We can do better, IBM’s vision for the next generation of Java runtimes

John Duimovich

IBM Distinguished Engineer
Java CTO
About me

- IBM Java CTO, responsible for all IBM Java runtimes
- 25 years experience developing virtual machines, runtimes, tools
- Developer of J9, IBM's high performance production JVM
- Eclipse Tools PMC lead

- john_duimovich@ca.ibm.com
- http://duimovich.blogspot.com
Title: "We can do better - IBM's vision for the next generation of Java runtimes"

Abstract: There is no question that the Java platform is successful. There are mature high performance runtimes, a rich ecosystem with extensive libraries, a tools ecosystem and broad innovative community supporting the platform. Even with all these great things, software development remains difficult, the reuse of components elusive and new runtime requirements for cloud, multicore, big data, and security are pushing the limits of these runtimes. Leveraging modularity and service architectures, coupled with runtime enhancements, we can enable continued innovation while maintaining compatibility, improving performance and making it easier to create, evolve and reuse software components.

A technology perspective on what needs to change in Java to address the challenges of producing modular software in the future
Todays talk (the switch)

1. Assess
   - Java, OSGi and the ecosystem today

2. Mock (nicely)
   - If we’re so smart, why do we do stupid things?

3. Hope
   - Opportunities for Java and modularity solving next generation problems

4. Profit!
   - Exercise for the audience
Java today

- Java is a mature ecosystem
- High performance and reliable runtimes
- Rich set of developer friendly tools, libraries
- Large developer friendly ecosystem
  - Eclipse, Apache, OSGi

- Java is perfect!

Perfect?
(@#$%^*)
Java's not done

- Developers face big challenges
  - “Cloud” - virtualization, footprint/density, runtime dynamism
  - “Big*.*” - data, threads, memory, scale out, scale up
  - Security - critical to connected software and hard to get right
  - Compatibility – need to innovate and protect existing investment in code and tools
  - Plus ... mobile, polyglot, development efficiency, simplicity, software lifecycle

- Java needs to evolve and address the new reality of increased complexity
  - Eclipse and OSGi will play a large role in developing the solutions
Java Platform Evolution

- 2011
  - Java 7 - Scripting (JSR 292), concurrency libraries, new “new I/O”
- 2013
  - Java 8 – Lambda, Date/Time,
  - Smaller SE profiles (Full + Compact 1-2-3)
- 2015
  - Java 9 – Modularity and ???

Too fast?

Innovation?

Too slow?
Java needs ...

- Cloud, Big*, Security and retain Compatibility

- Plus the wish list
  - Runtime resource control {memory, sockets, files} scoped by {context, module}
  - Runtime capabilities use {locks, threads, finalization} scoped by {context, modules}
  - Language extensions for new primitive types (value types) and packed data formats
  - High performance memory model primitives for improved parallelism (no unsafe)
  - High performance Foreign Function Interface (FFI) and “Structs”, a better Java Native Interface (JNI)
  - Reified generics and true lambdas
  - Large arrays, restartable exceptions, read-only objects, unsigned ints
  - ... and 200 other things

- You may argue the list at your leisure but we likely agree on:

a module system for good software engineering support and functional enhancements
What do I care about?

Performance

Simplicity

Cloud!

Software Engineering
The Patterns Approach: “exe for the Cloud”

Workload Pattern

Initial instance = 3

Virtual Application Instance

WAS cluster configured with session replication

<table>
<thead>
<tr>
<th>Web Application</th>
<th>Database</th>
<th>User Registry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaling Policy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load balancer

Monitoring, Life cycle management
IBM Application Pattern for Java in the IBM PureApplication System

Build

- Connect in
- HTTP Listener on 8080
- Web Application
- Java Application
- JVM Policy
- Apache Tomcat 7.0.27 Server
- Monitored file
- Apache Tomcat logs

Deploy

- Java Application
- Dependencies for fit for purpose application
- Templates to create your own PaaS

- Java Runtime
- Dynamic profile for cloud environment
- M Bean and Health Center monitoring

- Operating System
- Pre-configured for a cloud environment
- CPU, memory, storage and network monitoring

Monitor
Driving High Density Java in the cloud

- Share-nothing (maximum isolation)
  - Mission critical apps
  - 1+ GB / tenant
  - Hardware
  - Middleware
  - OS Images
  - Application

