JVM Languages & Scala at IDC

Increasing Productivity

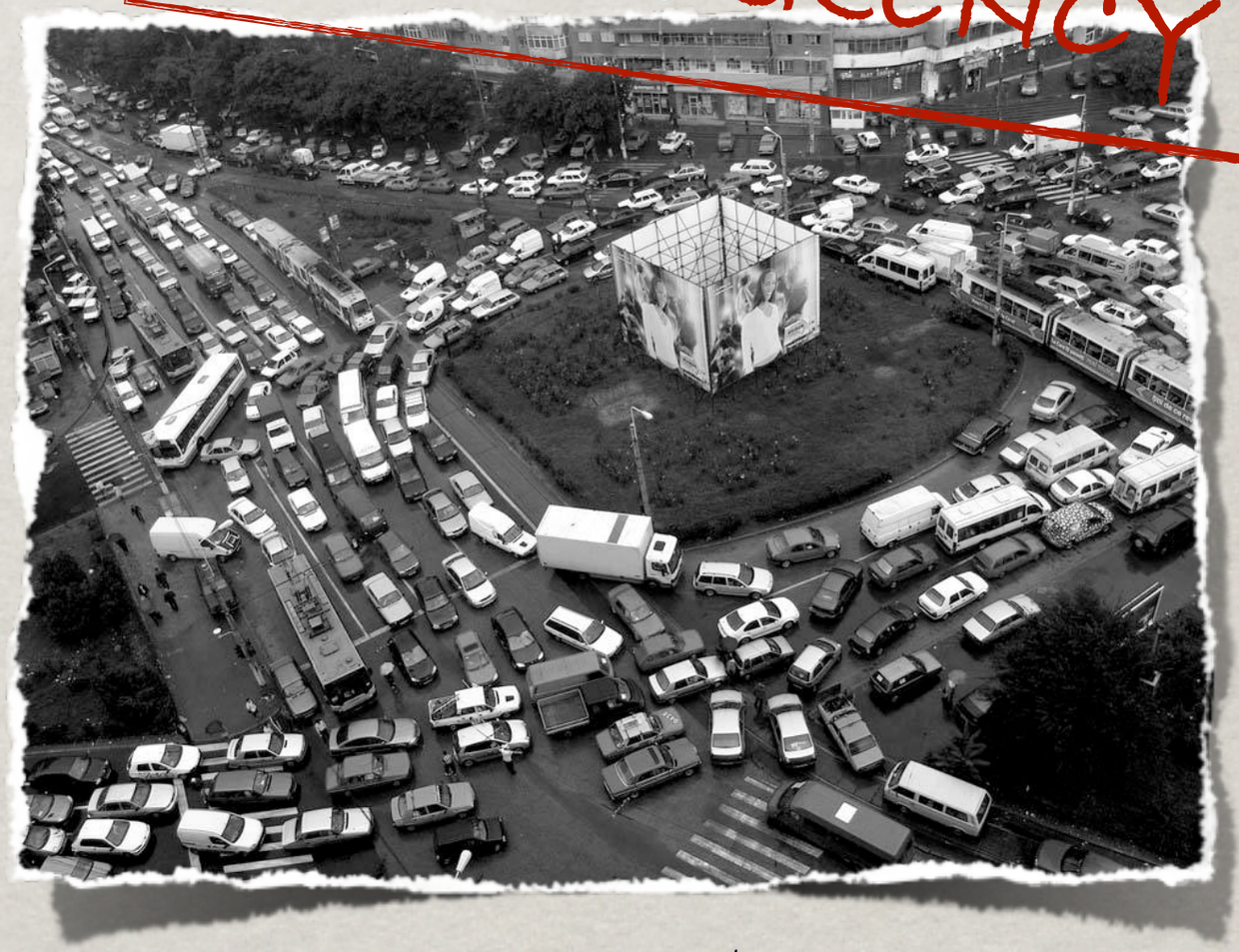
JVM languages: **what** they are, **why** we should care about them, and **which** ones are relevant to the work we do here at IDC

Step One

Admitting that you have a problem...

BLOAT

CONCURRENCY



...or two.

As Java developers, we have a problem
Actually, we have two problems: **bloat** and **concurrency**
Slow us down
Make us less productive
Let me show you what these problems look like...

