The Next Big JVM Language

Stephen Colebourne
Member of technical staff
OpenGamma Ltd

Agenda

- What is a Big Language?
- What did Java get right?
- What did Java get wrong?
- What can Java do to evolve?
- What will define the Next Big Language?
- What about <insert-favourite-language>?

What is a Big Language?

What is a Big Language?

- A key or dominant language
- Widely used
- Wide job market
- Established
- Ecosystem
  - tools, libraries, books, training
- Community
  - conferences, podcasts, websites

Big Languages

- C
- C++
- Java
- C#
- COBOL
- Visual Basic
- Perl
- PHP
- Javascript

- Ruby?
- Python?
- Fortran?

NBJL

- NBJL = Next Big JVM Language
- The language that will challenge Java
- and eventually displace it
- *on the JVM*

- Adding JVM is a significant restriction
- Much of this talk is applicable without the JVM

What did Java get right?

What did Java get right?

- Java is 15 years old
- Used by 10 million+ developers
- Most widely used enterprise language
- Clear step forward from C++

- Java got *lots* right
- Like C to C++, it was evolution not revolution

Original design goals

- Simple, Object Oriented and Familiar
- Robust and Secure
- Architecture Neutral and Portable
- High Performance
- Interpreted, Threaded and Dynamic

http://java.sun.com/docs/white/langenv/Intro.doc2.html

Creating Java

- Start from C++
  - and Objective C, Smalltalk, Modula...
- Simplify
  - remove redundancy and complexity
  - tackle bug hotspots
- But keep syntax
  - looks similar to C and C++
  - easy to learn

Learning from C /C++

- Well defined primitive types including boolean
- Array bounds checking
- No unsigned numbers
- No typedef, preprocessor or headers
- No struct, union, enum, function
- No multiple inheritance
- No operator overloading or goto
- No fragile superclasses
- No access to memory structures

Key features

- Object oriented
- Static typing
- Interfaces
- Packages
- Checked exceptions
- Unicode
- Standard library
- Reflection & Class Loader
- Multi-threading

JVM

- Old idea, but JVM made it popular
- Platform neutral
- Safety
  - verifier
  - array bounds checks
  - strong typing
- Garbage collection

What did Java get wrong?

What did Java get wrong?

- Judgement is historical!
- 15 years ago some things were good choices
- Now they are not
- Mostly design trade-offs

Wrong question!

What have we learnt?

- Real world experience with Java
- Knowledge from other languages
- Technology advancements
- Performance gains

- Learnt *lots*
- Examine 7 Java features we can learn from

Checked exceptions

```
String fileContent = generateFile();
try {
  // method call can throw IOException
  writeFile(file, fileContent);

} catch (IOException ex) {
  // ignore
}
```

- Good idea in theory
- Bad idea in practice
- Code cannot know if caller can handle exception
- Failed experiment
- NBJL will not have checked exceptions

Primitives

```
// specialist code for each primitive
public void process(Object obj) { ... }
public void process(byte b) { ... }
public void process(short b) { ... }
public void process(int b) { ... }
public void process(long b) { ... }
public void process(char b) { ... }
public void process(float b) { ... }
public void process(double b) { ... }
```

- Included to ensure good performance
- Split in the type system
- Breaks "everything is an object"
- Lots of repetitive methods
- NBJL will not have exposed primitives

Arrays

```
// array length
int len = array.length;

// assignment can cause errors
Integer[] intArray = new Integer[1];
Number[] numberArray = intArray;
numberArray[0] = Long.valueOf(3);  // ArrayStoreEx
```

- Directly exposes JVM bytecode features
- Split in the type system
- Breaks "everything is an object"
- Runtime check rather than compile time
- NBJL will not have exposed arrays

Everything is a monitor

```
public class Foo {
  private static final String LOCK = "LOCK";
  private final List<String> lock = new ArrayList<>();

  public void process() {
    synchronized (LOCK) {
      // use list
    }
  }
}
```

- Seems like a nice simple approach
- Results in unsafe effects
- Causes difficulty optimising JVM
- NBJL will not expose every object as a monitor

Static

```
public class BaseClass {
  public static void create(String str) { ... }
}

public class SubClass extends BaseClass {
  public static void create(String str) { ... }
}
```

- Static methods do not override
- Hard to call from a framework → use of reflection
- Cause proliferation of factory classes
- Static state hurts concurrency
- NBJL will avoid static, especially static state