- Shared hardware
  - Shared memory, shared CPU
  - 1+ GB / tenant
  - Hardware
  - OS Images
  - Middleware
  - Application

- Shared OS
  - 100’s MB / tenant
  - Hardware
  - OS Image
  - Middleware (e.g. WAS)
  - Application

- Shared Process
  - 10’ s MB / tenant
  - OS Image
  - Middleware (e.g. WAS)
  - Application
  - Hardware

- Share-everything (maximum sharing)
  - 10’ s KB / tenant
  - Isolate API
  - Tenant
  - Application
  - Middleware (e.g. WAS)
  - OS Image
  - Hardware

- 10 MB / tenant
- 10 MB / tenant
- 10 MB / tenant
- 10 MB / tenant

- Isolation
- Density
Driving High Density Java in Virtualized Environments

- Share More
- Cooperate
- Use Less
- Exploit

Modularity?
“Liberty” Profile – For Web, OSGi and Mobile Apps

Lightweight runtime for Cloud
• Web profile server <50 MB zip
• Small memory footprint < 50 MB
• Fast server start times: < 2 secs

Standards Based Modularity for Cloud
• Java EE++ built on OSGi modules and services
• Modularity now (Java SE 6 and up)

Dynamic Modular Runtime
• Dynamic configuration
• “Right-sized” app profiles
• Lifecycle support

Developer First Focus
• Fast easy setup
• Simple server configuration
• Integrated Eclipse AppDev tools
• No restart required code changes

A highly composable, dynamic server profile
If we’re so smart, why do we * do stupid things **?

*we == polite technical term for “the industry”
** example – insults in keynote
OSGi is great technology

✓ Solves a really hard problem
  – Enhanced **Java** with enforced Modularity, a Service oriented architecture, and dynamic module Lifecycle management

■ Modularity is goodness
  ✓ Decomposition of large software systems
  ✓ Modules as a software building component
  ✓ Provisioning and deployment
  ✓ Version and configuration management
  ✓ Explicit dependencies allow users to reason about modules

✓ Adopted by essentially all Enterprise Middleware Servers
  – Multiple implementations validate specification quality

✓ Supported by ecosystem of tools and runtime implementations
  – Eclipse, Netbeans, Apache, Bndtools and others

✓ Service Architecture to decouple dependencies, enable lifecycle
  – Service Registry decouples service consumers from service providers

✓ Evolving to distributed cloud services
  – OSGi Remote Services and Remote Service Admin

? OSGi is perfect !
OSGi and Java

- OSGi provides advanced modularity built on infrastructure not optimized for modules

- Java has basic modularity (aka encapsulation) features
  - Classes, Packages and public/default/protected
  - No module groupings for deployment, visibility control is insufficient
  - No versioning information (serialVersionUID notwithstanding)

- OSGi has worked within the limits of the Java platform
  - Class loaders as the base mechanism to create a module system is powerful but complex
  - “User space” classloaders, result in performance issues, complex implementations, heavyweight
  - Fragile as implementations deal with corner cases (unspecified concurrency rules)

- OSGi is powerful and solves complex problems but suffers because of Java’s limitations
  - The JVM guy says “hey ! the JVM can help ... “

Lets try to get modularity enhancements for OSGi into Java and the JVM... It’s time
JVM support for modularity?

- Imagine if you had JVM support for modularity implementations
  - “instant on”, shareable component, enhanced isolation, resource management, API evolution”
  - Enhancing security via layered additional control capabilities scoped by modularity
  - Plus a free unicorn with every runtime

- What do you need for this?
  - Module brokered class loading with proper support and clear semantics for trouble spots
    - “Class.forName” and Thread context classloaders
  - Deterministic high performance JVM class resolution and load
  - If you run lots of Java code between the JVM and a class resolution at runtime you lose
    - Need to pursue a cacheable - predictable location for all resolved classes while still enabling dynamic runtime facilities like ByteCode Instrumentation (BCI)

- Do we need a new module system?
  - No, but we could really use JVM support for building modular runtimes
Simplify!

