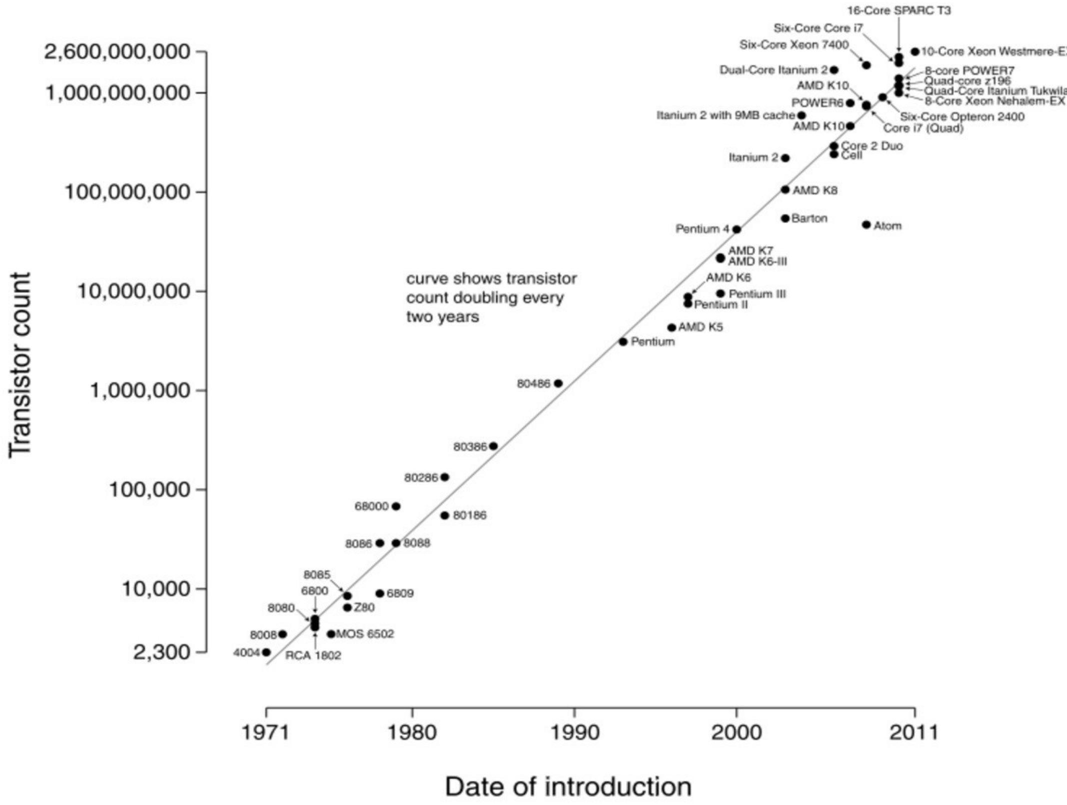
Red Hat working on the core

of the drools rule engine

Moore's law

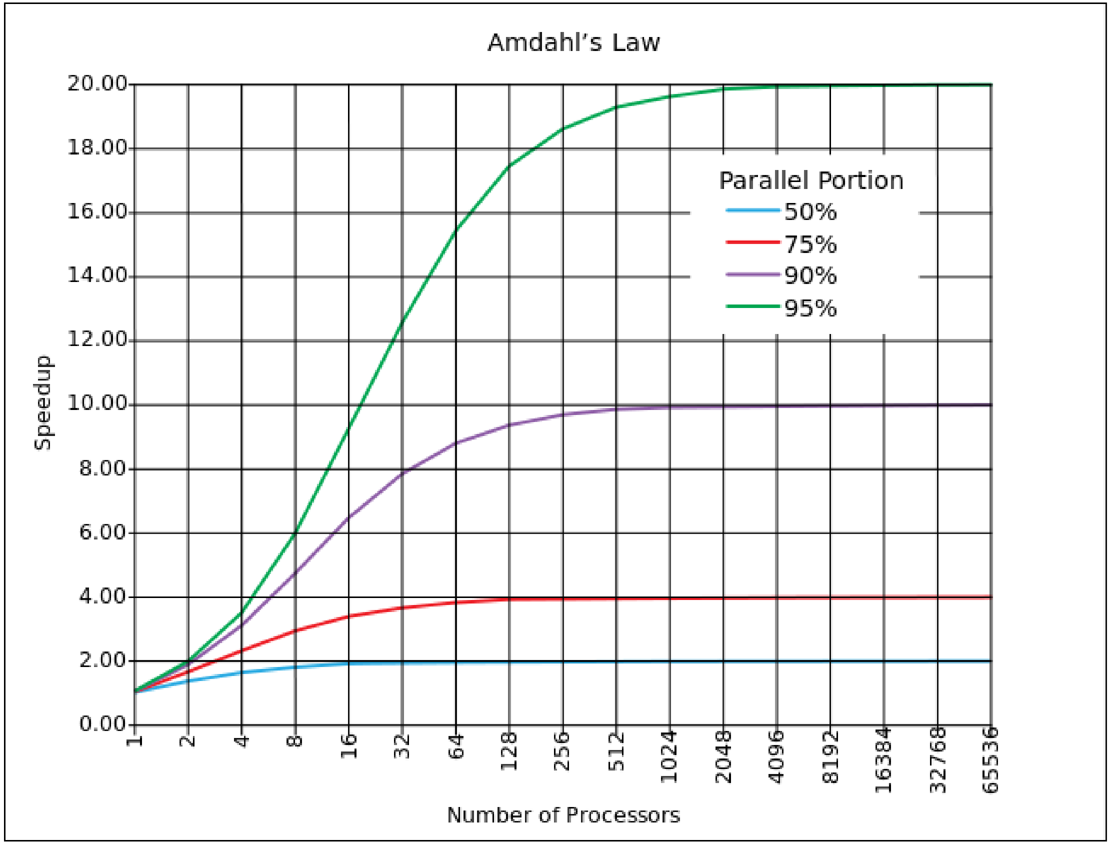
The number of transistors on integrated circuits doubles approximately every two years

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Amdahl's law

