Flight Recording im Application Life Cycle - vom UI bis zum Backend

Wolfgang Weigend
Sen. Leitender Systemberater
Java Technology and Architecture

CREATE
THE
FUTURE

JavaOne
ORACLE

Disclaimer

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.

Agenda

Application Lifecycle

- Java Development
- Java SE und Java EE

Betriebsunterstützung

- Java Versioning & JRE
- Werkzeuge
- Java SE Advanced Support

Development mit Java SE und Java EE

Entwicklungsumgebungen mit Standard Java IDE’s

Build Tool Integration

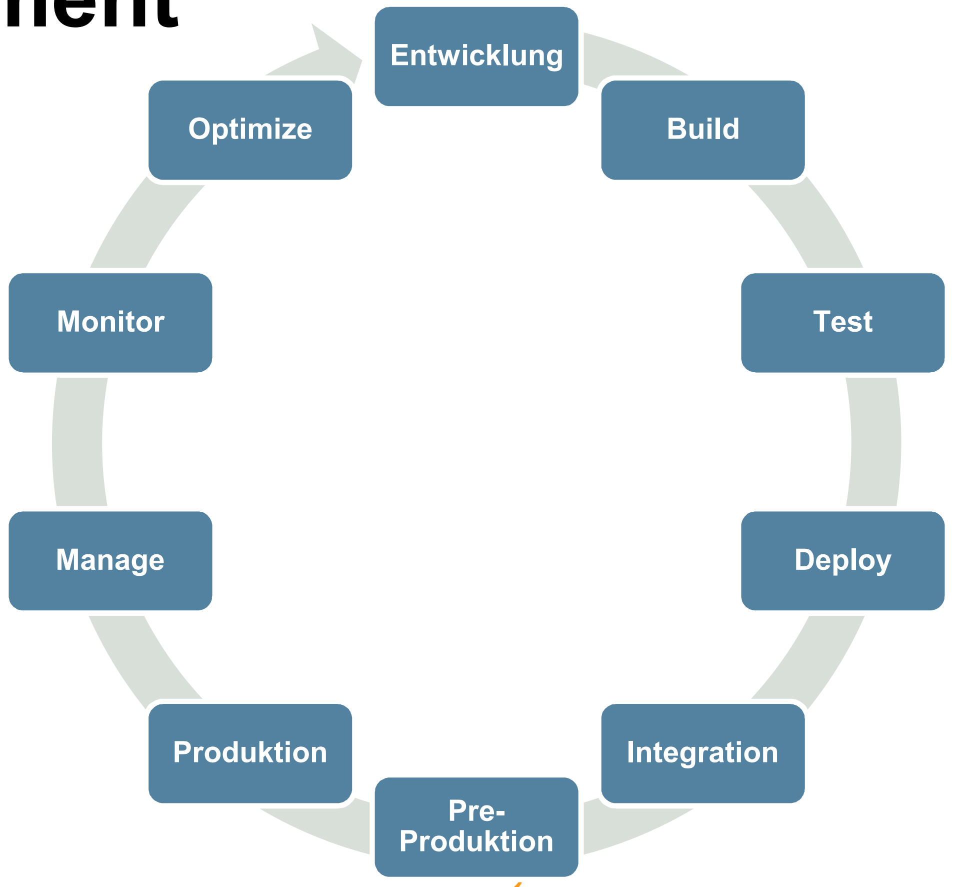
Built in Java VM Flight Recorder und Diagnostics