- OSGi is powerful, deals with complex issues but needs to address simplicity
  - Some implementations of OSGi have error messages that are worse than C++ errors

```
constraint violation for package 'bar' when resolving module 11.0 between existing import 8.0.bar
BLAMED ON [[11.0] package; (&(package=bar)(version>=1.0.0)(!(version>=2.0.0)))
and uses constraint 10.0.bar
BLAMED ON 
[[11.0] package; (&(package=foo)(version>=1.0.0)(!(version>=2.0.0))),
[9.0] package; (&(package=boz)(version>=1.0.0)(!(version>=2.0.0)))
```

- (I have no concrete suggestions except OSGi implementations and Eclipse tools should address this)
  - In this case DON’T ASK A JVM GUY - JVM classloading, verify, and other errors also mystifying
Opportunities

Java and OSGi
Solving next generation problems
and driving adoption
Innovation and Java

- Innovation won’t wait for any particular technology
  - Rate and pace of innovation is accelerating and much of it outside of Java
  - Java and OSGi need to move faster for Cloud, Big *.*, game changers
  - “What have you done lately” - key to staying relevant
  - OSGi, Eclipse, Apache are in a great position to solve big problems

- Blame “compatibility” or standards bodies speed or open source groups or IBM or Oracle
  - It’s a myth that “Java can’t change, because it’s too <insert favourite excuse>”
Opportunities

- Compatibility
  - cripples innovation or innovation to solve compatibility?

- Multi-tenancy
  - Critical to cost efficiency in cloud scenarios

- Optimized runtimes, fit for purpose and “instance on” runtimes
  - Performance is not optional

- Driving OSGi adoption on new Java profiles
  - Embracing Java 8 profiles

- Discussions @theBar
Compatibility: two edge sword

- Binary Compatibility has been a key strength of Java
  - Protect customer investment by “never” breaking code
  - Clean API specification ensures evolution of API within specification and TCK
  - Ridiculously old code will run unmodified

- Supporting “eternal” binary compatibility is difficult
  - JVM complexity – old code drives special use cases in JVM
  - Code bloat – deprecated is “just a suggestion” and code is loaded whether it’s used or not
## Compatibility is hard!

### Evolving API Interfaces

Evolving API interfaces is somewhat more straightforward than API classes since all methods are `public` and `abstract`, all fields are `public static` and `final`, all type members are `public` and `static`, and there are no constructors. Annotation types (`@interface`), which are a form of interface, are also covered.

<table>
<thead>
<tr>
<th>Action</th>
<th>Compatibility Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add API method</td>
<td>Binary compatible (0)</td>
</tr>
<tr>
<td>Delete API method</td>
<td></td>
</tr>
<tr>
<td>Move API method up type hierarchy</td>
<td>Breaks compatibility (1)</td>
</tr>
<tr>
<td>Move API method down type hierarchy</td>
<td>Breaks compatibility (7)</td>
</tr>
<tr>
<td>Add API field</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Delete API field</td>
<td>Breaks compatibility (2)</td>
</tr>
<tr>
<td>Expand superinterface set (direct or inherited)</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Contract superinterface set (direct or inherited)</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>Add, delete, or change static initializers</td>
<td>Breaks compatibility (3)</td>
</tr>
<tr>
<td>Add API type member</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Delete API type member</td>
<td>Breaks compatibility (7)</td>
</tr>
<tr>
<td>Re-order field, method, and type member declarations</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Add type parameter</td>
<td>Breaks compatibility (2)</td>
</tr>
<tr>
<td>Delete type parameter</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Re-order type parameters</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Rename type parameter</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Add, delete, or change type bounds of type parameter</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>Add element to annotation type</td>
<td>Breaks compatibility (5)</td>
</tr>
<tr>
<td>Delete element from annotation type</td>
<td>Breaks compatibility (6)</td>
</tr>
</tbody>
</table>

© 2012 IBM Corporation
Versioning and Compatibility?

- How can modularity and versioning help here?
  - Meta information about the software and versions contain semantic intent declared by developer
  - So what?