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

Clojure

```
(map - (filter even? (range 1000)))

(map inc (filter #(= (str %) (apply str (reverse (str %)))) (range 1000)))
```


Top we have some Java code that...
...palindromes are character sequences that read the same when reversed
I picked a fairly typical operation to implement ... this pattern comes up a lot
This is what I mean by bloat; taking 10x

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

Scala

```
(0 until 1000).filter( _ % 2 == 0 ).map( -_ )

(0 until 1000).filter( n => n.toString == n.toString.reverse ).map{1+}
```


Object Oriented and Functional hybrid that runs on the JVM
...focusing a little bit more on it later

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

Haskell

```
map negate $ filter even [0..1000]

map (+1) $ filter (\n -> (show n) == (reverse $ show n)) [0..1000]
```


Few “pure” functional languages in use today
Used as a research language
...made some inroads in the financial sector
Make their way into more mainstream languages
...lazy evaluation and type inferencing

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

Python

```
map(lambda n: -n, filter(lambda n: n % 2 == 0, range(1000)))

map(lambda n: n + 1, filter(lambda n: str(n) == str(n)[::-1], range(1000)))
```


We've all heard of Python
Original Google web crawler was written in Python
...it remains one of the few approved languages at Google

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

C#

```
Enumerable.Range(0, 1000).Where(n => n % 2 == 0).Select(n => -n);
Enumerable.Range(0, 1000).Where(n => n.ToString().Equals(new string(n.ToString().Reverse().ToArray())))
  .Select(n => n + 1);
```


Surprise! C# – Java clone, imperative

Microsoft Research has been actively sneaking functional programming techniques into .NET through their LINQ extension and the recent

inclusion of F# in Visual Studios, installed on how many thousands of developer machines around the world

C# and Java compete in the same market, my opinion that this poses a huge challenge

No good developer wants to write 10x the code and I doubt IT managers would be very happy about it either

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

JavaScript

```
var xs = new Array();
for (var i = 0; i < 1000; ++i) {
   xs[i] = i;
}
xs.filter(function(x) { return x % 2 == 0 }).map(function(x) { return -x });
xs.filter(function(x) { return x.toString() == x.toString().split('').reverse().join('') }).map(function(x) { return x + 1 })
```


Fun!

Arguably the most hated programming language in recent memory (the bad parts) – is cleaner and more concise

Same sequence generation problem though

Still, it’s quite a bit smaller

Why?

Why are all those other languages so concise?
What common attributes do they share...
...which allow them to perform the same task...
...but with less explicit instruction?

Imperative

```
final List<String> names = new ArrayList<String>(Arrays.asList(
    new String[] {"fred", "rOb", "john", "Mike", "paul", "will", "jefF"}));
final Iterator<String> it = names.iterator();
while (it.hasNext()) {
    final String name = it.next();
    for (int i = 0; i < name.length(); ++i) {
        if (Character.isUpperCase(name.charAt(i))) {
            it.remove();
            break;
        }
    }
}
System.out.println(names);
```

Declarative

```
(remove (fn [name] (some #(Character/isUpperCase %) name))
  [“fred” “rOb” “john” “Mike” “paul” “will” “jefF”])
```


In simple terms: focus on **what** to do, rather than **how** to do it
focus on the **logic** of computation are called **declarative** / focus on describing the control flow (for loops incrementing array offsets) are called **imperative**
Declarative operate at a higher level / abstracting over the implementation details
Skipping the “how” = less code / more readable: “remove each name containing some character that is upper case”
Let’s go back to those examples and take a closer look at the actual language features responsible for the reduction in size...