The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program



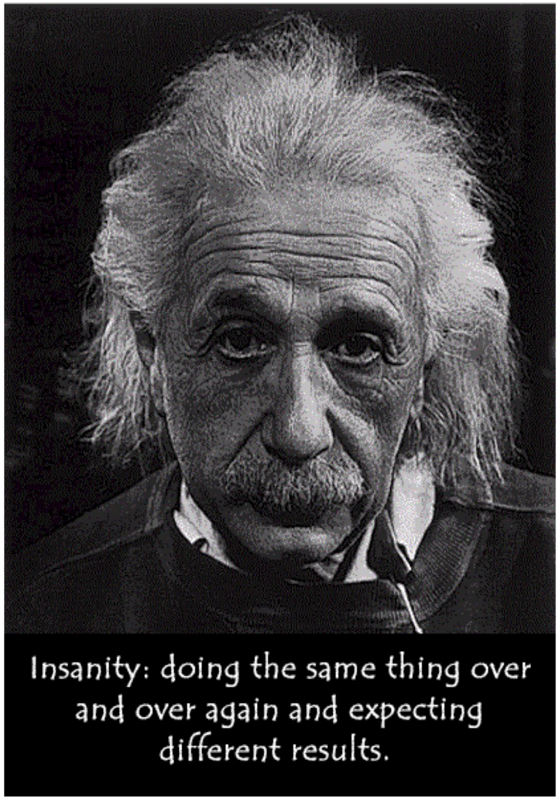
Concurrent programming

Managing concurrent requests

Parallel programming

Running multiple tasks at the same time

The root of the problem


Insanity: doing the same thing over and over again and expecting different results.