Method overloading

```
public class Foo {
  public void process(Integer i, double d) { ... }
  public void process(int i, Double d) { ... }
}

Short s = 6;
double d = 1.2
foo.process(s, d);
```

- Hard for humans to work out which is called
- Also hard for the compiler!
- Most complex part of compiler
- Superclass, interface, varargs, generics, boxing...
- NBJL may not have method overloading

Generic wildcards and variance

```
public <T,U> Function<T, U> createFn(
  List<? extends Function<? extends T, ? extends U>> a,
  Predicate<? super T> predicateT,
  Predicate<? super U> predicateU) { ... }

public <T extends Enum<T>> List<T> getList() { .. }
```

- Wildcards and variance are hard
- Backwards compatibility → erasure
- Integration with non-generics → warnings
- Very confusing and complex!
- NBJL will have a better approach to generics

What can Java do to evolve?

Evolution

- Prime directive

  Backwards compatibility

- Secondary considerations

  Following the "feel" of Java

  Conceptual integrity

VERY HARD to evolve Java now

Evolution - Clashes

- Generics clashes with
  - primitives
  - arrays
  - varargs
  - method resolution
- "Closures" clashes with
  - primitives
  - arrays
  - checked exceptions
  - method resolution

Evolution – small changes

- Diminishing returns of evolution
- But still big benefits due to 10 million+ users
- Still simple worthwhile additions
  - project Coin
  - other small ideas
    - map for-each
    - default-if-null operator
    - switch-break fix
    - catching "impossible" checked exceptions
    - ...

Evolution – bigger changes

- Properties
- Control abstraction
- Reified generics
- Traits
- Continuations
- Design by Contract
- Nullable types
- Immutability

- Expect slow (or no) progress on these
- Many complications
- How to add in a backwards compatible way

What will define the NBJL?

Language design

- Balancing numerous factors
- Academic research
- Lessons learnt from elsewhere
- Language goals
- Human factors

Human factors are critical!

Human factors

- Take a sample piece of code
  - sensible example of new language
  - typical code from expected day-to-day use
- Give it to a mid-level developer
  - not interested in other languages
  - not an alpha geek
- Can they make a reasonable guess?
  - no tuition
  - no documentation

The "Journeyman"

"Journeyman Programmers aren't idiots or morons: they often are just as good as rockstars, just not as passionate. That means that they're not investigating programming language features and Open Source libraries in their spare time, they're not going to meetups and blogging and tweeting and everything else. Raw talent often, passion seldom."

Blue collar

- Java is a "blue collar" language
- Designed for everyday usage
- Purity is a non-goal
- Collected good ideas from elsewhere

- NBJL will be a "blue collar" language
- Relatively unexciting
- Alpha geeks will find it boring

Syntax

- Syntax important in language adoption
  - curly braces – C-like
  - meaningful indentation – Python-like
  - wordy – Ruby-like
- Java has emphasised C-like as mainstream

- NBJL will have C-like syntax
- Quicker to grasp for language newcomers
- Good enough

Paradigm

- Object oriented
  - classic example – Smalltalk
  - classes, objects and message sending
- Functional
  - classic example – Lisp
  - immutability, functions, code is data
- Pure solutions on each side not very popular

Paradigm

"Java has no functions. Object-oriented programming supersedes functional and procedural styles. Mixing the two styles just leads to confusion and dilutes the purity of an object-oriented language. Anything you can do with a function you can do just as well by defining a class and creating methods for that class."