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

Clojure

```
(map - (filter even? (range 1000)))

(map inc (filter #(= (str %) (apply str (reverse (str %)))) (range 1000)))
```


First language feature **higher order functions**
Functions that accept or return other functions
...**map**, which applies a function (negation function) to each element in seq
...**filter**, which applies a predicate function (even?) to each element in a seq, keeping those which are true for that predicate
filter = for+if / map = for+add(transformed)

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

Haskell

```
map negate $ filter even [0..1000]

map (+1) $ filter (\n -> (show n) == (reverse $ show n)) [0..1000]
```


Second language feature is **partial function application**
(+1) / the addition function takes minimum of two
Pre-filling some parameters, leaving the rest to be applied later
When all parameters are filled in, the function executes
Reduces parameter duplication and function arity

Java

```
public static List<Integer> negateEvens(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        if (x % 2 == 0) {
            result.add(-x);
        }
    }
    return result;
}

public static List<Integer> incrementPalindromes(final List<Integer> xs) {
    final List<Integer> result = new ArrayList<Integer>();
    for (final Integer x : xs) {
        final String original = x.toString();
        final String reverse = new StringBuilder(original).reverse().toString();
        if (original.equals(reverse)) {
            result.add(x + 1);
        }
    }
    return result;
}

public static void main(final String[] args) {
    final List<Integer> xs = new ArrayList<Integer>();
    for (int i = 0; i < 1000; ++i) {
        xs.add(i);
    }
    System.out.println(negateEvens(xs));
    System.out.println(incrementPalendromes(xs));
}
```

Python

```
map(lambda n: -n, filter(lambda n: n % 2 == 0, range(1000)))

map(lambda n: n + 1, filter(lambda n: str(n) == str(n)[::-1], range(1000)))
```


Third language feature **anonymous function literals**
Functions that are defined without being bound to an identifier
Useful for encapsulating/passing around bits of one-off functionality + sequences
Now, we saw Java can be overly verbose, requiring low level details and lots of ceremony
More declarative languages don't suffer from this problem / **It follows then, that using them would solve our bloat problem**

What about concurrency?

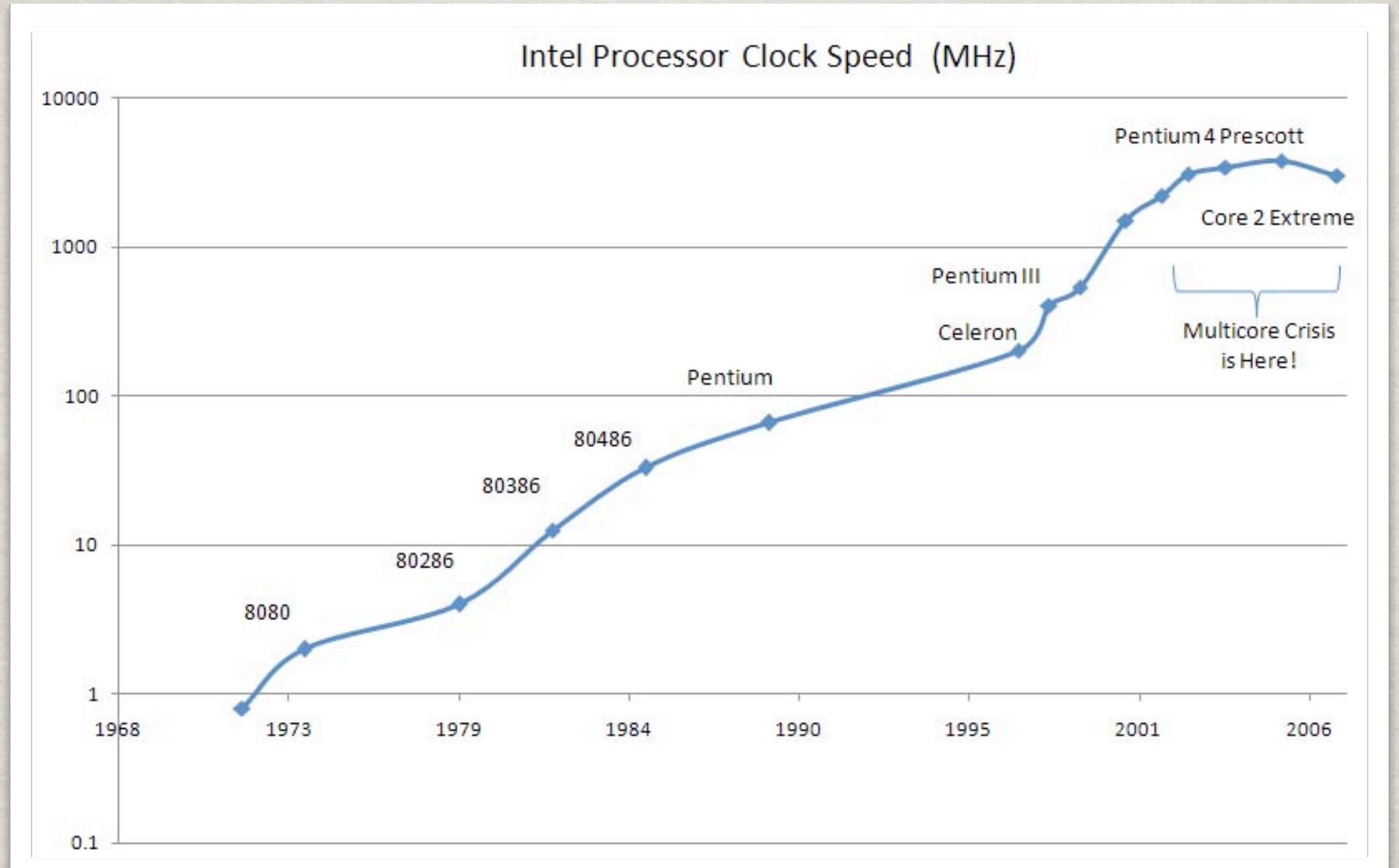
What about our second problem: concurrency?

