
Java is a trademark of Sun Microsystems, Inc.

JavaOne℠

# JSR 326 – Diagnosing Deadly Java™ Platform Problems

Steve Poole
IBM

# Todays BOF

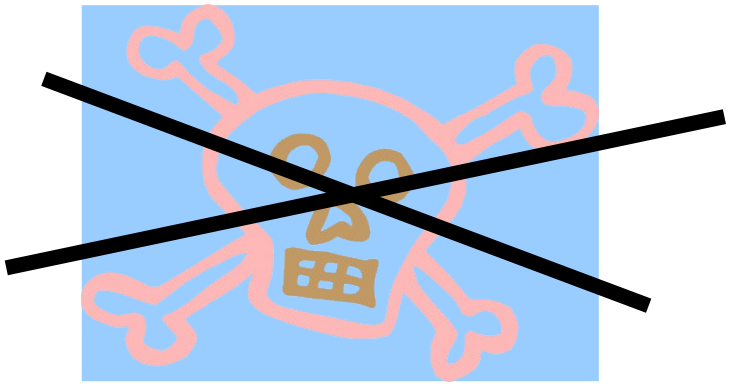
- Its a discussion – feel free to ask questions
- Its about
  - Diagnosing Deadly Java Platform Problems
  - the problem space
    - Why “one size fits all” - doesn't
    - Why size matters
  - JSR 326 and Apache Kato
    - What does “Post Mortem mean”
    - Objectives
    - What the API looks like now
  - “Demoettes”
  - Next steps

# JSR 326 – Post Mortem JVM Diagnostics API

- Raised by IBM
- Supported by Sun, Oracle , Intel , Eclipse , HP , SAP, Nortel , ASF
- Expert Group consists of
  - IBM , Sun , Intel , Oracle , dyna Trace
  - 2 independents
- First Early Draft Review targeted for July 31 2009
- Looking to complete JSR by 1Q 2010

# What does Post Mortem Mean?

- Post Mortem means "*after the fact*"
  - Dead JVMs are not a pre-requisite
  - Think "snap shot"



# Problem Space

- Limited options for diagnosing problems
  - Especially intermittent or unexpected problems
- Outside “Live Monitoring” there is no standard way to get diagnostic information
  - (Even “Live Monitoring” has issues)
- Tools space is fragmented
  - Analysing “Out of Memory” problems is OK
  - Most tools are JVM specific
  - System.out.println is a common tool

# Problem Space

- Lack of standard  post mortem API is steadily driving problem solving down the stack
- Industry spends significant resource diagnosing customer application problems
- Emerging trends indicate this is going to get worse

# “Emerging trends indicate this is going to get worse”

- Increase of multiple cores: 2,4,8,16,32,64,128...
- Growth of GB memory sizes: 1,2,4,8,100,1000...
- New languages : Ruby,Python,PHP …
- New capabilities: NIO, Shared Classes …
- Wider audience – clouds , billions of devices?
  - Even “Live Monitoring” is effected – size of system and rate of change makes analysis and prediction increasing difficult...

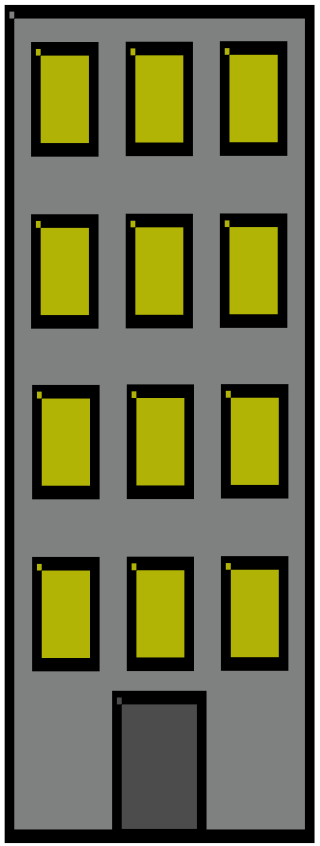
# Finally...