http://java.sun.com/docs/white/langenv/Simple.doc2.html#5642

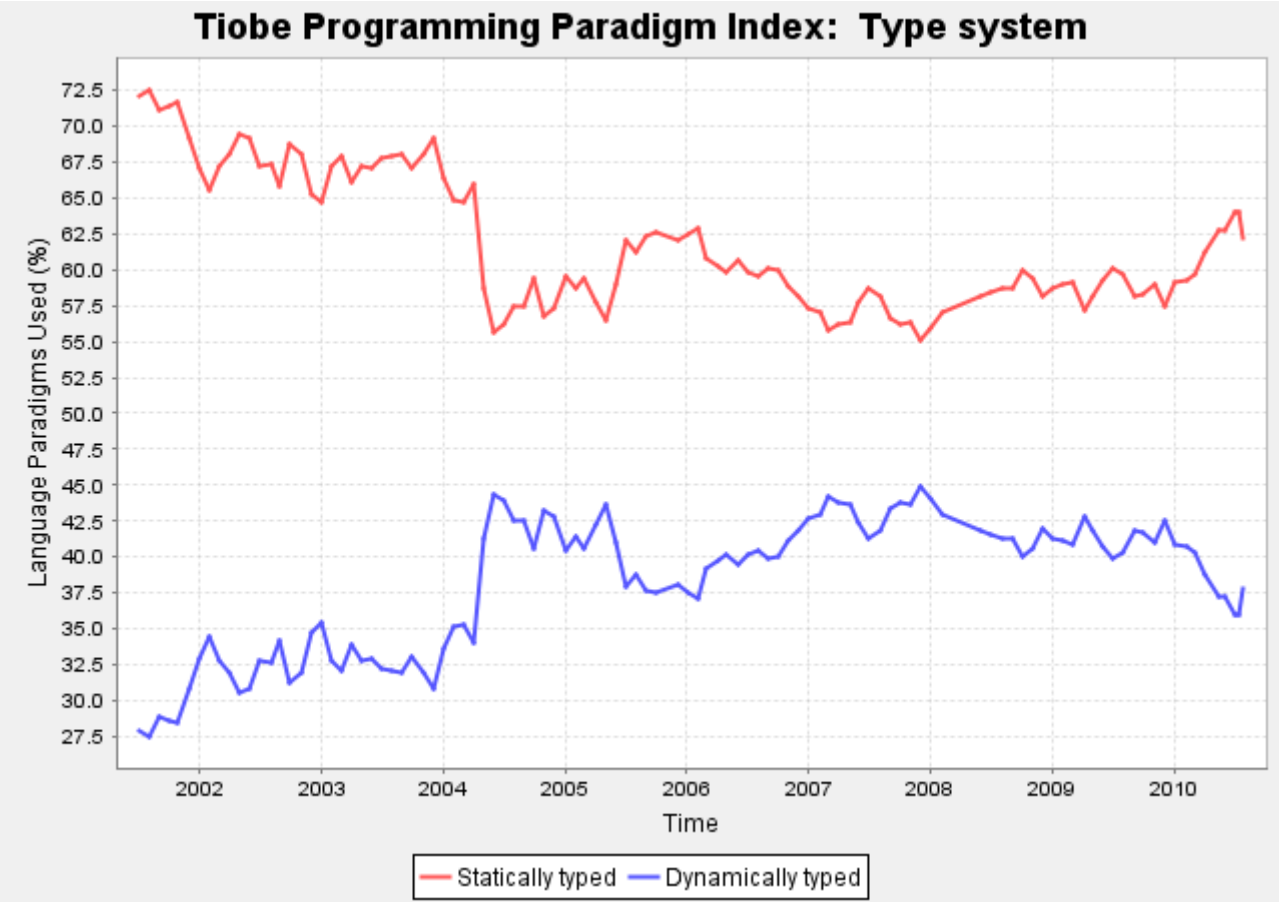
Paradigm

- Java is Object Oriented, not Functional
  - OO is less pure than Smalltalk
  - popularity may be linked to being less pure
- Potential to mix OO and Functional
  - where is the right boundary?

- NBJL will be a combination of OO & Functional
- Pure Functional too hard for mainstream
- Huge developer base that knows OO
- Closures & first class functions will be included

Type system

- Static vs Dynamic
- Industry split



Static typing has a Bad Press

- Rise of Ruby (dynamic)
  - "10 times more productive"
- Verbosity of Java (static)
  - "converts large XML files into large stack traces"

Java is Verbose...

Java has static typing...

Therefore static typing is Bad!

Invalid connection!!!

