JavaOne™

JavaOne™

Java Flight Recorder Behind the Scenes

Staffan Larsen
Java SE Serviceability Architect

staffan.larsen@oracle.com
@stalar

MAKE THE
FUTURE
JAVA

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Java Flight Recorder

- Tracer and Profiler
- Non-intrusive
- Built into the JVM itself
- On-demand profiling
- After-the-fact capture and analysis

- First released in 7u40

Photograph by Jeffrey Milstein
http://butdoesitfloat.com/In-the-presence-of-massive-gravitational-fields-this-perfect

Tracer and Profiler

- Captures both JVM and application data
  - Garbage Collections
  - Synchronization
  - Compiler
  - CPU Usage
  - Exceptions
  - I/O
- Sampling-based profiler
  - Very low overhead
  - Accurate data

Non-Intrusive

- Typical overhead in benchmarks: 2-3% (!)
- Often not noticeable in typical production environments
- Turn on and off in runtime
- Information already available in the JVM
  - Zero extra cost

Photograph: Andrew Rivett
http://www.flickr.com/photos/veggiefrog/3435380297/

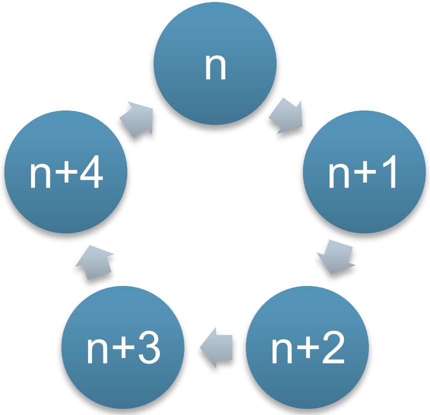
Built into the JVM itself

- Core of JFR is inside the JVM
- Can easily interact with other JVM subsystems
- Optimized C++ code
- Supporting functionality written in Java

On-Demand Profiling

- Start from Java Mission Control
  - Or from the command line
- Easily configure the amount of information to capture
- For a profile, a higher overhead can be acceptable
- When done, no overhead
- Powerful GUI for analysis

After-the-Fact Analysis



- In its default mode, very low overhead
- Designed to be always-on
- Uses circular buffers to store data
  - In-memory or on-disk
- When an SLA breach is detected, dump the current buffers
- Dump will have information **leading up to** the problem