Frontend Java UI

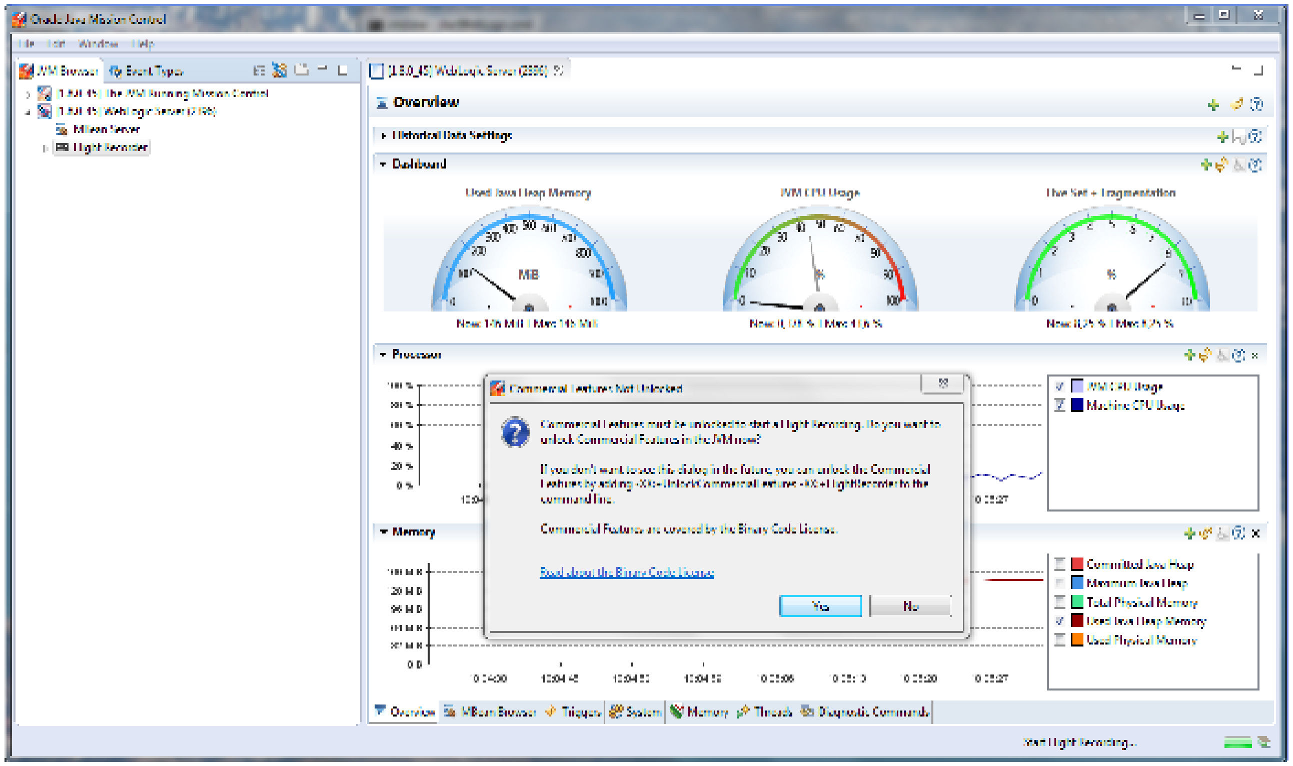
Applikationsserver mit Java EE

Lifecycle Management

- Java development
- Continuous integration
- MOS integration
- Holistic view of application
- lifecycle management
- DevOps
  - Application development and systems operation



Java Mission Control 5.5



Commercial Features Not Unlocked

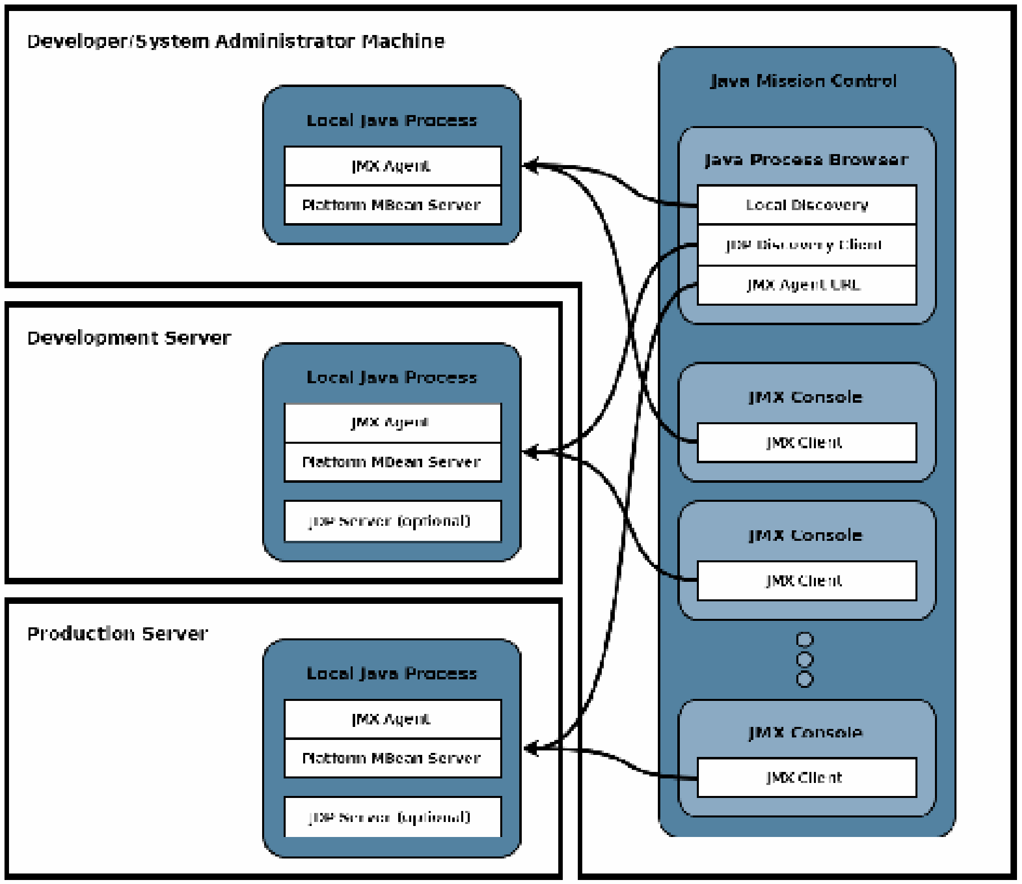
Commercial Features must be unlocked to start a Flight Recording. Do you want to unlock Commercial Features in the JVM now?