Type system

- Weaknesses with dynamic
  - If a method operates on a Person, what use is it to allow an Address to be passed in?
  - Dynamic code often has unit tests that simulate type checks performed by static compiler
  - Static typing on API is documentation

- NBJL will have a static type system
- Preferred for enterprise scale systems
- Good for teams of mixed ability

Type system

- Adding too much power to type system can have negative side effects
  - Need to think in the way that the compiler thinks
  - Have to write long/complicated API signatures
  - Harder to read an API signature

- NBJL will not have an excessive static type system
- Developers should not have to think like a compiler
- Everyone must be able to read API signatures

Reflection

- Reflection is needed in static typed language
- Way to simulate some dynamic behaviour
- Reflection in Java is hard

- NBJL will have easy access to reflection
- Will have literals for methods and fields
- Reflective calls will very simple

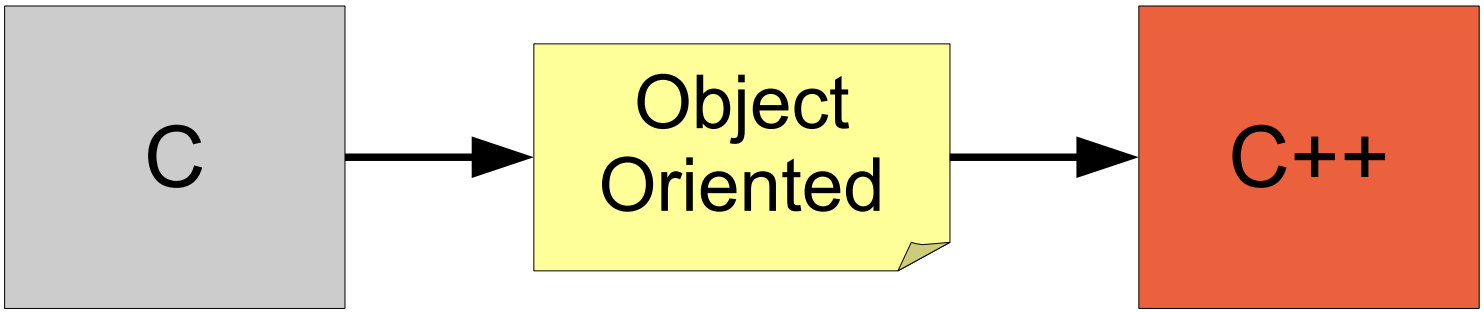
Patterns

- An excess of patterns is a language smell
- Java has lots of patterns

- Java is quite a smelly language!
- One way languages evolve is via Patterns

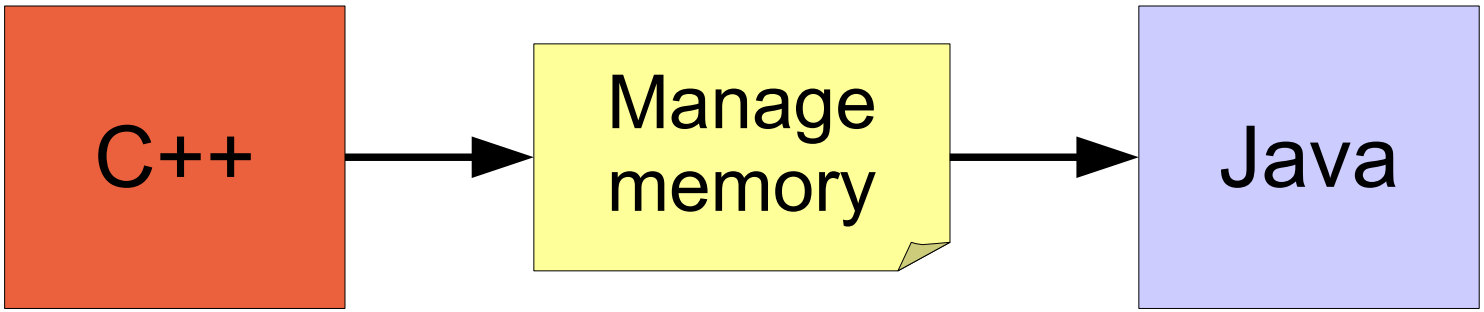
Patterns

- Can write Object Oriented code in C
- Specific design pattern to follow
- Evolution to C++ encoded pattern into language