~~Mutable state~~ +
Parallel processing =
Non-determinism

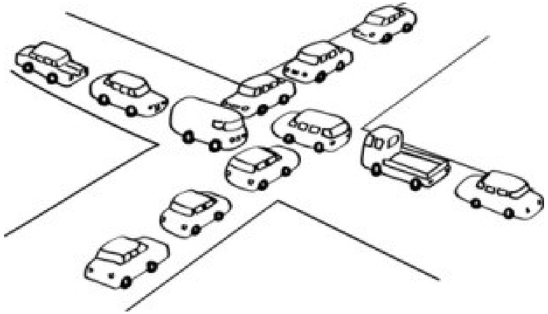
Functional Programming

… and its effects

Race conditions

Starvation

Deadlocks



Livelocks

Epic data race failure

Data races is not a multiuser system's feature,
the Therac-25 was a radiation therapy machine

http://en.wikipedia.org/wiki/Therac-25

It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation, approximately 100 times the intended dose.
These accidents highlighted the dangers of software control of safety-critical systems.

Therac-25 killed three patients and injured several others

Epic data race failure

http://en.wikipedia.org/wiki/Northeast_blackout_of_2003

The blackout affected an estimated 10 million people in Ontario and 45 million people in eight U.S. States.

A software bug known as a race condition existed in General Electric Energy's Unix-based XA/21 energy management system. Once triggered, the bug stalled FirstEnergy's control room alarm system for over an hour.
On Thursday, August 14, 2003, just before 4:10 p.m. While some power was restored by 11 p.m.,
many did not get power back until two days later

Java concurrency model

Threads

Semaphores

Locks

Synchronization

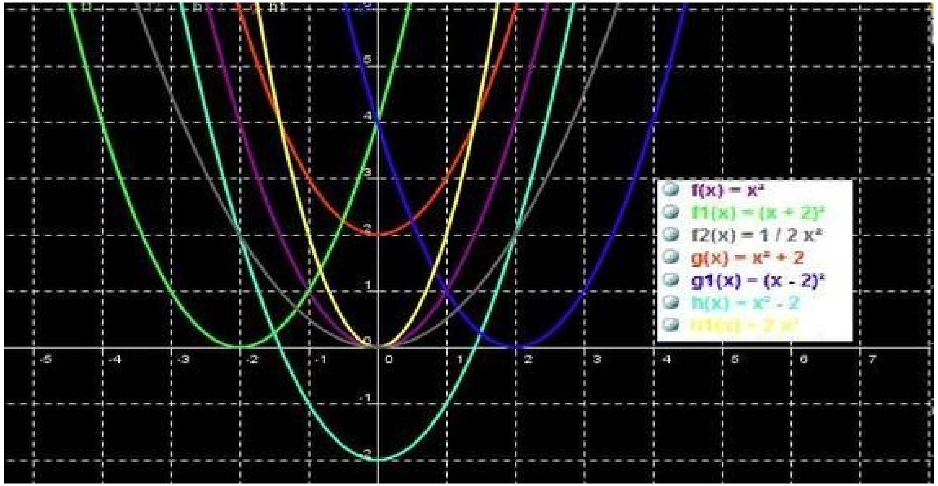
They are sometimes plain evil …

… and sometimes a necessary pain …

… but always the **wrong default**

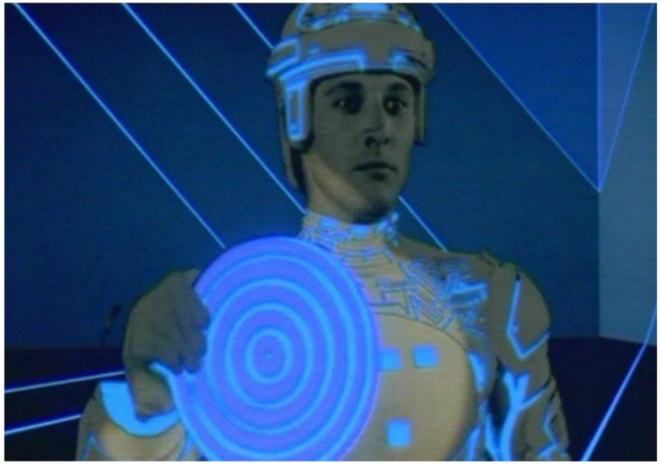
Different concurrency models

Shared mutable state
(threads + locks)