If you don't want to see this dialog in the future, you can unlock the Commercial Features by adding -XX:+UnlockCommercialFeatures -XX:+FlightRecorder to the command line.

Commercial Features are covered by the Binary Code License.

Read about the Binary Code License

Java Process Browser and JMX Console



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

Java Flight Recorder - Tracer & Profiler

- Captures both JVM and application data
  - Garbage Collections
  - Sychronization
  - 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

Built into the JVM itself

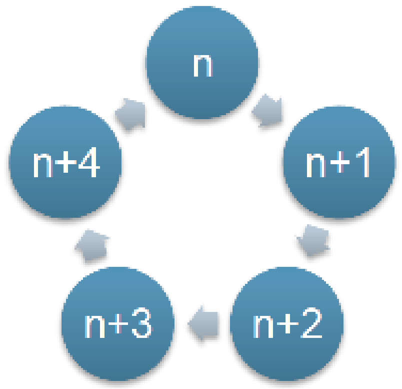
- Core of JFR is inside the JVM
- Can easily interact with other JVM subsystems
- Optimized C++ code
- Supporting functionality written in Java

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

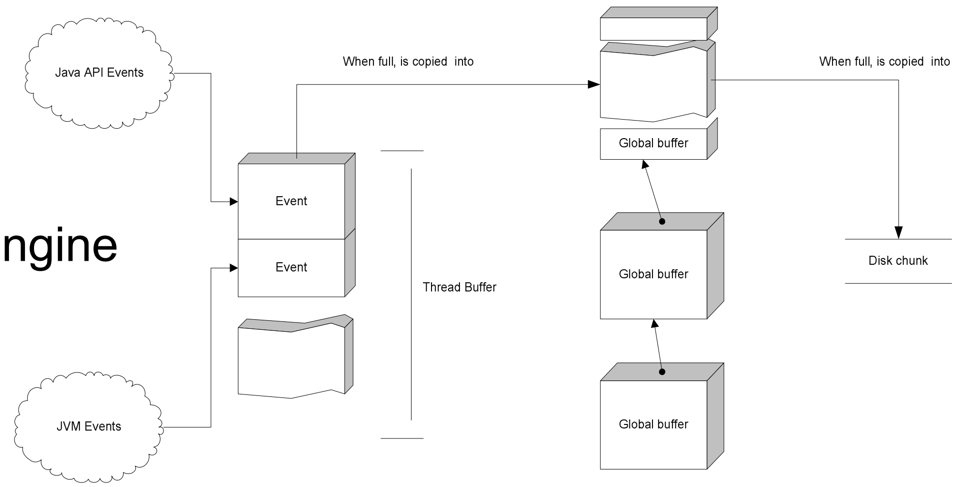
- In its default mode, very low overhead
- Designed to be always-on
- Uses circular buffers to store data
  - In-memory or on-disk
- When an SLA breach is detected, dump the current buffers
- Dump will have information leading up to the problem