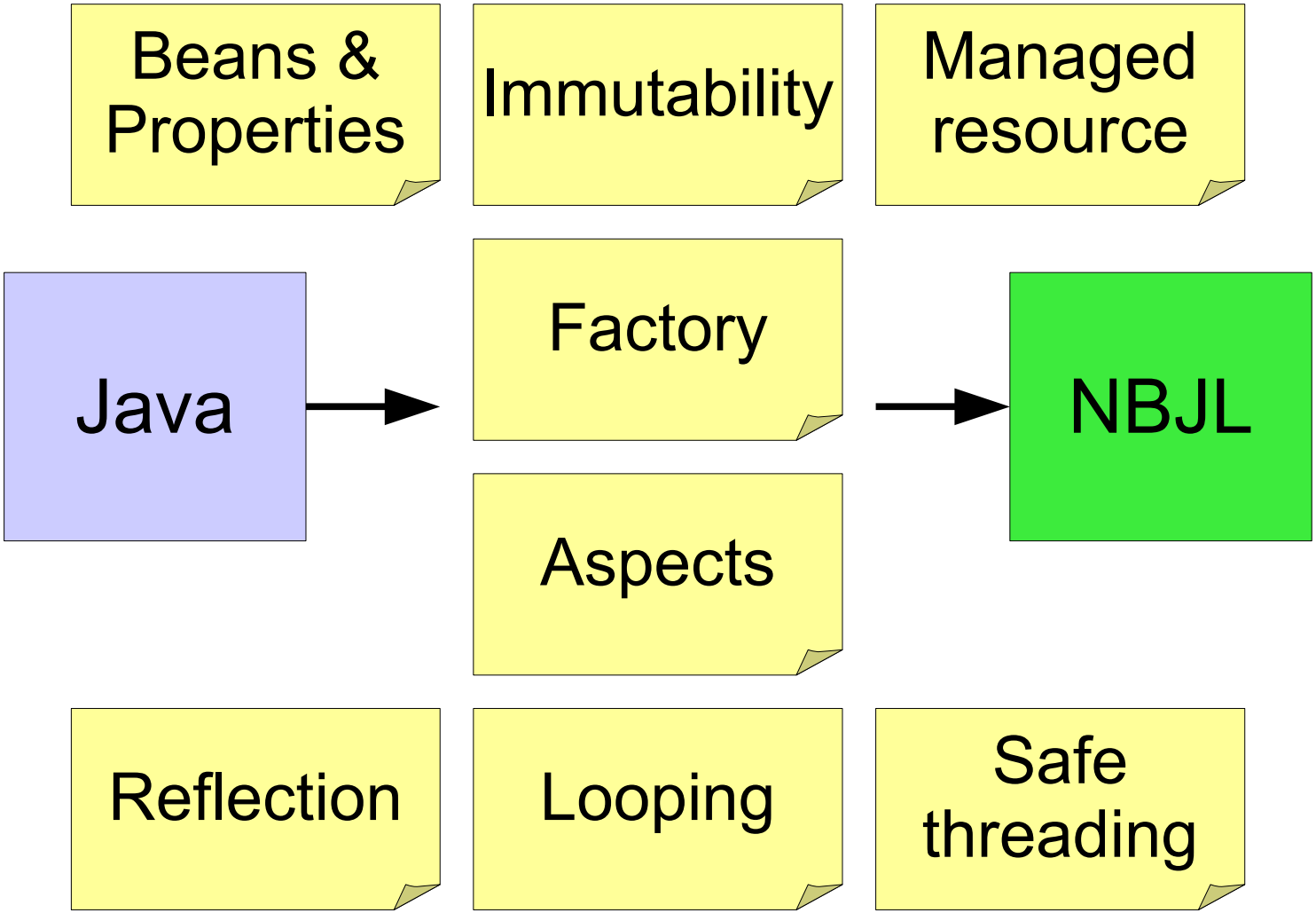
Patterns

- Can write safe memory management in C++
- Specific design pattern to follow
- Evolution to Java encoded pattern into language



Patterns

- Java has lots of patterns to encode in NBJL



Beans and properties

- Java Bean pattern very wasteful
- Properties raise the abstraction level
- Meta level access
  - to and from XML, JSON, NoSQL
  - query languages – Xpath, EL

- NBJL will have properties
- Core language feature
- Probably have some kind of linked events

Closures

- Looping in Java is very procedural
- Closures capture looping patterns
  - filtering – retain only some elements
  - searching – find first/best element
  - totalling – find a sum/total for all elements

- NBJL will have closures
- Core language feature
- May be the only way too loop
- May also capture resource management pattern

Factories

- Java libraries are full of factories
- Function types
  - "factory is a function that produces desired type"
- Method references
  - "give me a function for this method"

- NBJL will have function types and method refs
- Core language feature
- Key elements of functional style
- Avoids needless SAM factory interfaces

Null handling

- Checking for null is a pattern
- Hard to track which data can be null
- Source of frequent bugs
- Capture knowledge in language/library

- NBJL will have better null-handling
- Will avoid most NPEs
- Few reasons to check directly for null

Concurrency

- Major issue as machines now multi-core
- Code needs to use cores successfully
- Developers need to code differently?