Purely immutable (pure functions)

Isolated mutable state (actors)



Summing attendants ages

Threads

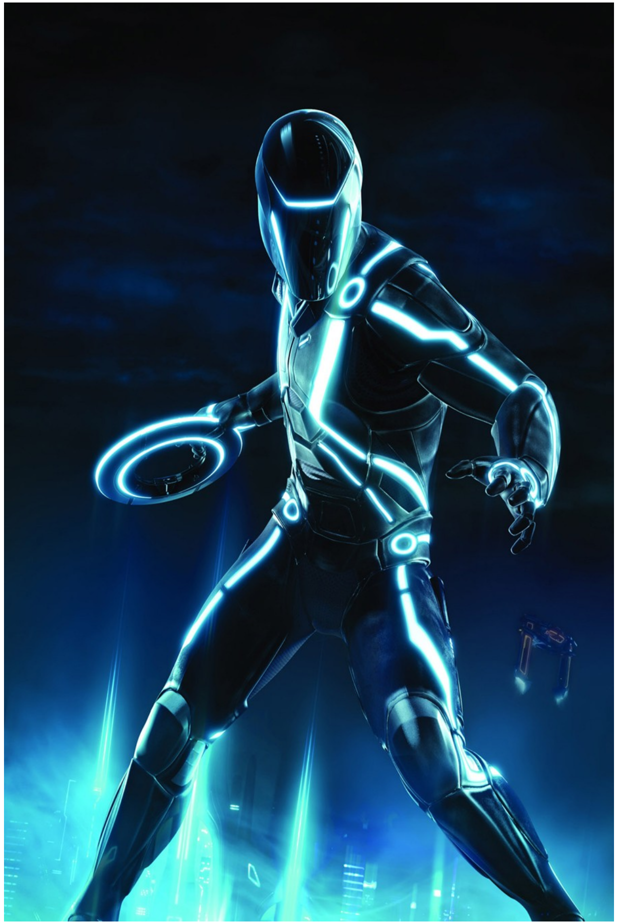
```
class Blackboard {
    int sum = 0;
    int read() { return sum; }
    void write(int value) {
       sum = value; }
    }
}
```

```
class Attendant implements Runnable {
    int age;
    Blackboard blackboard;

    public void run() {
        synchronized(blackboard) {
            int oldSum = blackboard.read();
            int newSum = oldSum + age;
            blackboard.write(newSum);
        }
    }
}
```

Summing attendants ages

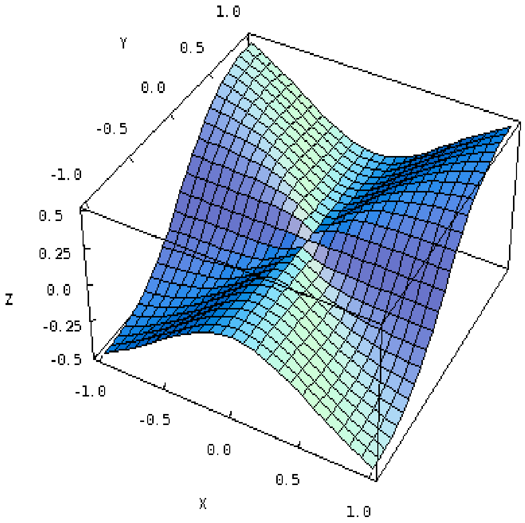
Actors

```
class Blackboard extends UntypedActors {
    int sum = 0;
    public void onReceive(Object message) {
        if (message instanceof Integer) {
            sum += (Integer)message;
        }
    }
}

   class Attendant {
       int age;
       Blackboard blackboard;

       public void sendAge() {
           blackboard.sendOneWay(age);
       }
   }
```