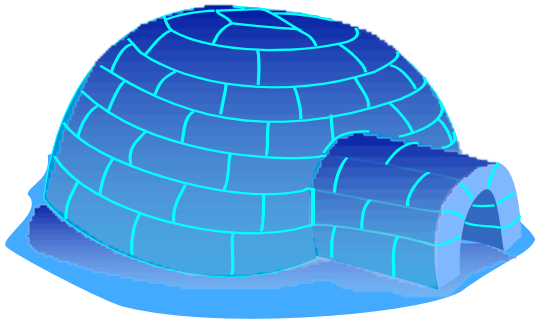
- We can't ask users to move to latest and greatest Java to get better diagnosics
- We must improve diagnostics across the board
  - Even if it's on a “best can do” basis
  - Using whatever data is available

# JSR 326 & Apache Kato
## Objectives

- Create a standard Java API to enable new tools which can interact seamlessly with the various diagnostic output from multiple Java Virtual Machines.
- API must standardize triggering and consumption of post-mortem diagnostic artefacts
- Develop API specification, reference implementation and TCK as Open Source so every one can participate
  - Use “user stories” and demonstration tools to keep the API real , credible and consumable

# JSR 326 – just what are we designing?





# JSR 326 – just what are we designing?

- A single “one size fits all” API approach is not sensible
  - Diagnostic artifacts are produced for various reasons and with various contents
  -
- Users have differing requirements
- 2 broad categories
  - Data Visualization: “show me everything”
  - Situation Analysis: look for specific problems, snapshot monitoring
  -
  -

# •Practical “TechDemos”

| Tool type or problem area | Interactive (GUI) or batch (CLI) | Startup time or processing time | Random or serial access | Lightweight or heavyweight |
|---|---|---|---|---|
| Java Debug Interface connector (for example, using Eclipse to debug an artifact) | Interactive | Startup | Random | Lightweight |
| Artifact explorers | Interactive | Startup | Random | Lightweight |
| Native-memory analysis | Batch | Processing | Serial | Heavyweight |
| Trend analysis | Batch | Startup | Random | Heavyweight |
| Artifact analysers that detect potential problems | Batch | Processing | Serial | Heavyweight |
| Java heap analyser | Interactive | Startup | Serial | Lightweight |

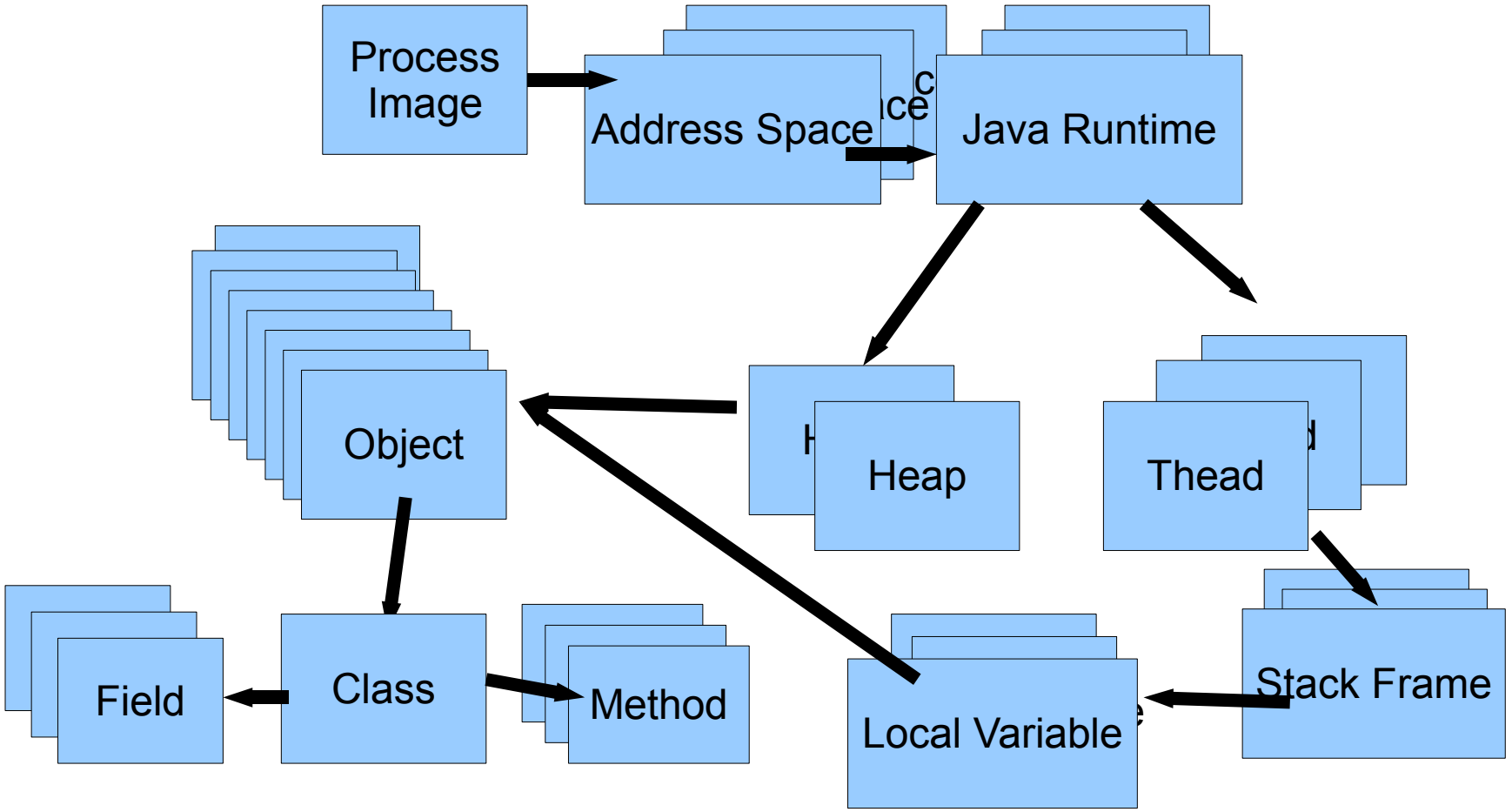
# Can we help ?