“Concurrency is something only a particular group have to worry about:
operating systems, high frequency trading apps, broadcast management systems, and so forth...”

Business applications can continue to be written as serial, sequential batch jobs, and that it will be good enough

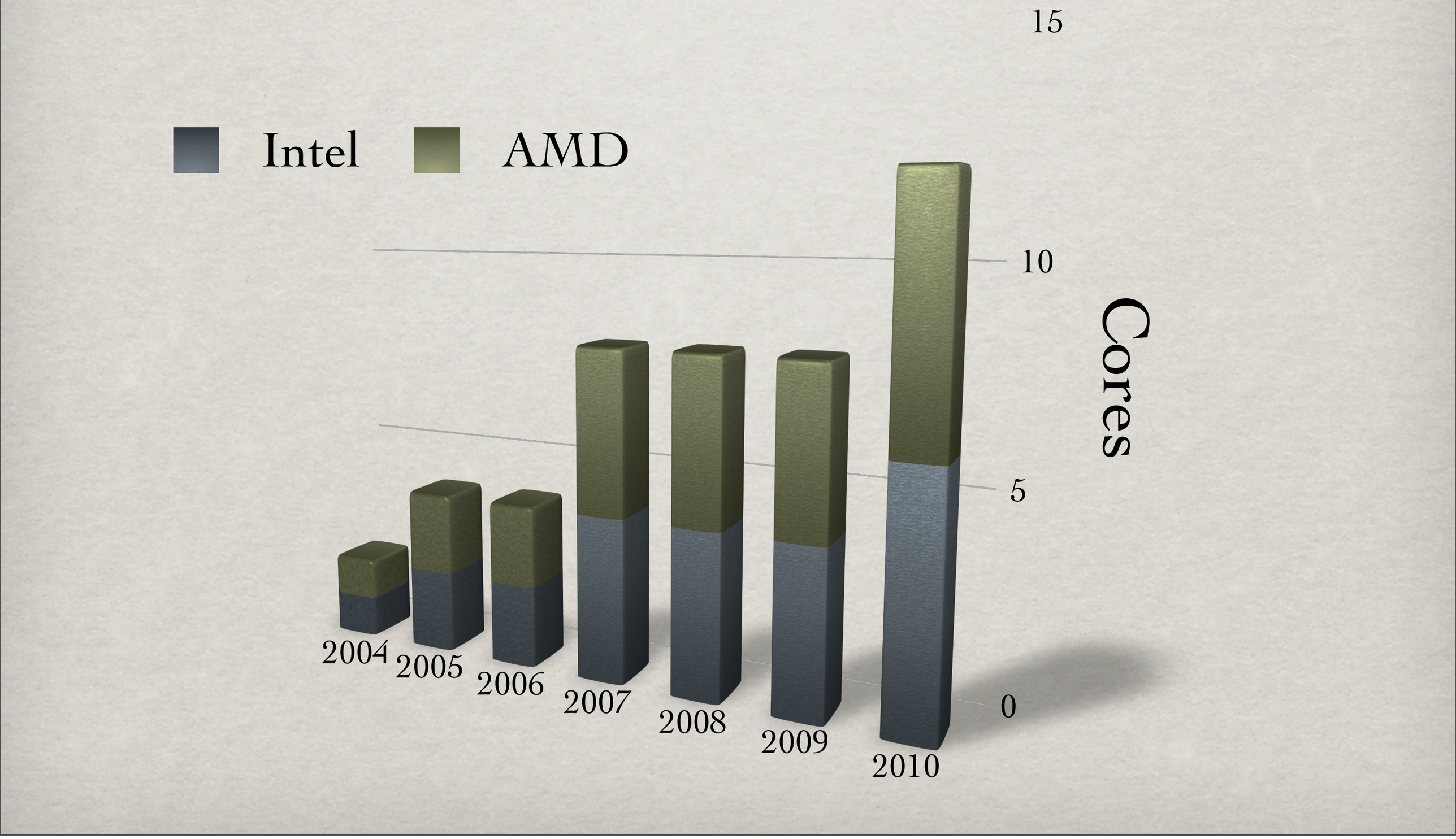
Let me show you why that conclusion used to be correct, but is now wrong