Summing attendants ages

functional



```
class Blackboard {
    final int sum;
    Blackboard(int sum) { this.sum = sum; }
}
class Attendant {
    int age;
    Attendant next;

    public Blackboard addMyAge(Blackboard blackboard) {
        final Blackboard b = new Blackboard(blackboard.sum + age);
        return next == null ? b : next.myAge(b);
    }
}

attendants.foldLeft(
     new Blackboard(0),(att, b) -> new Blackboard(b.sum + att.age)
);
```

NO

Avoidable Side effects

Reassigning a variable

Modifying a data structure in place

Setting a field on an object

Throwing an exception or halting with an error

Deferrable Side effects

Printing to the console

Reading user input

Reading from or writing to a file

Drawing on the screen

functional program

Functional programming
is a restriction on
how
we write programs, but not on
what
they can do

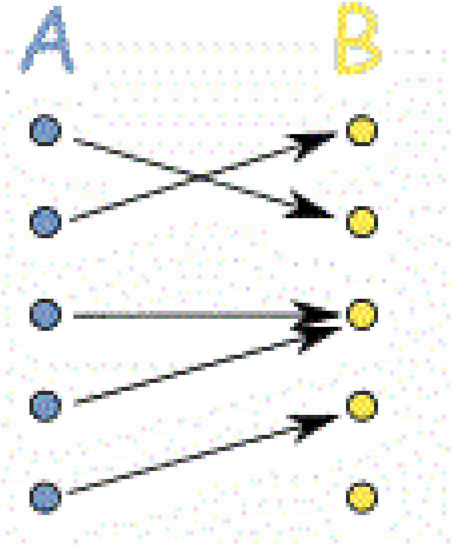
The OOP/FP dualism

```
class Bird
class Cat {
  def capture(b: Bird): Unit = ...
  def eat(): Unit = ...
}
val cat = new Cat
val bird = new Bird

cat.capture(bird)
cat.eat()
```

```
class Cat
class Bird
trait Catch
trait FullStomach
def capture(prey: Bird, hunter: Cat): Cat with Catch
def eat(consumer: Cat with Catch): Cat with FullStomach

val story = (capture _) andThen (eat _)
story(new Bird, new Cat)
```

Pure function

A function with input type A and output type B is a computation which relates every value a of type A to exactly one value b of type B such that b is determined solely by the value of a



Pure function

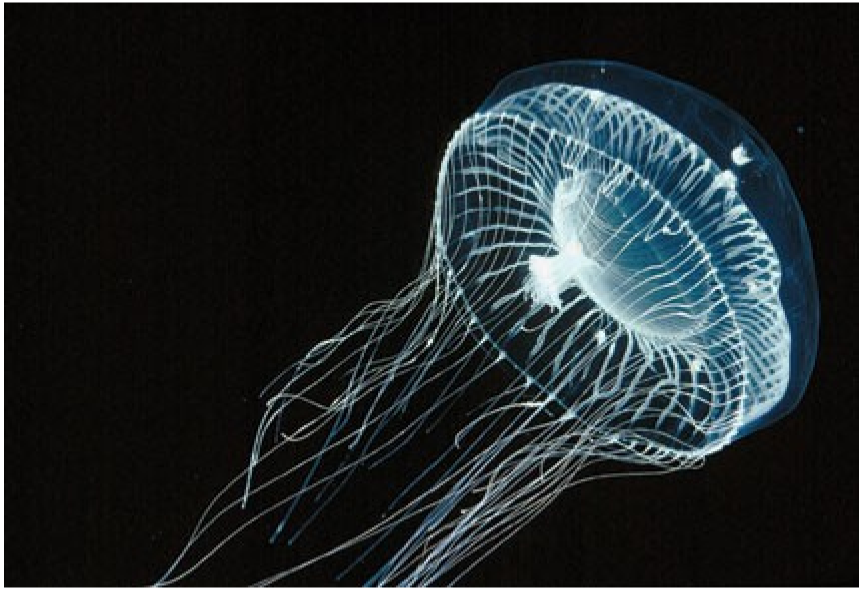
But, if it really
is a function,
it will
do nothing
else