DEMO

Agenda

- Overview of JFR
- Demo!
- Configuration topics
- Implementation details

Configuration

- Enable

```
-XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

- Start

```
-XX:StartFlightRecording=filename=<path>,duration=<time>
```

- Or

```
jcmd <pid> JFR.start filename=<path> duration=<time>
```

Advanced Configuration

Per Recording Session	
Max age of data	**maxage=**<time>
Max size to keep	**maxsize=**<size>

Global Settings (-XX:FlightRecorderOptions)	
Max stack trace depth	**stackdepth=**<n> (default 64)
Save recording on exit	**dumponexit=**true
Logging	**loglevel=**[ERROR|WARN|INFO|DEBUG|TRACE]
Repository path	**repository=**<path>

Recording Sessions

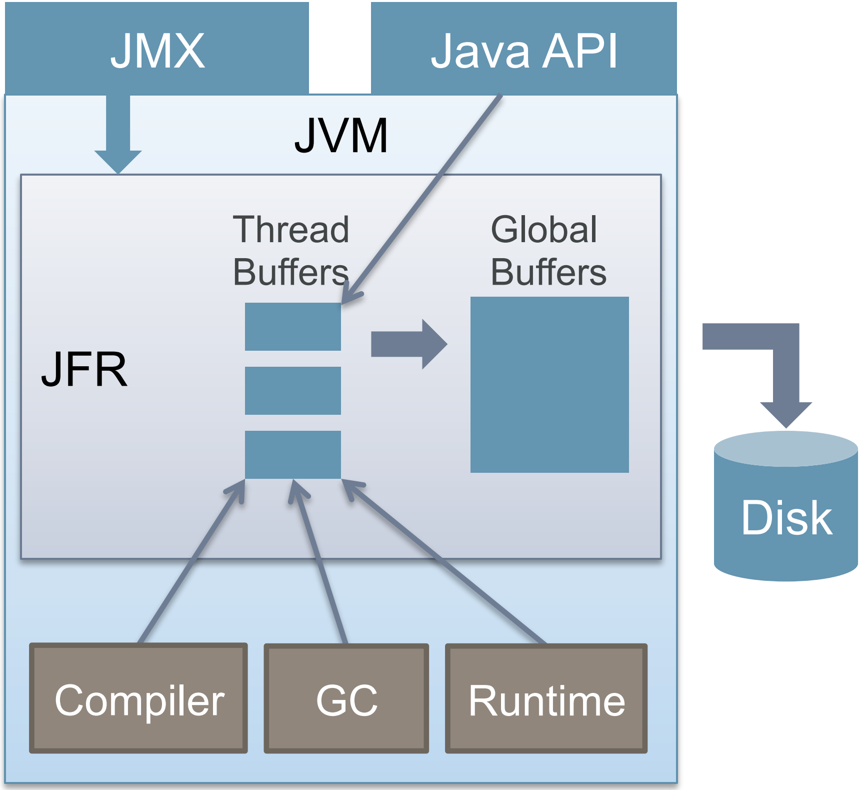
- Recordings can specify exactly which information to capture
  - ~80 events with 3 settings each
- But: two preconfigured settings
  - “default”: provides as much information as possible while keeping overhead to a minimum
  - “profile”: has more information, but also higher overhead
- You can configure your own favorites in Mission Control

Many Simultaneous Recording Sessions

- This works great
- Each session can have its own settings
- Caveat: If there are multiple sessions all of them get the **union** of the enabled events
  - Ex: If event A is enabled in on recording, all recordings will see event A
  - Ex: If event B has two different thresholds, the lower value will apply

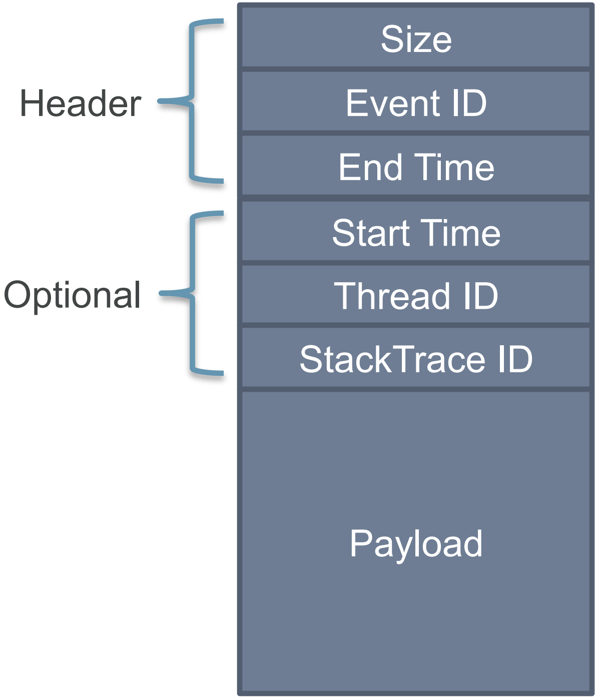
How Is It Built?

- Information gathering
  - Instrumentation calls all over the JVM
  - Application information via Java API
- Collected in Thread Local buffers
  - ⇢ Global Buffers ⇢Disk
- Binary, proprietary file format
- Managed via JMX



“Everything Is an Event”

- Header
- Payload
  - Event specific data



Event Types

- Instant
  - Single point in time
  - Ex: Thread starts
- Duration
  - Timing for something
  - Ex: GC
- Requestable
  - Happens with a specified frequency
  - Ex: CPU Usage every second

Event Meta Data

- For every event
  - Name, Path, Description
- For every payload item
  - Name, Type, Description, Content Type

“Content Type”

- Describes the **semantics** of a value
- Used to correctly display the value in the UI

Content Type	Displayed as
Bytes	4 MB
Percentage	34 %
Address	0x23CDA540
Millis	17 ms
Nanos	4711 ns

Event Definition in Hotspot