Serial Processing Limit



Graph of clock speed over the last 30+ years
From 70's until mid-2000, seeing steady increases in clock speeds, Intel brags about Mhz & Ghz / clock speed = performance
Developer could be confident that any additional features or work, would run in some reasonable time
The Problem: we're hitting a physical limit in the speed of serial processing
If we want out applications to have more features or process more data, we can't just wait until tomorrow for a free boost

The New Hotness: Multithreading



So what is Intel bragging about today? **Cores** / # cores = performance

The problem: for an app to take advantage of these new cores, developer has to do some work:
write his app in a multithreaded manner; needs to do work in parallel

In other words, he needs to write a **concurrent** application

So what's the problem?

Java's Solution: Locks

```
public class Deadlock {

   public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
    public synchronized void bow(Friend bower) {
        System.out.format("%s: %s has bowed to me!%n",
                this.name, bower.getName());
        bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s has bowed back to me!%n",
                this.name, bower.getName());
    }
  }
}
```

"Locks are too hard!!1!"

- a lot of very smart programmers

Java's **solution** is the problem: to prevent multiple threads from accessing a shared memory location at the same time, Java objects and classes provide **locks**

A thread needs to acquire an object's lock before it can invoke one of its synchronized methods

Other threads that wish to invoke that object's synchronized methods must wait, blocking until the lock is released

In practice, getting this all right is nearly impossible:

...in background is a 20 line app (w/locks) / most of the time deadlocks / but sometimes not (scheduler) / debugging nightmare.

Solution: JVM Languages

- Declarative (solves the bloat problem)
- No locks (solves the concurrency problem)
- Integrates with our existing 1.5 million lines of Java code
- Runs on our existing infrastructure
- Utilizes the vast ecosystem of Java libraries

There are numerous declarative JVM languages
Some of these offer a better alternative to locks
IDC has over one and a half million lines of Java code written. JVM languages can integrate seamlessly with this existing codebase.
These languages run on our existing infrastructure; we can deploy them to our JBoss servers and hook up our existing profiling tools
Unlike some other declarative languages like Haskell or Prolog, these languages can use any Java library or framework

What are JVM languages?

Any language that can run on the Java Virtual Machine

```
final Integer foo = 1000 + 44;
final Integer bar = foo - 31;
final Integer baz = bar - 25;
```

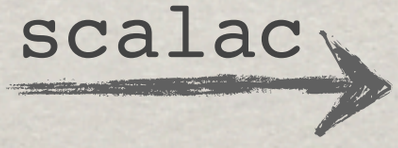
javac

```
(def foo (+ 1000 44))
(def bar (- foo 31))
(def baz (- bar 25))
```

cljc

```
val foo = 1000 + 44
val bar = foo - 31
val baz = bar - 25
```

scalac



```
 0:   iconst_2
 1:   istore_1
 2:   iload_1
 3:   sipush     1000
 6:   if_icmpge  44
 9:   iconst_2
 10:  istore_2
 11:  iload_2
 12:  iload_1
 13:  if_icmpge  31
 16:  iload_1
 17:  iload_2
 18:  irem
 19:  ifne       25
```


In other words, any language that compiles to JVM bytecode
On the left you have some JVM languages: Java, Clojure, and Scala.
Once you run them their respective compilers, you will end up with similar bytecode.
The bytecode probably won't be exactly the same, because each language provides their own primitive wrappers, define operations, and so on,
..but it will be pure JVM bytecode that can call or be called by one another.