Flight Recorder - Arbeitsweise

Focus on performance

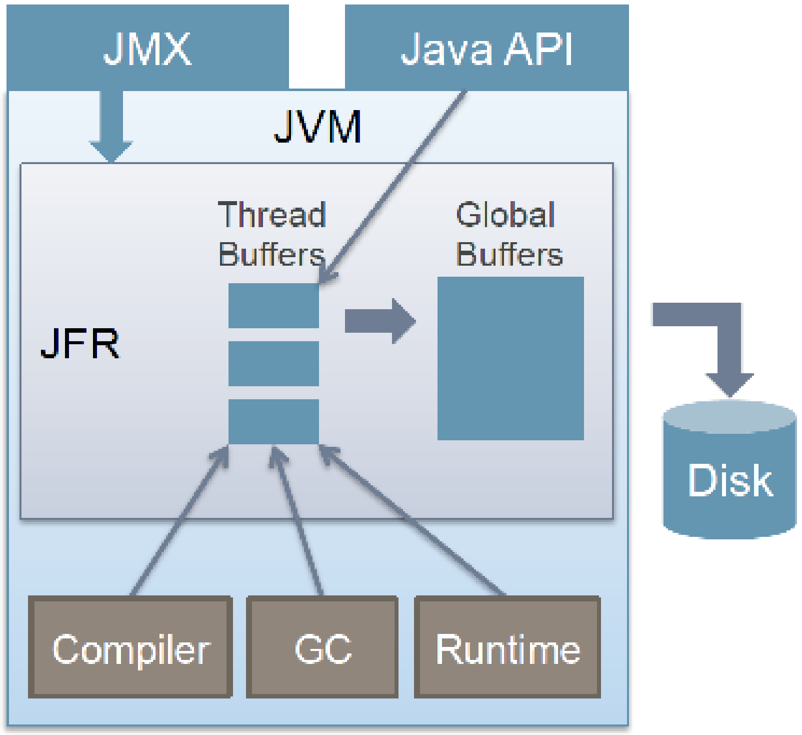
- Extremely low overhead
  - Using data already gathered
  - High performance recording engine
- Testing
- Third party events
  - WLS
  - Dynamic Monitoring Service (DMS) custom WLST commands
  - JavaFX
  - You can add your own events, but this is not yet supported!



Java Flight Recorder – How is it built?

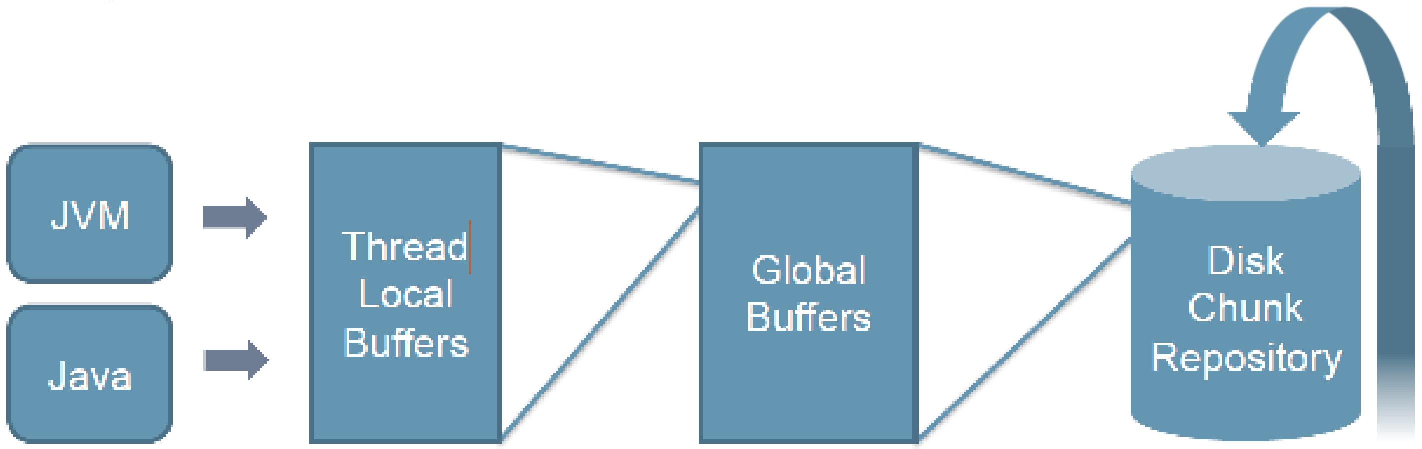
- Information gathering
  - Instrumentation calls all over the JVM
  - Application information via Java API
- Collected in Thread Local buffers

  ⇢ Global Buffers ⇢Disk
- Binary, proprietary file format
- Managed via JMX
- Java Flight Recorder
  - Start from JMC 5.5 or CLI
- Activate Flight Recorder
  - -XX: +UnlockCommercialFeatures
  - -XX: +FlightRecorder