```
<event  id="ThreadSleep"
        path="java/thread_sleep"
        label="Java Thread Sleep" ...>
   <value  field="time"
           type="MILLIS"
           label="Sleep Time"/>
</event>
```

- XML definitions are processed into C++ classes

Event Emission in Hotspot

```
JVM_Sleep(int millis) {
   EventThreadSleep event;

   ... // actual sleep happens here

   event.set_time(millis);
   event.commit();
}
```

- Done! Data is now available in JFR

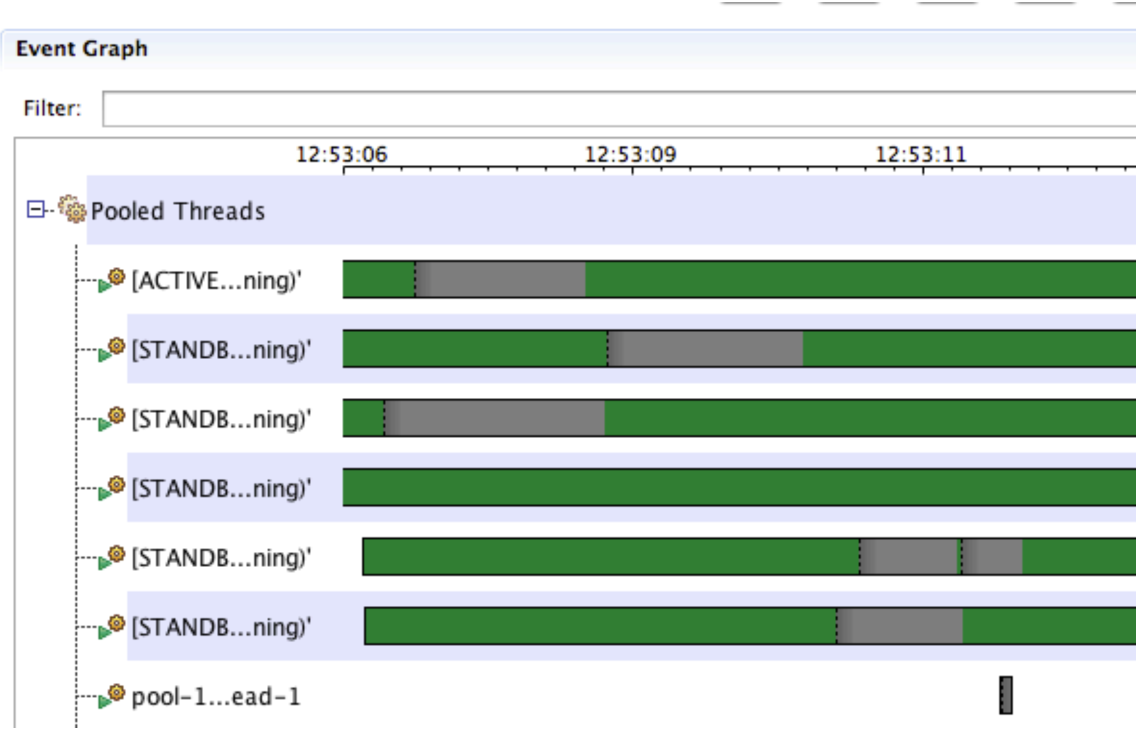
Thread Park

