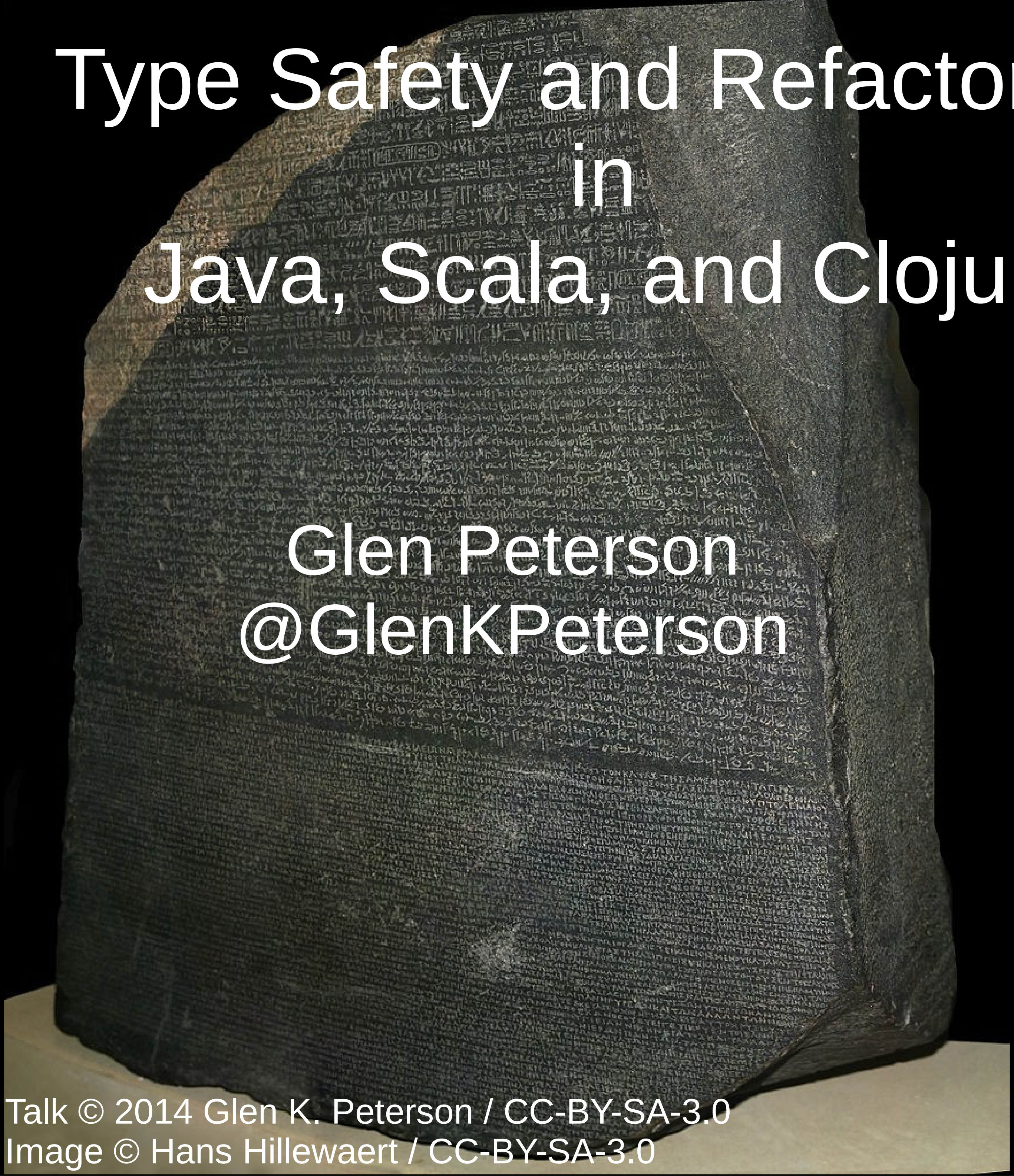


Type Safety and Refactoring in Java, Scala, and Clojure

Glen Peterson
@GlenKPeterson

Contact Us: 978.409.1248

Customer Support: 978.432.9339

Email Us: TheTeam@planbase.com

PlanBase Hoshin

Elevate Your Organization's Strategic Planning Process

PlanBase Scorecard

Achieve Operational Excellence Across Your Organization

Outline

- Motivation: Explore type safety through examples
- Initial design
  - Java
  - Scala
  - Clojure
- Change underlying data and refactor
  - Java
  - Scala
  - Clojure
- Conclusions

What is Type Safety?

Image: http://en.wikipedia.org/wiki/File:Kobushi-mini-dvi.jpg

Benefits of Type Safety:

“The earlier in the development cycle you can find a bug, the cheaper it is to fix.” Catching bugs in the compiler or IDE is many times cheaper than a bug a client discovers in production.