- Why are there so many http connections being queued?
- Which monitor is locking all threads ?
- How can I detect which class is consuming a lot of memory?
- Where is the native memory going?
- Why did I run out of sockets?

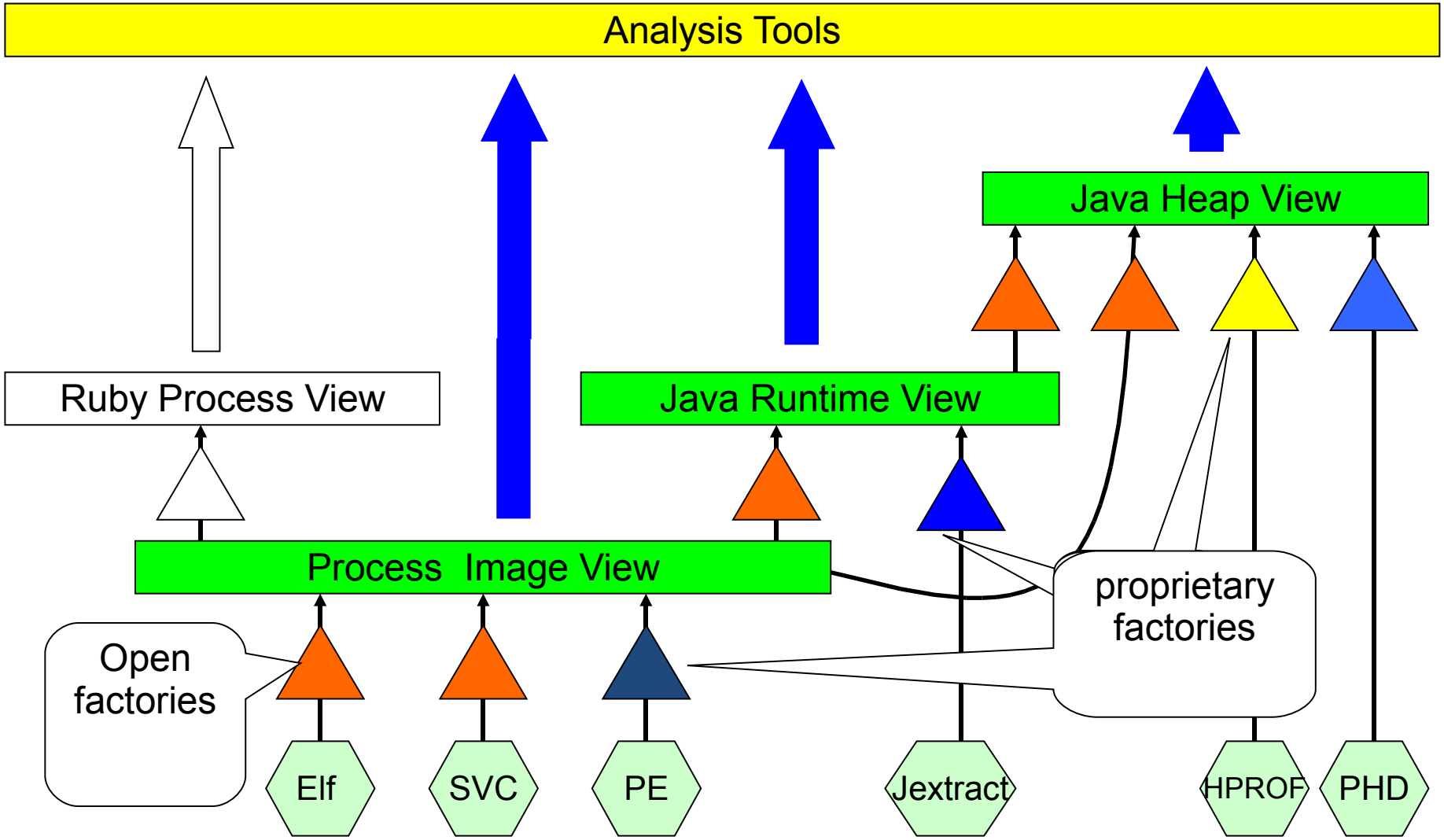
- Have we already seen this problem?
- Deadlock analysis
- Native locks
- Non responsive sockets
- What made my JNI program crash?
- Help me resolve this IndexOutOfBounds Exception