- Version information and explicit dependencies are critical to the solution
  - Versions enable software to reason and use meta-facilities to assist in providing compatibility
  - Dependencies render explicit what needs to work and what can be “dropped”

- Enables thought experiments and optimizations ...
  - What if they don’t use “X”?
  - What if all versions loaded > 2.0?
Imagine: Version a class or package

- **API evolution?**
  - How about JVM supported field and method rename / rewrite on load?
  - Selective visibility based on compile time versions
  - JVM assisted refactoring for complex cases
  - Tools help migrate to new version apply source refactoring automatically from metadata in binaries

- **Serialization robustness?**
  - Class versions aware of prior versions and shapes and can provide conversions routines
  - Automatic “schema” migration via tools

- **Optimized deployment?**
  - Java instances “strict latest version” mode to ensure up to date platform code from loading
  - Optimized instances that offer smaller, faster, efficient deployment
  - Unused components not loaded, corner cases removed
    - Thread.stop, Thread.suspend, Thread.resume
Technology to help address compatible class evolution?

```java
public class Example V1.0 {
    public void a();
    public void b();
    public int badField;
}

Oops, don't use badField

public class Example V2.0 {
    public void a();
    public void b();
    @deprecated V1.0 public int badField;
    public int getField();
    public void setField(int);
}

Note: Imaginary Syntax

public class Example V3.0 {
    public void a();
    public void b();
    public int getField();
    @deprecated V1.0-2.0 public int badField map(getField, setField);
}
```
Versions to evolve class shapes

```java
public class Example V3.0 {
    public void a();
    public void b();
    public int getField();
    public void setField(int);
    
    // deprecated V1.0-2.0
    public int badField = map(getField, setField);
}
```

Run, no errors – binary compatibility ok

Re-compile

Re-write rules corrects reference

No mapping needed
Java modularity landscape

- Java 8
  - Lambda (Closures) are now main feature for Java 8
    - Note Lambda includes functionality for API evolution in “defender methods” for interfaces

- Modularity moves to Java 9
  - Jigsaw development continues with more time to “get it right”
  - OpenJDK JEP 161 has proposed Java SE profiles for smaller runtimes in the interim

- OSGi has more runway to influence the JVM support for modularity
  - Ensuring primitive operations (classloaders++) are supported to better support OSGi (Penrose)
  - Influence Jigsaw architecture to ensure separation of concerns for module system independence
  - Seamless OSGi on all JDKs with high performance now
Project Penrose's goal is to demonstrate interoperability between Jigsaw and OSGi.

- Evolving the JVM to support modularity is the right technical approach
  - Getting JVM primitives right is the key to OSGi performance bliss
  - Imagine your favorite module system on an optimized JVM foundation
  - Collaborator Mailing list to enable discussion between modularity implementers and JVM types

- OSGi is established and will continue to be used
  - Opportunities to drive adoption, understanding use cases in Java community
  - Adoption with customers will drive the future direction
Java SE 8 and Profiles

- JEP 161 @ OpenJDK is proposing three Java SE profiles execution environments
- Eclipse Java tools are already enabled to target a execution environment
- OSGi enables easy consumption of new profiles

<table>
<thead>
<tr>
<th>java.lang</th>
<th>java.sql</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>javax.sql</td>
</tr>
<tr>
<td>java.nio</td>
<td>javax.sql.rowset</td>
</tr>
<tr>
<td>java.text</td>
<td>javax.sql.rowset</td>
</tr>
<tr>
<td>java.math</td>
<td>javax.security.auth.kerberos</td>
</tr>
<tr>
<td>java.net</td>
<td>org.ietf.jgss</td>
</tr>
<tr>
<td>javax.net</td>
<td>javax.script</td>
</tr>
<tr>
<td>java.util</td>
<td>javax.xml.crypto</td>
</tr>
<tr>
<td>java.util.logging</td>
<td>java.util.prefs</td>
</tr>
<tr>
<td>java.security</td>
<td>javax.security.sasl</td>
</tr>
<tr>
<td>javax.security.security</td>
<td>java.security.acl</td>
</tr>
<tr>
<td>javax.lang.instrument</td>
<td>java.lang.instrument</td>
</tr>
<tr>
<td>javax.annotation.processing</td>
<td>javax.annotation.processing</td>
</tr>
<tr>
<td>javax.lang.model</td>
<td>javax.lang.model</td>
</tr>
<tr>
<td>javax.lang.model.element</td>
<td>javax.lang.model.element</td>
</tr>
<tr>
<td>javax.lang.model.type</td>
<td>javax.lang.model.type</td>
</tr>
<tr>
<td>javax.lang.model.util</td>
<td>javax.lang.model.util</td>
</tr>
<tr>
<td>javax.tools</td>
<td>javax.tools</td>
</tr>
</tbody>
</table>
Multi-tenancy