```
<event id="ThreadPark" path="java/thread_park"
      label="Java Thread Park"
      has_thread="true" has_stacktrace="true" is_instant="false">
   <value type="CLASS" field="klass" label="Class Parked On"/>
   <value type="MILLIS" field="timeout" label="Park Timeout"/>
   <value type="ADDRESS" field="address"
          label="Address of Object Parked"/>
</event>
```

Thread Park

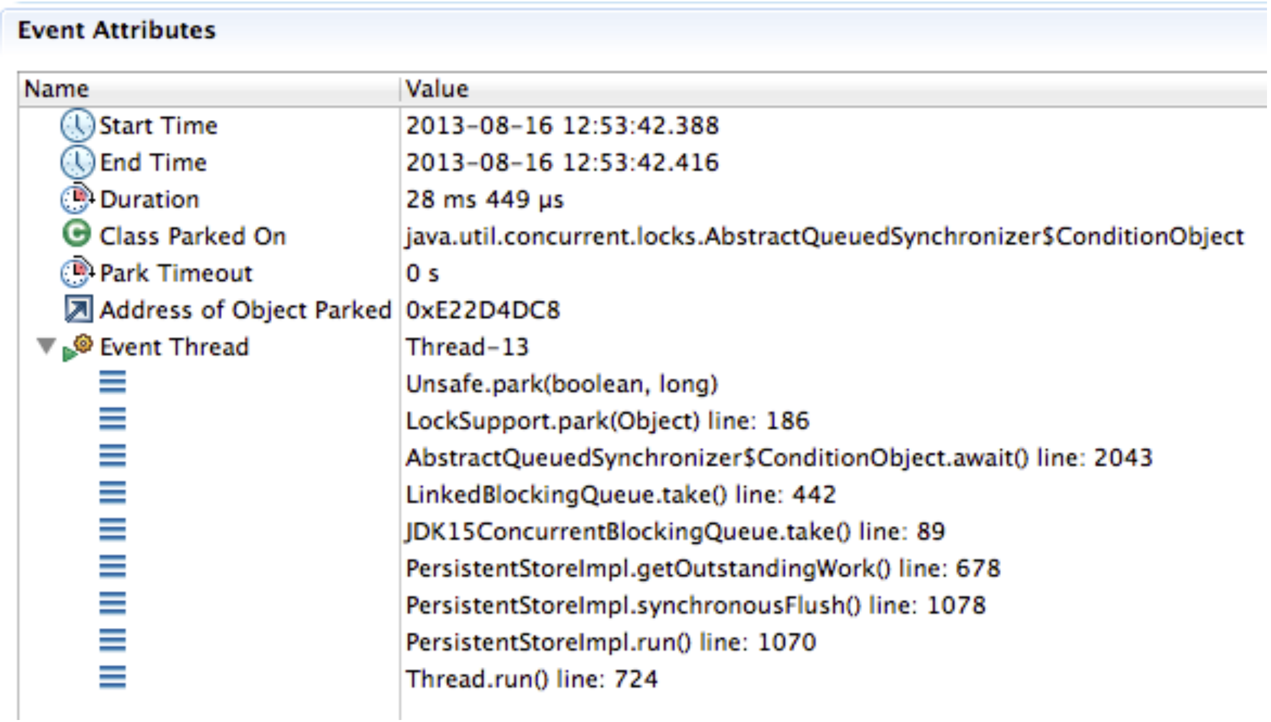
```
UnsafePark(jboolean isAbsolute, jlong time) {
  EventThreadPark event;
  JavaThreadParkedState jtps(thread, time != 0);
  thread->parker()->park(isAbsolute != 0, time);
  if (event.should_commit()) {
    oop obj = thread->current_park_blocker();
    event.set_klass(obj ? obj->klass() : NULL);
    event.set_timeout(time);
    event.set_address(obj ? (TYPE_ADDRESS)obj : 0);
    event.commit();
  }
}
```

Event Graph



Event Graph
Filter:
12:53:06
12:53:09
12:53:11
Pooled Threads
[ACTIVE...ning)'
[STANDB...ning)'
[STANDB...ning)'
[STANDB...ning)'
[STANDB...ning)'
[STANDB...ning)'
pool-1...ead-1

Event Details

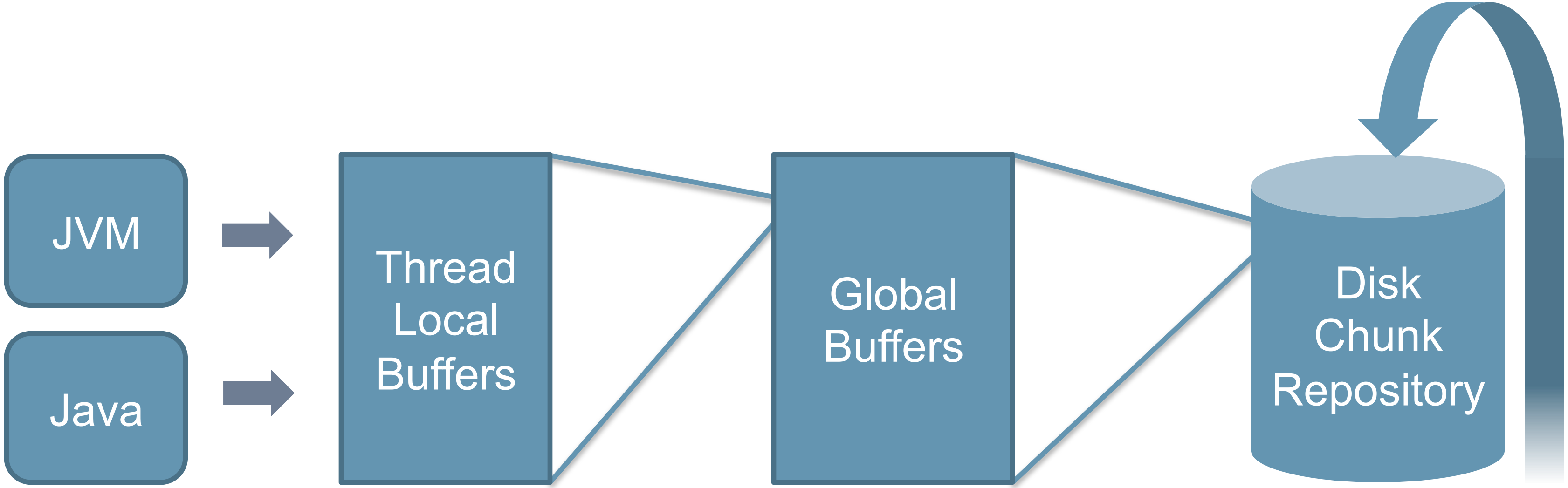


Event Attributes

Name	Value
Start Time	2013-08-16 12:53:42.388
End Time	2013-08-16 12:53:42.416
Duration	28 ms 449 µs
Class Parked On	java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject
Park Timeout	0 s
Address of Object Parked	0xE22D4DC8
▼ Event Thread	Thread-13