- Not necessarily!!!
- Web application server is naturally parallel
- Typical developer can still write the same code
- Easy to over-sell concurrency problem

Concurrency

- Java threading was a major advance
  - now concurrent libraries are state of the art
- Different models being tried
  - actors
  - software transactional memory (STM)

- NBJL will not rely on threads
- Threads may not even be exposed
- Immutable objects very important

Modules

- Java is retrofitting modules
  - very difficult challenge
- Better to build modules in from the start
- Huge benefits for versioning code

- NBJL will have modules
- Core language feature
- May be based on OSGI

Tools

- Tooling is critically important
- Developers expect and require IDE
  - all in development, especially auto-complete
- Common JVM provides some tools
  - profiler, debugging hooks

- NBJL will have good tooling
- Language design will make tooling easy

Extensibility

- Java is a locked down language
  - only Oracle makes changes
- Lisp allows any change
  - everyone codes in their own dialect/language
- Need a middle ground

- NBJL will be extensible
- May be a combination of language features
- May be a single dedicated extension point
- Extensions will be controllable

Other features

- Performance
- Map/list/regex literals
- Standard scoping – public/private/protected
- Some form of operator overloading
- Standard build, test and log approaches

Java → NBJL

```
public class Person {
  private String surname;
  private String forename;
  private List<Address> addrs = new ArrayList<Address>();
  public String getSurname() { ... }
  public void setSurname(String surname) { ... }
  public String getForename() { ... }
  public void setForename(String forename) { ... }
  public List<Address> getAddresses() { ... }

  public List<Address> getAddresses(String type) {
    List<Address> filtered = new ArrayList<Address>();
    for (Address addr : addrs) {
      if (addr.getType().equals(type)) {
        filtered.add(addr);
      }
    }
    return filtered;
  }
}
```

Java → NBJL

```
public class Person {
  private String surname
  private String forename
  private List<Address> addrs = new ArrayList<Address>()
  public String getSurname() { ... }
  public void setSurname(String surname) { ... }
  public String getForename() { ... }
  public void setForename(String forename) { ... }
  public List<Address> getAddresses() { ... }

  public List<Address> getAddresses(String type) {
    List<Address> filtered = new ArrayList<Address>()
    for (Address addr : addrs) {
      if (addr.getType().equals(type)) {
        filtered.add(addr)
      }
    }
    return filtered
  }
}
```

Java → NBJL

```
public class Person {
  public String surname
  public String forename
  public List<Address> addrs = new ArrayList<Address>()

  public List<Address> getAddresses(String type) {
    List<Address> filtered = new ArrayList<Address>()
    for (Address addr : addrs) {
      if (addr.type.equals(type)) {
        filtered.add(addr)
      }
    }
    return filtered
  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  List<Address> addrs = new ArrayList<Address>()

  List<Address> getAddresses(String type) {
    List<Address> filtered = new ArrayList<Address>()
    for (Address addr : addrs) {
      if (addr.type.equals(type)) {
        filtered.add(addr)
      }
    }
    return filtered
  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  Address[] addrs = []

  Address[] getAddresses(String type) {
    var filtered = Address[]
    for (Address addr : addrs) {
      if (addr.type.equals(type)) {
        filtered.add(addr)
      }
    }
    return filtered
  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  Address[] addrs = []

  Address[] getAddresses(String type) {
    var filtered = Address[]
    for (Address addr : addrs) {
      if (addr.type == type) {
        filtered.add(addr)
      }
    }
    return filtered
  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  Address[] addrs = []

  Address[] getAddresses(String type) {
    var filtered = Address[]
    for (Address addr : addrs) {
      if (addr.type == type) {
        filtered.add(addr)
      }
    }
    return filtered
  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  Address[] addrs = []

  Address[] getAddresses(String type) {
    return addrs.filtered( #(addr){addr.type == type} )

  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  Address[] addrs = []

  Address[] getAddresses(String type) {
    return addrs.filtered #(addr){addr.type == type}

  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  Address[] addrs = []

  Address[] getAddresses(String type) {
    return addrs.filtered #(addr){addr.type == type}
  }
}
```