- JVMs evolution to support *isolation* within a single JVM
  - Single copy of code, multiple copy of static variables
  - Resource management within isolation context
- The drive to ultra light tenants – 10s of K vs MBs per tenant safely

One copy of code, many tenants
Sharing

- J9 JVMs using sharing to reduce costs
  - Shared classes cache for read-only shared artifacts (bytecodes)
  - Dynamic AOT - reuse JIT code from multiple JVMs
  - Reduce memory use by 20%, improve startup time 10-30%

- Cloud?
  - 100-500 JVMs starting up at the same time

Compile code once, run faster!

JVM shared index

Verified bytecodes

AOT – Ahead Of Time
(JIT code saved for next JVM)
OSGi and Multi-tenancy

- Shared Code is critical to size reductions
  - Key to sharing is correctly identifying exact same versions, shapes across multiple uses boundaries

- Multi-tenancy use cases
  - Single copy of code, with multiple copies of statics (these used to be called DLLs)
  - Isolation of state within a single JVM
  - Bundles have key properties that could help improve this usecase (Versions, Lifecycle)

- OSGi as an enabler for converting modules to support various multi-tenant usage models?
  - Single copy of application code, private per tenant data
  - Single copy or (master) middleware managing resource across multiple tenants
  - Lifecycle phases to support master-initialize, tenant-initialize, tenant-stop, tenant-restart
  - Module boundary for memory use, resource use, security
Today: Isolating Statics Through Class Loaders

- Isolation through the “usual” class loader tricks
  - Every time you load a tenant, use a new class loader
  - Statics are now no longer shared!
Today: Isolating Statics Through Class Loaders

- Isolation through the usual class loader tricks
  - Every time you load a tenant, stick it in a new class loader
  - Statics are now no longer shared!

- Plenty of duplication!
  - Class data
  - JITted code
  - Duplicated static variables, most of which are likely identical
Solving the Data Isolation Challenge

- We’ve enabled our JVM to understand isolation contexts and show each context a different view of static variables
- Using the @TenantScope annotation.

@TenantScope Semantics: Static variable values are stored per-tenant
Each tenant has their own LocaleSettings.defaultLocale
Now many tenants can share a single LocaleSettings class
Can we support isolating statics with OSGi?

- Enhance OSGi to enable static state isolation
  - Leverage JVM support for isolation of state
  - Bundles with new sharing options ala @TenantScope?
  - New Lifecycle (attach / detach) to tenant?

### Diagram

- **Master Tenant**
  - Class Loader
  - Class X

- **Tenant₁**
  - Statics for A

- **Tenant₁**
  - Statics for A

- **Tenant₃**
  - Statics for A

- **Bundle**
  - Class Loader
  - Class A

### Notes

- No code duplication
  - Single copy of class data
  - Single copy of JITted code
  - Optional shared state across bundles

### JVM Magic

JVM "isolates" - each gets it’s own static variables
Dynamic Behavior

- Dynamic memory resize, processor reallocation and app migration
  - JVM will react in real time to resource events
  - Integration across JVM/OS/HV for best performance
OSGi – Supporting more Dynamic Behaviour with Reconfiguration

- Intelligent runtimes
  - Automatically scale to the right resource levels and dynamically adjust for QoS needs
  - Coordination across JVM, Hypervisors and Operating Systems
  - OSGi already has the concept of lifecycle
    - start, stop
  - Extend lifecycle to support a new events common for Cloud
    - Attache/Detach Tenant
    - Memory resized by the OS or hypervisor
    - vCPU allocation changes
    - Application is being live migrated to new location

- Dynamic Modular Profiles
  - Modularity for right sized platforms, fast start, smaller footprint
  - Self-install and configure just the right components
Java is a team sport

- Java is still a community effort
  - Eclipse, Apache and even OpenJDK

- Java Innovations hand in hand with Eclipse and OSGi innovation
  - Cloud: multi-tenancy, sharing, dealing with dynamic behaviour
  - Compatibility: new use cases for leveraging semantic versioning
  - Performance: Drive optimization benefits from explicit dependencies model
  - Software Engineering discipline paying dividends
IBM.says("Thank you");