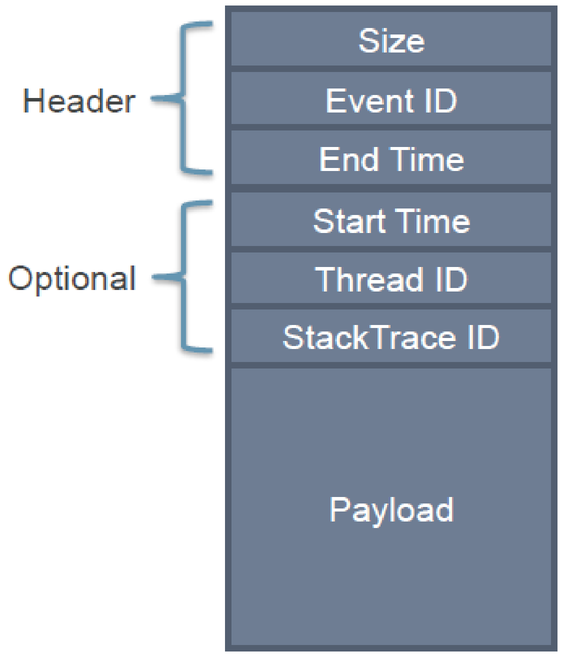
Java Flight Recorder – Buffers

- “Circular”
- Designed for low contention



“Everything is an Event”

- Header
- Payload
  - Event specific data



Event Types

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

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();
}
```

- Now the data will be available in JFR

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
  - Example: 5 Minutes of recording,
    result in a Flight Recorder File with a size of 957 KB

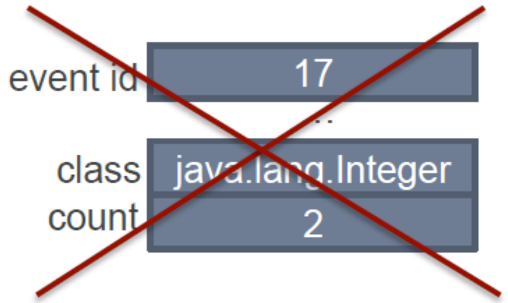
Header	**Event Records**	**Event Definitions**

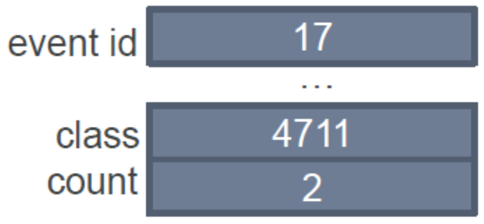
Dynamic Runtime and Long-Running Recordings

- 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 Sample: Many Events Reference Classes

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





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

Problem Sample: When do we write the Class IDs?

- ID’s 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 Sample: 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

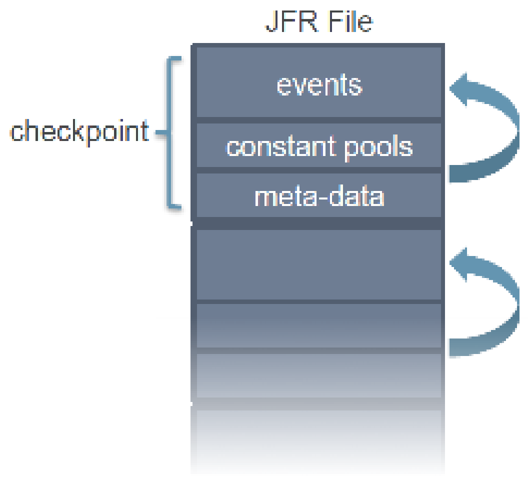
```
class_pool.lookup(4711)
  → java.lang.Integer

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



Java Flight Recorder Releases

- Resulted from the JRockit and HotSpot JVM Convergence
- Java Mission Control released with the 7u40 JDK
- Java Mission Control 5.4.0 released with JDK 8u20
- Java Mission Control 5.5.0 released with JDK 8u40
  - Dynamic enablement of Flight Recorder (no startup flags needed)
  - Hiding of Lambda Form methods

Zusammenfassung

- Java Flight Recorder liefert eine gemeinsame Sicht auf die JVM und die Java-Applikation
  - JVM Events und Java API Events
- Extremely low overhead (<= 2%)
  - Can keep it always on, dump when necessary
- Tooling for analysing recordings built into the Oracle JDK via Java Mission Control
- Java APIs available for recording custom information into the Flight Recorder in the Oracle JDK
- Third party integration giving holistic view of the detailed information recorded by the Flight Recorder (WebLogic Server, JavaFX)

Danke!

Wolfgang.Weigend@oracle.com

Twitter: @wolflook