Java → NBJL

```
public class Person {
  private String surname;
  private String forename;
  private List<Address> addrs = new ArrayList<Address>();
  public String getSurname() { ... }
  public void setSurname(String surname) { ... }
  public String getForename() { ... }
  public void setForename(String forename) { ... }
  public List<Address> getAddresses() { ... }

  public List<Address> getAddresses(String type) {
    List<Address> filtered = new ArrayList<Address>();
    for (Address addr : addrs) {
      if (addr.getType().equals(type)) {
        filtered.add(addr);
      }
    }
    return filtered;
  }
}
```

Java → NBJL

```
class Person {
  String surname
  String forename
  Address[] addrs = []

  Address[] getAddresses(String type) {
    return addrs.filtered #(addr){addr.type == type}
  }
}
```

Verbosity

- Verbose code obscures meaning
- Java developers tend not to see it
- Trained ourselves to be blind

- NBJL will not be verbose
- Code will still be clear and well-documented

What about ⟨ Clojure
Groovy
Scala
Fantom
... ⟩?

Clojure

- Modern Lisp
- Fully functional
- Software transactional memory
- Clever approach

- Very different to Java
- Fully functional approach not suited to "blue collar"
- Not C-like syntax, or many other familiar items
- Not NBJL

Groovy

- Dynamic
- Syntax and style very close to Java
- Lots of convenience elements
- Performance has been a problem
  - no longer such an issue

- Similar to Java, including many of the warts
- Not a clean, fresh start
- Dynamic
- Useful complement to Java, not a replacement
- Not NBJL

Scala

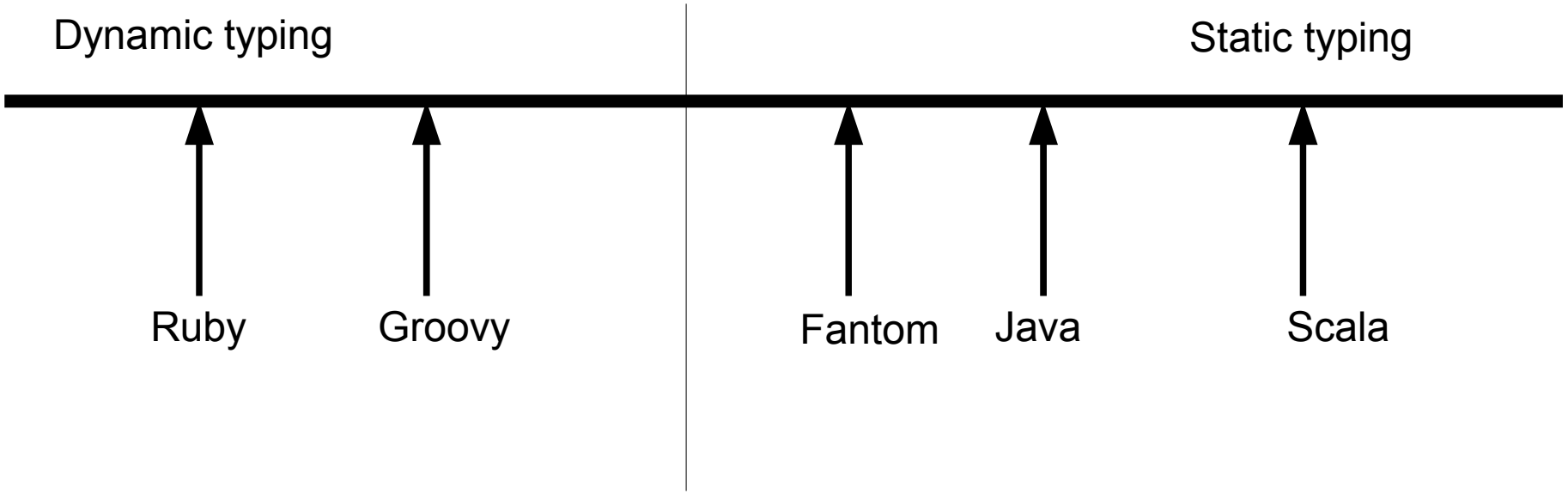
- Static typing
- Goal of being "scalable" to small/large systems
- OO/Functional hybrid
  - emphasis on functional
- Focus on minimal language
  - enabling maximal libraries
  - many language features become library features
- Syntax is C-like braces, but reversed types