# The API today

# Basic API structure today



# Possible API architecture



# Getting started

- Central registry using javax.imageio.spi.ServiceRegistry
  - Auto discovery
  - Use the registry to get the Process Image
  -

```
FactoryRegistry registry=FactoryRegistry.getDefaultRegistry();
```

-

```
File atifact=new File(“path to artifact”)

Image image=registry.getImage(artifact)
```

# A sampler – what we can do right now

- Multiple JVM support
- Data Visualisation
- Simple queries
- Deadlock analysis

- JVM support Demoette

# •JVM support Demoette

- HPROF support available
- Proprietary support for IBM core files coming
- Experimenting with alternative forms of artifact
- Exploring the API: not all data is available
  - Can be missing for various reasons
  - API has to deal with optional/missing data

- Exploring the API some more

Why do this?

```
For thread in runtime.getThreads()
    For stackframe in thread.stackFrames()
        For var in stackframe.getVariables()
            Print var.name
```

When you can do this? - katoview

*threads/stackframes/variables/name*

- Exploring the API some more

KatoView – command line exploration tool

Shows you can extract the data you need (almost)

Deadlocks,
Queries,
Application specific data

Not the only way to explore though - how about a more familiar way?

# JDI Connector

- Lets you view standard dumps in a familiar way
- New experimental diagnostic artifacts will expose local variable data too

# Native memory problems...

- Some things are relatively straight forward
  - List of native memory allocations
    - Just a matter of understanding malloc!
    - * Per platform * per memory allocation system * time
- What you do with the data?
  - Conservative scan of the image can help find potential owners
  - For NIO we “know” where the allocation is held
    - So MAT for NIO Native Memory is feasible
- BUT – generally, unless the allocator participates, its always going to be hit and miss.

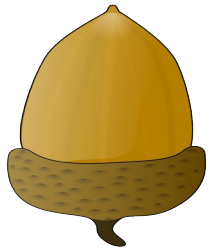
# Can we help? - some now, some later , some much later

- Why are there so many http connections being queued?
- Which monitor is locking all threads ?
- How can I detect which class is consuming a lot of memory?
- Where is the native memory going?
- Why did I run out of sockets?

- Have we already seen this problem?
- Deadlock analysis
- Native locks
- Non responsive sockets
- What made my JNI program crash?
- Help me resolve this IndexOutOfBounds Exception

# Summary

- API development is still underway
  - Have a good foundation
  - Lots of potential
  - Still much to do



# Where next

- Visit the Apache Kato website
- Contribute to the open source project
  - Participate on the mailing list
  - Help develop the project
    - Help us write better tools (or write your own)
    - Tell us what problems you want us to tackle

# Questions

?


# Thank You

Steve Poole
spoole@uk.ibm.com
kato-spec@incubator.apache.org

http://incubator.apache.org/kato/