Referential transparency

An expression **e**
is *referentially transparent*
if for all programs **p**,
all occurrences of **e** in **p**
can be
replaced
by the result of evaluating **e**, without
affecting
the observable behavior of **p**

Referential transparency

A function **f**
is *pure*
if the expression **f(x)**
is referentially transparent
for all
referentially transparent **x**



Referential transparency

```
Referencial transparency
String x = "purple";
String r1 = x.replace('p', 't');
String r2 = x.replace('p', 't');

String r1 = "purple".replace('p', 't'); r1: "turtle"
String r2 = "purple".replace('p', 't'); r2: "turtle"
```

```
Not Referencial transparency
StringBuilder x = new StringBuilder("Hi");
StringBuilder y = x.append(", mom");
String r1 = y.toString();
String r2 = y.toString();

String r1 = x.append(", mom").toString();  r1: "Hi, mom"
String r2 = x.append(", mom").toString();  r1: "Hi, mom, mom"
```

Referential transparency

RT Wins

Under a **developer** point of view:

Easier to reason about
since
effects of evaluation are purely local

Use of the *substitution model*: it's possible
to replace
a term with an equivalent one

Referential transparency

RT Wins

Under a **performance** point of view:

The JVM is free to optimize the code by safely reordering the instructions

No need to synchronize access to shared data

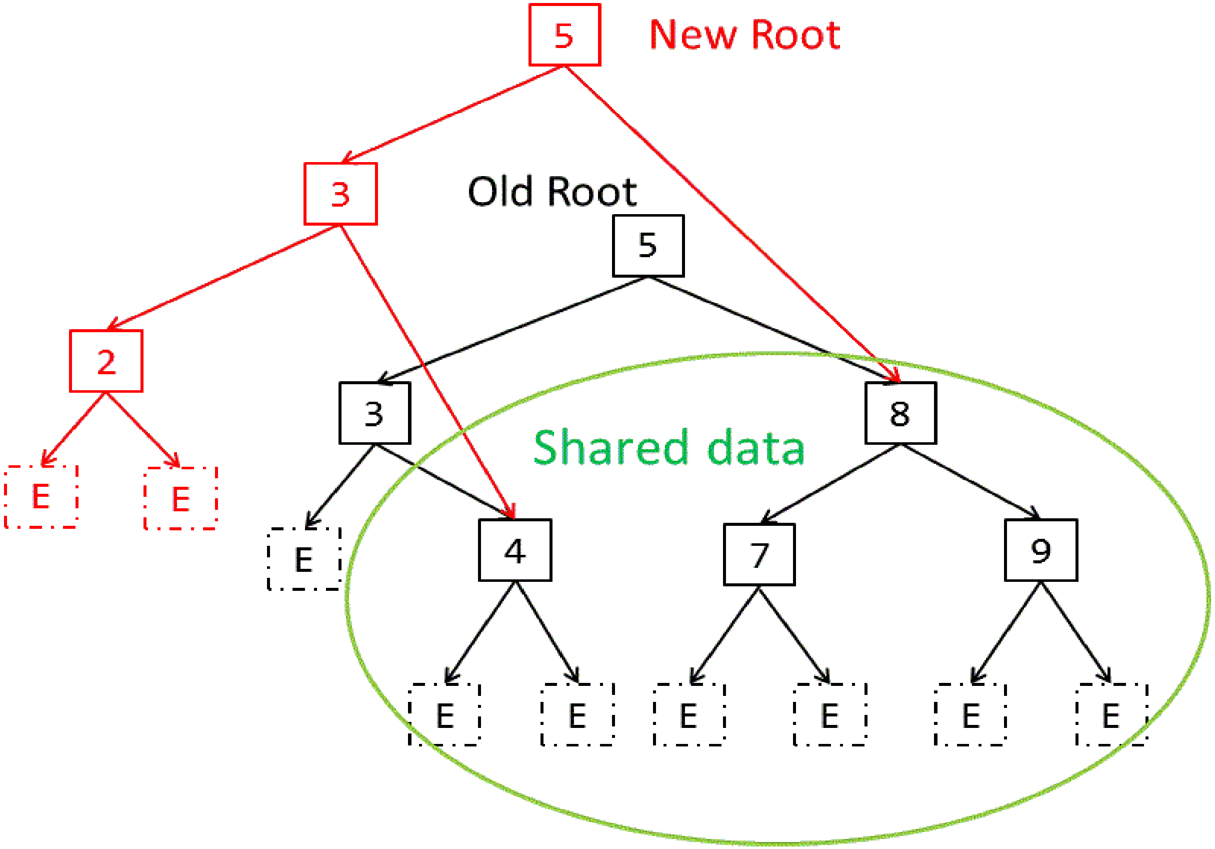
Immutability

Immutable objects
can be shared
among many threads exactly
because none of them can
modify it

Immutability

In the same way
immutable (persistent)