Documenting Expectations/Limits (compiler verified documentation)

Refactoring a large code base: if fixing one thing breaks anything else, type safety forces you to fix all of those things before your code will compile, hence “safe.”

Run-time performance on JVM languages is better with type safety. Significant?

Benefits of Dynamic Languages:

“I'm more productive without worrying about types.”

Solve the core problem first, then sweat out the details.

Try out half-baked ideas quickly to see how well they work - don't have to “make all the types work out” before you can compile something just to try it.

Can be less code

Can be simpler code

Design as you go

Type Safety: Good or Bad?

Image: http://en.wikipedia.org/wiki/File:Computer-connector-sockets.jpg

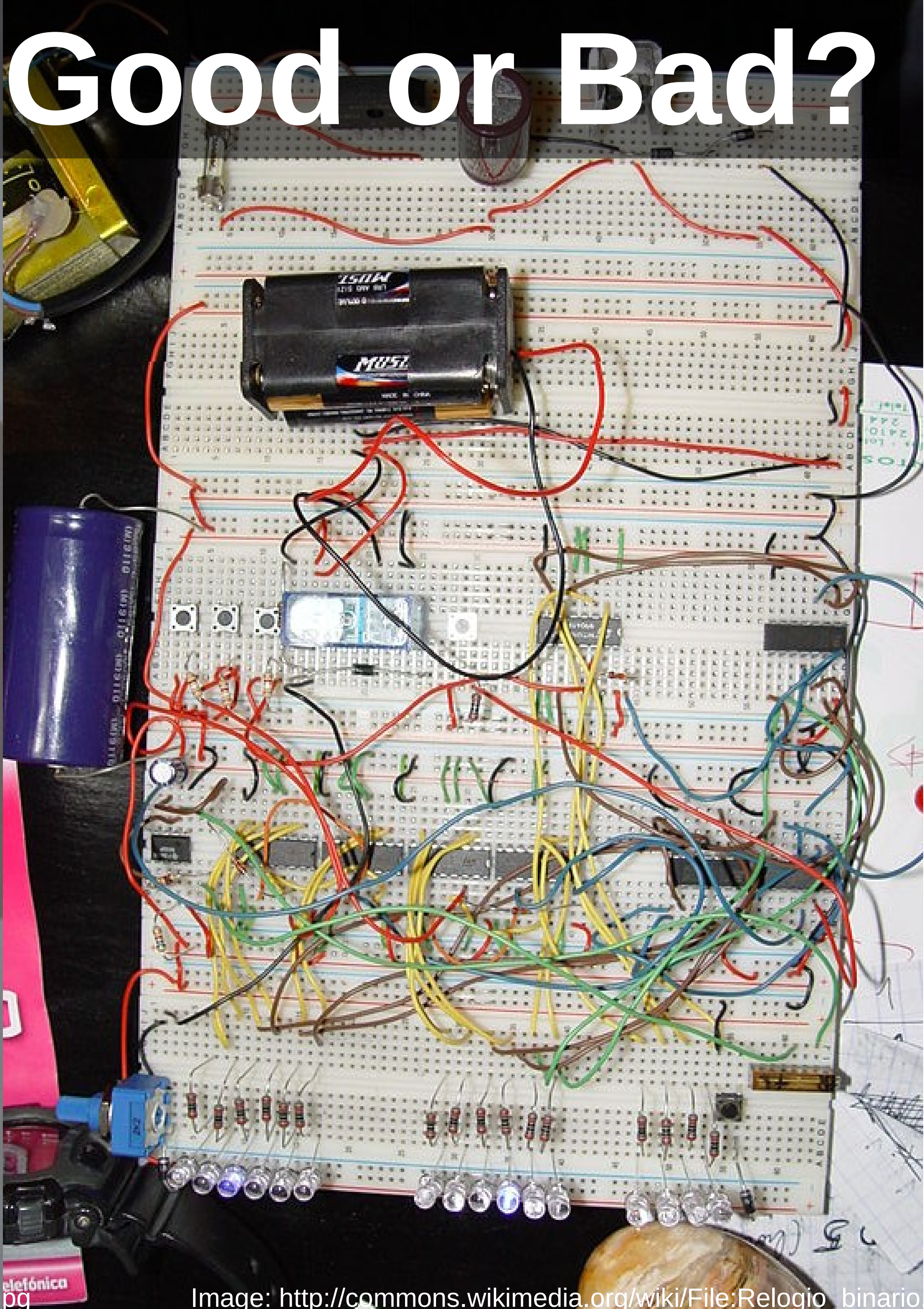
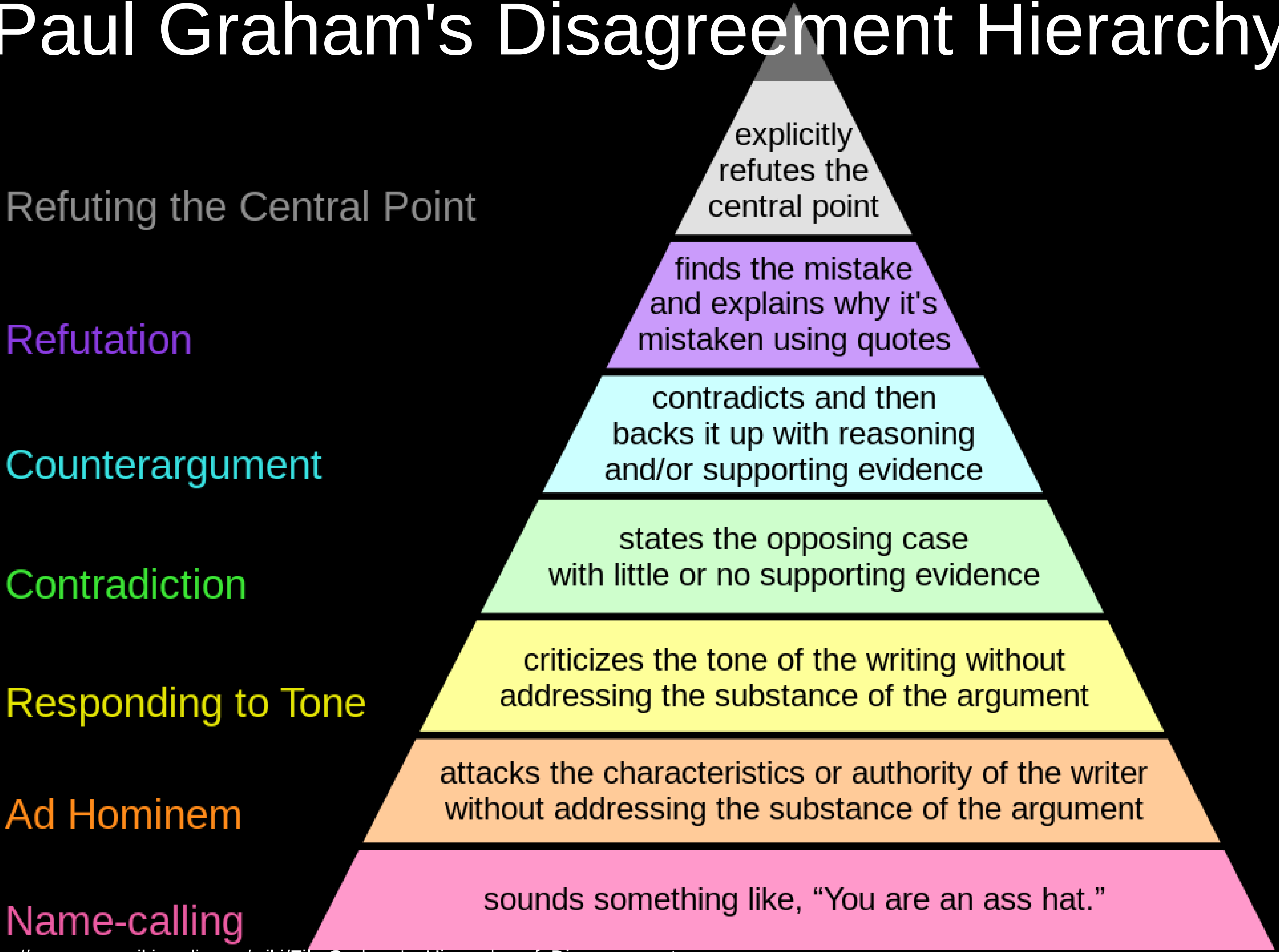