Which language is right for us?

- There are over 60 languages to choose from
- Half are ports of existing languages
- Half are newly developed languages
- Mix of object oriented, functional, logic, and hybrid
- Mix of dynamic, static, and type inferenced

So if we come to the conclusion that we need to keep the Java platform, but augment the Java language with something more productive, how do we decide which JVM language is right for us?

IBM NetRexx
Kawa
Yeti
Adobe ColdFusion
Erjang
Judoscript
Jython
Bigloo
Railo
C
JGNAT
Scala
Sleep
Alef++
jLogo
Frink
CLforJava
myForth
Jill
Pizza
JRuby
Jelly
Noop
X10
Fantom
Mirah
Processing
ObjectScript
Groovy
XLogo
CAL
Jatha
Joy
Kahlua
Nice
SISC
Pnuts
Clojure
Open BlueDragon
Jaskell
Jacl
Hecl
Ioke
NetLogo
Fortress
Rhino
Veryant isCobol
OCaml-Java
JScheme
Jawk

Wow, okay. This is only **some** of the languages currently running on the JVM.
So we definitely need some sort of filter to cut this list down a bit.
First, let's remove any language without some published books, since we're most likely going to be studying them independently
(not many colleagues know them, training might be hard to come by, and so on...)

IBM NetRexx

Adobe ColdFusion

Judoscript

Jython

Bigloo

Railo

Scala

JRuby

X10

Processing

Groovy

SISC

Clojure

Open BlueDragon

NetLogo

Rhino

Okay, that's better.
So now let's select those languages showing some short term, long term, or exceptionally powerful momentum;
we wouldn't want to spend our limited time learning some fringe language with poor adoption rates.
* TIOBE Programming Community Index

Jython

Scala

JRuby

Groovy

Clojure

Now, each one of these languages is mature, has strong communities, multiple books, and forward momentum.
Any of these would be worth learning, and each would provide a decent productivity boost.
Jython and JRuby are obviously ports of existing languages, while Scala and Clojure are mostly new.
So which one to choose? Well, they're all more declarative than Java, but what about our second problem, concurrency?
Let's select those languages with a compelling answer to the concurrency problem.

Scala

Clojure

All five of the previous languages supported your standard Java concurrency primitives, but these two go a step further...

Scala provides message passing concurrency via **Actors** (concurrent components communicate by exchanging messages) / Erlang

Clojure provides shared memory concurrency via a **STM** (concurrent components communicate by altering the contents of shared memory locations) / C#, Java

Usually some form of locking is used; with a STM, there are no locks; developers treat the shared memory much as they would a transactional database

Let's pick the language with a flatter learning curve...

Scala

Clojure - being a Lisp dialect - has a steeper learning curve due to its syntax and more purely functional nature.
Since Scala is instead a hybrid language, both object oriented and functional, it's completely possible to write imperative, Java-style Scala.
This allows developers to slowly ease into the language, adding more powerful functional techniques as they become more comfortable.
So when management asks, "Where am I gonna find Scala programmers?", we can answer,
"The same places you find Java programmers: throughout IDC and the greater market," **almost honestly**...

So what's Scala look like?