☰	Unsafe.park(boolean, long)
☰	LockSupport.park(Object) line: 186
☰	AbstractQueuedSynchronizer$ConditionObject.await() line: 2043
☰	LinkedBlockingQueue.take() line: 442
☰	JDK15ConcurrentBlockingQueue.take() line: 89
☰	PersistentStoreImpl.getOutstandingWork() line: 678
☰	PersistentStoreImpl.synchronousFlush() line: 1078
☰	PersistentStoreImpl.run() line: 1070
☰	Thread.run() line: 724

Buffers

- “Circular”
- Designed for low contention

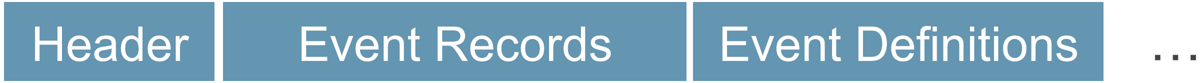


Filtering Early

- Enable/disable event
- Thresholds
  - Only if duration is longer than X
- Enable/disable stack trace
- Frequency
  - Sample every X

File Format

- Self-contained
  - Everything needed to parse an event is included in the file
  - New events instantly viewable in the UI
- Binary, proprietary
- Designed for fast writing
- Single file, no dependencies



Header

Event Records

Event Definitions

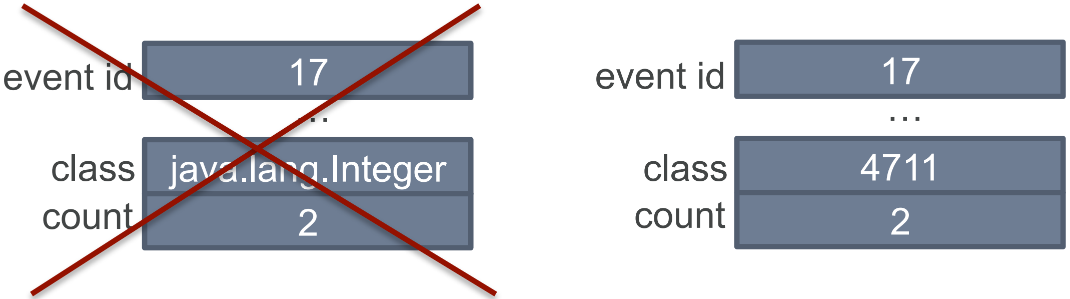
…

Dynamic Runtime and Long-Running Recordings

- Can’t leak memory
  - Can’t aggregate information eternally
  - Can’t keep references that prohibits class unloading
- Dynamic Runtime
  - Classes can come and go
  - Threads can come and go
- Solutions: Constant Pools, Checkpoints

Problem: Many Events Reference Classes

- If every event contained the class name as a string, we would waste lots of space
- Solution: Class IDs