Image: http://commons.wikimedia.org/wiki/File:Relogio_binario

Paul Graham's Disagreement Hierarchy


Refuting the Central Point
explicitly refutes the central point
Refutation
finds the mistake and explains why it's mistaken using quotes
Counterargument
contradicts and then backs it up with reasoning and/or supporting evidence
Contradiction
states the opposing case with little or no supporting evidence
Responding to Tone
criticizes the tone of the writing without addressing the substance of the argument
Ad Hominem
attacks the characteristics or authority of the writer without addressing the substance of the argument
Name-calling
sounds something like, "You are an ass hat."

http://commons.wikimedia.org/wiki/File:Graham's_Hierarchy_of_Disagreement.svg

Problem Description: Original Design

- Work with dates as a year and month combination, ignoring days
- A “Fiscal Month”, not a date! Doesn't necessarily correspond exactly to a calendar month
- Two Fields:
  - Year
  - Month
- Simple
- No packing/unpacking
- Fast modification
- Easy to read

Problem Description: Queries

```
-- Querying a single YearMonth is Simple,
-- safe, and fast!
select * from monthly_thing mt
  where mt.year = :year and
        mt.month = :month
```

Format for upcoming code

- Present typical solutions
  - Java
  - Scala
  - Clojure
- Each solution:
  - Includes addMonths function
  - Meaningful toString
  - Immutable
  - Hopefully idiomatic for that language

Show Code

Show Original Code

Query Problems

```
-- Querying a single YearMonth is Simple,
-- safe, and fast!
select * from monthly_thing mt
  where mt.year = :year and
        mt.month = :month

-- But querying a range (start-ym to end-ym)
-- is Confusing, error-prone, and slow.
select * from monthly_thing mt
  where (mt.year > :startYear and
         mt.year < :endYear) or
        (mt.year = :startYear and
         mt.month >= :startMonth) or
        (mt.year = :endYear and
         mt.month <= :endMonth)
  order by mt.year, mt.month
```

Problem Description: Changed Design

- One Field:
  - Year * 100 + Month = YyyyMm
- Sorting alphabetically = sorting numerically = sorting by date
- Still easy to read
- Easy/fast to query either singly or as a range

Better Queries

```
-- Simpler, still safe, and faster!
select * from monthly_thing mt
  where mt.yyyy_mm = :yyyyMm

-- Simpler, safer, and faster!
select * from monthly_thing mt where
   mt.yyyy_mm >= :startYyyyMm and
   mt.yyyy_mm <= :endYyyyMm
   order by mt.yyyy_mm
```

Problem Description: Java Thoughts

- Would I write an adapter, then wrap each implementing class in the adapter everywhere it is used?
- No. That's just as much or more work than updating all the implementing class.
- Problems like this happen all the time. Sometimes it is a change of data representation, sometimes a change of function signature or functionality.

Show Code

Show Changed Code Now

Other Considerations

- Compiling Scala is slow, taking 2-3x as long as Java. Sbt, on the other hand, is extremely clever, maximizing processor usage and deciding not to compile everything unless it needs to, so compiling Scala with SBT may effectively be faster than compiling Java with Ant.
- Compiled Clojure code might execute 50% slower than comparable Scala/Java code (YMMV)?
- Both Scala and Clojure require a few small jar files to compile which hold their specific APIs.

Recap: All Three Languages:

- Accomplished the work
- Required specifying logic/functions

Clojure vs. Java

Rich Hickey says, “Clojure is a replacement for Java”

Jason Wolfe's* team started missing type-safety-ish guarantees in Clojure after 3 Person Years (Jason authored Prismatic/schema).

- REPL == easier testing as you go. Ends bad ideas quickly.
- Less code == simplicity
- Lisp == simplicity
- Strongly functional == safety
- Assumes immutability == safety

Scala and Clojure

The big wins associated with these languages have to do with an assumption of immutability and functional programming.

	Scala	Clojure
Types	Strongly-typed	Dynamic
Syntax	C/Java meets Haskell	Lisp
Style	Object-Oriented and Functional	Functional
Mutability	Mutable or Immutable	Biased toward Immutability
Attitude	Scala can do that	Keep it simple

Java...

- Java solved virtually every major issue with C++
- Popularized fast, reliable garbage collection
- Created the JVM which these other two languages are built on.
- Huge API and existing tool set (accessible from both Scala and Clojure)
- Compiles faster than Scala
- More backward-compatible than any language except COBOL. :-)

Type-safe Scala

- Has ???, Nothing, and Anything types so that you can compile to try things out without dotting every I and crossing every T
- Type safety is an iron-clad guarantee vs. certain kinds of bugs. Testing can only ever catch what you test for. Fewer tests means simpler code.
- Types are documentation that the compiler verifies. This can make code easier to READ.
- Ideal for large projects, multiple developers

Dynamic Clojure

- Define functions / data-transformations first
- Data structures almost disappear altogether
- Less to think about when coding – simpler and faster to WRITE
- The cost is run-time safety and compiler-enforced documentation
- Comprehensive unit test coverage can mitigate this risk, but that is another kind of effort, which introduces complexity and keeping tests updated has a high maintenance cost.
- Ideal for < 3 person-year projects
- Prismatic/schema can add safety after-the-fact

Additional Solutions

- Java: OOP:Interpreters, Functional:Visitors
  - Any good typing course can improve your speed and accuracy.
- Scala: Traits are the official solution (as shown)
  - “Independently Extensible Solutions to the Expression Problem” paper by Zenger/Odersky
- Clojure: simplicity. defrecord/defprotocol or multimethods.
  - “Clojure's Solutions to the Expression Problem” talk by Chris Houser on InfoQ