Fantom

- Static typing, with dynamic elements
- Goal of being "blue collar"
- OO with functional elements
- Variables are not-null/nullable
- Immutability built in
- No method overloading
- Modules
- Syntax has similar look to Java

Scala and Fantom type system

- Different approaches to type system
  - Scala adds more knowledge and strength
  - Fantom takes a more relaxed approach
- Both fully define types in APIs



Scala and Fantom code

```
// Java
int calculate(
    int baseCost, int rate, int amount) {
      baseCost + rate * amount;
}

// Scala
def calculate(
    baseCost : Int, rate : Int, amount : Int) : Int
      = baseCost + rate * amount

// Fantom
Int calculate(
    Int baseCost, Int rate, Int amount) {
      baseCost + rate * amount
}
```

Scala and Fantom code

```
// Scala
def calculate(
    baseCost : Int, rate : Int, amount : Int) : Int
      = baseCost + rate * amount
```

Scala and Fantom code

```
// Scala
def calculate(
    baseCost : Int, rate : Int, amount : Int) : Int
      = baseCost.+(rate.*(amount))
```

Scala and Fantom

- Scala enables library-based features
  - operators are just method calls
  - full operator overloading
  - any symbol can be a method
- Fantom uses language-based features
  - operators are defined by language
  - limited operator overloading
  - cannot add additional operators

Scala and Fantom code

```
// Java
Person p = findPerson();
String city = "";
if (p != null &&
    p.getAddress() != null &&
    p.getAddress().getCity() != null) {
  city = p.getAddress().getCity()
}
```

Scala and Fantom code

```
// Scala
def findPerson(key : Key) : Option[Person] { ... }

val p = findPerson(key);
val city = p flatMap (_.address)
                              flatmap (_.city) getOrElse ""


// Fantom
Person? findPerson(Key key) { ... }

p := findPerson(key);
city := p?.address?.city ?: ""
```

Scala and Fantom

- Scala enables library-based features
  - null-handling is just a library
  - functional Option type
  - force the functional style to be learnt
- Fantom uses language-based features
  - null-handling is language feature
  - not-null/nullable types
  - ?. and ?: operators

Scala and Fantom code

```
// Scala
val list = Array(1, 2, 3, 4)

val sum = list.foldLeft(0)(_+_)

val sum = (0 /: list)(_+_)


// Fantom
list := [1, 2, 3, 4]

sum := list.reduce(0) |a, b|{a + b}
```

Scala and Fantom

- Scala is a big jump from Java
  - culture is functional programming, not better Java
  - full-featured type system
  - developers need lots of training

- Fantom is a smaller jump from Java
  - features chosen to not be scary
  - simple type system
  - most features are small and independent

Scala or Fantom?

- Scala is too complex
- Provides too much rope to hang yourself
- Easy to become a write-only language
- Type system is excessive

- Fantom is closer
- Language design has good feature set
- Likely to remain readable
- Type system is probably too weak

- Seems unlikely that either will be NBJL

Opportunity

- 10 million+ developers
  - most do not read blogs
  - most do not go to conferences
  - most will not get training for a new language
- Do we force them to use Java forever?
- Need to focus on their needs

Opportunity

- Make 10 million developers 5% more efficient
- Huge financial benefit

- Middle-of-the-road static type system
- Learn the language lessons from Java
- Also learn the platform lessons
- Work with existing Java libraries
- Easy to learn

Java

- What about Java?

- Create a backwards incompatible JDK version
  - Remove 15 years of language warts
  - Add new features like closures easily/properly
  - Enhance platform – modules and versioning
- Tool converts "old Java" to "new Java"
- Similar enough to minimise training
- Evolution, not revolution

- What if this was JDK 8 ?

Summary

Summary

- All good languages are Turing Complete
- Different ways of looking at same problem
- Each has specific goals
- "Which is better" is always subjective
- Can make some more objective judgements

Summary

- NBJL
  - learns from Java
  - "blue collar"
  - static typing
  - C-like syntax
  - OO with functional elements
  - closures
  - broad set of expected features

Summary

- In my opinion
  - NBJL isn't Clojure!
  - NBJL isn't Groovy!
  - NBJL isn't Scala!
  - NBJL isn't Fantom!

- Best answer may be a backwards incompatible Java version, perhaps instead of language changes in currently planned JDK 8

Questions

Stephen Colebourne
Member of technical staff
OpenGamma Ltd