```
class Klass {
    …
    u8 _trace_id;
    …
}
```

Problem: When Do We Write the Class IDs?

- IDs need to be part of the file
- Classes can be unloaded at any time
  - Class may not be around until end of recording
- Solution: write Class ID when classes are unloaded

Problem: Size of the Class List

- Many classes are loaded, not all are referenced in events, we want to save space
- Solution: when a class ID is referenced, the class is also “tagged”
  - Write only tagged classes in the JFR file

```
#define CLASS_USED 1

void use_class_id(Klass* const klass) {
  klass->_trace_id |= CLASS_USED;
}
```

Problem: Leaking Memory

- Over time many classes will be tagged, the size of the class list will increase
- Solution: reset the tags each time a class list is written to disk
- We call this a “Checkpoint”

- A recording file may contain many class lists, each one is only valid for the data immediately preceding it

Constant Pools

- The Class List is a special case of a Constant Pool

- Classes
- Methods
- Threads
- Thread Groups
- Stack Traces
- Strings

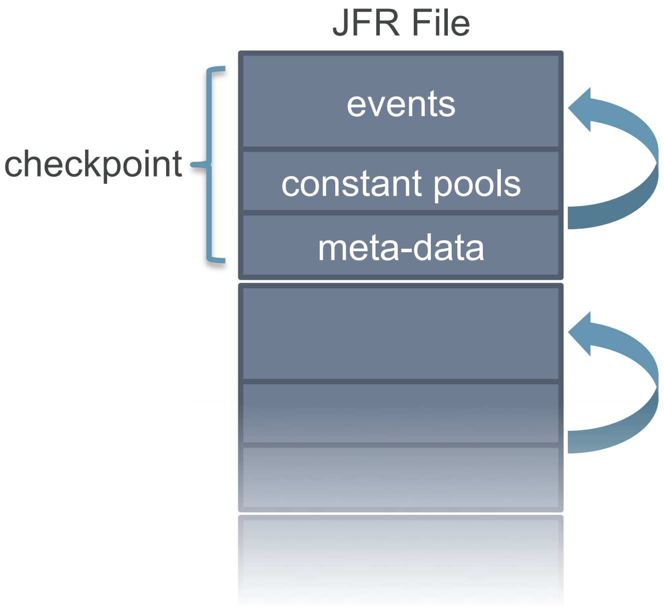
```
class_pool.lookup(4711)
  → java.lang.Integer

method_pool.lookup(1729)
  → java.lang.Math:pow()
```

Checkpoints

- At regular intervals, a “checkpoint” is created in the recording
- Has everything needed to parse the recording since the last checkpoint

checkpoint =
events
+ constant pools
+ event meta-data



Optimizations

- Fast Timestamps
  - Fast, high resolution CPU time where available
  - Invariant TSC instructions

- Stack Traces
  - Each event stores the thread’s stack trace
  - Pool of stack traces

Differences vs. JRockit

- I/O: File path, Socket address
- Exceptions
- Reverse call trace view in Mission Control
- Easier configuration in Mission Control
- Deeper (configurable) stack traces
- Internal JVM differences: GC

More Information

- Whitepaper

  http://www.oracle.com/missioncontrol

- User Guide

  http://docs.oracle.com/javase/7/docs/technotes/guides/jfr/index.html

- Forum

  https://forums.oracle.com/community/developer/english/java/java_hotspot_virtual_machine/java_mission_control

Remember

```
-XX:+UnlockCommercialFeatures
-XX:+FlightRecorder
```

Q&A

@stalar

staffan.larsen@oracle.com

serviceability-dev@openjdk.java.net

JavaOne™