data structures
can be shared
without
any need to synchronize the different
threads accessing them

Persistent Collections



Persistent Collections

Tree with high branching factor (at least 32 per node)
reduce the time for operations on the tries.
High branching factor require four levels to hold up a million of elements.

Phil Bagwell (EPFL) Rich Hickey (Clojure)

http://lampwww.epfl.ch/papers/idealhashtrees.pdf

http://infoscience.epfl.ch/record/169879/files/RMTrees.pdf

http://infoscience.epfl.ch/record/169879/files/RMTrees.pdf?version=1

Modularity

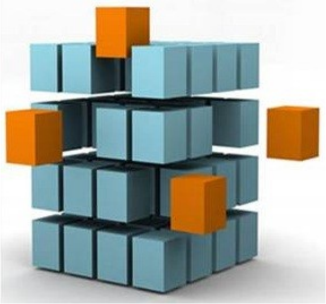
```
class Player { String name; int score; }
```

~~public void declareWinner(Player p) {~~
    ~~System.out.println(p.name + " wins!");~~
~~}~~

```
public void winner(Player p1, Player p2) {
    if (p1.score > p2.score) declareWinner(p1)
    else declareWinner(p2);
}
```

Separate computational logic from side effects

```
public Player maxScore(Player p1, Player p2) {
    return p1.score > p2.score ? p1 : p2;
}
public void winner(Player p1, Player p2) {
    declareWinner(maxScore(p1, p2));
}
```

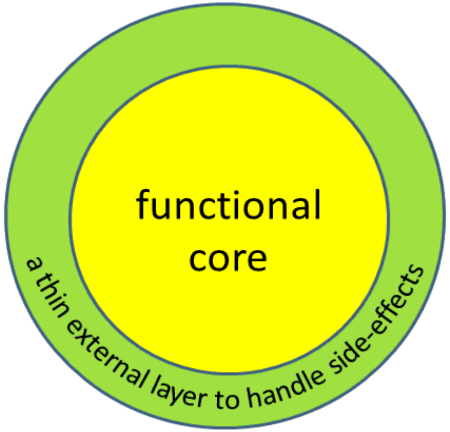


```
declareWinner(players.reduceLeft(maxScore))
```

reuse maxScore to compute the winner among a list of players

Functional core

a thin external layer to handle side-effects



Any function with side-effects can be split into a pure function at the core and a pair of functions with side-effect. This transformation can be repeated to push side-effects to the outer layers of the program.

Q & A

Thank you for your attention !