So now that we've picked Scala as IDC's first alternative JVM language, let's see what it actually looks like

Object Oriented

```
class Person(val firstName: String, val lastName: String) {
  private var position: String = _
  println("Creating " + toString())

  def this (firstName: String, lastName: String, positionHeld: String) {
    this (firstName, lastName)
    position = positionHeld
  }

  override def toString() : String = {
    firstName + " " + lastName + " holds " + position + " position "
  }
}

val john = new Person("John", "Smith", "Analyst")
println(john)
val bill = new Person("Bill", "Walker")
println(bill)
```


Scala = more OOP than Java / dumped all statics in favor of a native singleton, simplifies inheritance
Primary constructor parameters / Private field / Private accessor methods are created for you / The _ sets the initial value to the default for that type
Any expression or executable statement you put in the class definition is executed as part of the primary constructor
Auxiliary constructor / Scala requires an **override** keyword, while Java makes it optional
Prevents accidental overriding a superclass method or implementing a new method when your intention was to override an existing one

Functional (yay!)

Higher-order functions	`def apply(n: Int, fn: Int => Int): Int = { fn(n) }`
Anonymous function literals	`apply(5, i => i + 1)  // 6`
Partial function application	`val inc = apply(_ : Int, i => i + 1)` `inc(7)  // 8`
Function composition	`val incAndNegate = inc andThen (-_)` `incAndNegate(4)  // -5`

Closures	`var count = 0`
	`apply(1, step => { count += step })`
	`// count is now 1`

Currying	`def apply(n: Int)(fn: Int => Int): Int = { fn(n) }` `apply(3) { i => i + 1 }  // 4`

Scala has all of the functional features we saw in our bloat examples before
Currying in Scala is a bit odd; the standard definition of currying is that of partial function application. In other words, they typically refer to the same concept.
In Scala, however, currying refers to this splitting up of parameter lists to enable programmers to write function literals between curly braces.

This can make the method call feel more like a control abstraction and it improves the appearance of larger function literal definitions.

Hello World

1. mvn archetype:generate -U \
   -DarchetypeGroupId=net.liftweb \
   -DarchetypeArtifactId=lift-archetype-basic_2.8.0 \
   -DarchetypeVersion=2.1-RC2 \
   -DremoteRepositories=http://scala-tools.org/repo-releases \
   -DgroupId=demo.helloworld \
   -DartifactId=helloworld \
   -Dversion=1.0-SNAPSHOT

2. cd helloworld

3. mvn jetty:run

4. http://localhost:8080/

So what language introduction would be complete without a good old Hello World demonstration?

mvn archetype:generate -U -DarchetypeGroupId=net.liftweb -DarchetypeArtifactId=lift-archetype-basic_2.8.0 -DarchetypeVersion=2.1-RC2 -DremoteRepositories=http://scala-tools.org/repo-releases -DgroupId=demo.helloworld -DartifactId=helloworld -Dversion=1.0-SNAPSHOT

You now have a complete Scala web application up and running. **Afterwards, show REPL**

An IDC Scala Library

`/commons-nlp`

- Natural Language Processing services client
- Initial package is a port of existing Java
- 70% reduction in lines of code
- No concurrency / Actors (yet)
- Various improvements

Background: an NLP Skunkworks project / Bronnie from Web Channel wanted to move forward with it
I wrote the the prototype in Clojure because I had multiple readers and writers of this shared data structure.
If it was in Java, I would have had to: 1) move the shared integration point to a db, 2) used locks, ...3) abandon doing any of these things at the same time and move to a batch process, doing writes at 2AM in a single thread, then stop it and allow HTTP reads
Let's take a look at some of the improvements we've made thanks to the Scala feature set

The Takeaway

- The existing investment and high quality of the Java platform makes it a keeper
- The Java language has some serious problems we can't afford to ignore
- We can improve our productivity and value by adding these languages to our toolbox
- Some of us have already started. Get in touch.

There is a cost to these languages:
...I have an issue w/syntax (as a hybrid, lots of features = lots of syntax),
Alex had issues w/documentation & debugging
Both of us found the plugins buggy
None of these costs are unique to Scala; they are found in any new technology

Resources

This presentation is available online
http://public.iwork.com/document/?a=p1045023190&d=JVM_Languages_at_IDC.key

Alex has started writing up some “tips & tricks” Wiki articles on Scala
http://usfr-confluence.insideidc.com/display/framework/Scala

The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software
http://www.gotw.ca/publications/concurrency-ddj.htm

The IDC **commons-nlp** Calais library
http://svn.insideidc.com/repos/frameworks/commons/trunk/commons-nlp/

The Scala IRC channel on Freenode: #scala