Continuous Monitoring with JDK Flight Recorder

@MikaelVidstedt
Director, Java Virtual Machine
Java Platform Group, Oracle
Safe Harbor

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, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle’s future plans, expectations, beliefs, intentions and prospects are “forward-looking statements” and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle’s Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading “Risk Factors.” These filings are available on the SEC’s website or on Oracle’s website at http://www.oracle.com/investor. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.
FLIGHT RECORDER
DO NOT OPEN
Agenda

Overview: What is JDK Flight Recorder (JFR)?
JFR Events
Designed for use in Production
Using JFR
Future Work
What is JDK Flight Recorder?
JFR In a Nutshell

JFR = JDK Flight Recorder
Available now, in a JDK near you!

An event based tracing framework
Built into the Java Runtime
Extremely low overhead, suitable for production environments
Allows correlation of data from different subsystems/software layers
With APIs for
  Producing application level events
  Consuming event streams
Demo: Simple Monitoring
History

200x

JRockit
History

JDK 7u4 - 2012
Initial Hotspot version
(Oracle internal use only)

JDK 9 - 2017
Public APIs for creating and consuming data

JDK 11 - 2018
Open Sourced!
The Anatomy of a JFR Event

- Event ID
- Timestamp (CPU ticks)
- Duration (CPU ticks)
- Thread ID
- StackTrace ID
- Event Specific Payload
import jdk.jfr.Event;

class MyEvent extends Event {
}
The Anatomy of a JFR Event

```java
import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        // do important stuff here
    }
}
```
import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        MyEvent e = new MyEvent();
        e.begin();
        // do important stuff here
        e.end();
        e.commit();
    }
}
The Anatomy of a JFR Event

import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        MyEvent e = new MyEvent();
        e.begin();
        // do important stuff here
        e.end();
        e.commit();
    }
}
The Anatomy of a JFR Event

```java
import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        MyEvent e = new MyEvent();

        e.begin();

        // do important stuff here

        e.end();
        e.commit();
    }
}
```
The Anatomy of a JFR Event

```java
import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        MyEvent e = new MyEvent();
        e.begin();
        // do important stuff here
        e.end();
        e.commit();
    }
}
```
import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        MyEvent e = new MyEvent();
        e.begin();

        // do important stuff here

        e.end();
        e.commit();
    }
}
The Anatomy of a JFR Event

```java
import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        MyEvent e = new MyEvent();
        e.begin();
        // do important stuff here
        e.end();
        e.commit(); // implicit end()
    }
}
```
The Anatomy of a JFR Event

```java
import jdk.jfr.Event;

class MyEvent extends Event {
    void doThing() {
        MyEvent e = new MyEvent();
        e.begin();
        // do important stuff here
        e.commit();
    }
}
```
import jdk.jfr.Event;

class MyEvent extends Event {
    String message;
    int value;
}

void doThing() {
    MyEvent e = new MyEvent();
    e.message = "Hello";
    e.value = 4711;
    e.begin();
    // do important stuff here
    e.commit();
}
The Anatomy of a JFR Event

import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;

@Name("com.oracle.foo.CoolThing")
@Label("Cool Thing")
class MyEvent extends Event {
    @Label("Message")
    String message;

    @Label("Value")
    int value;
}

void doThing() {
    MyEvent e = new MyEvent();
    e.message = "Hello";
    e.value = 4711;
    e.begin();
    // do important stuff here
    e.commit();
}
## JFR Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Name</td>
<td>Set explicit name. Recommended for all event classes. Recommended format: [org</td>
<td>com</td>
</tr>
<tr>
<td>@Label</td>
<td>Human readable name</td>
<td>N/A</td>
</tr>
<tr>
<td>@Description</td>
<td>More detailed description (~1-2 sentences)</td>
<td>N/A</td>
</tr>
<tr>
<td>@Category</td>
<td>Category to which this event logically belongs</td>
<td>N/A</td>
</tr>
<tr>
<td>@Threshold</td>
<td>Default minimum duration for the event to be included in the recording</td>
<td>0 ns</td>
</tr>
<tr>
<td>@Enabled</td>
<td>Controls whether the event should be enabled by default</td>
<td>true (enabled)</td>
</tr>
<tr>
<td>@StackTrace</td>
<td>Controls whether the stack trace should be included in the event by default</td>
<td>true (enabled/included)</td>
</tr>
</tbody>
</table>

Note: List is not exhaustive, see jdk.jfr.* javadoc for more annotations and information.
Events Generated by the Java Runtime

~140 event types in Java Runtime (and growing)

<table>
<thead>
<tr>
<th>Category</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>Command line</td>
</tr>
<tr>
<td></td>
<td>JDK Version Information</td>
</tr>
<tr>
<td></td>
<td>OS</td>
</tr>
<tr>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td>Java Execution</td>
<td>I/O: File &amp; Network</td>
</tr>
<tr>
<td></td>
<td>Thread Sampling</td>
</tr>
<tr>
<td>JVM Operations</td>
<td>Class Loading</td>
</tr>
<tr>
<td></td>
<td>GC</td>
</tr>
<tr>
<td></td>
<td>JIT Compiler</td>
</tr>
</tbody>
</table>
Behind the Scenes: Event Data Flow

1. When full, is copied into Global buffer.
2. Global buffer is copied into Disk chunk once per second, or when full.
3. JVM Events and Java API Events are processed through the Event Flow.
4. Event streaming is used for real-time processing.

1 with event streaming.
The JFR File Format

Compact binary format
Varint 128 LEB encoding (JDK 9+)
Self-describing

Metadata information describing how to interpret events
Data necessary for resolving the preceding events
Event Filtering

Events can be filtered by
Type / Name
Duration
Event Correlation

Events from **multiple levels** of the stack in the same stream
Application, Java Runtime libraries, JVM, OS

Enables powerful in-depth analysis
Start on high level, go as deep as needed
Designed for Use in Production
Designed for Production

Designed from the start **for use in production**

*Extremely* low overhead
Piggy-backs on JVM operations
Events generated into thread-local buffers

**Default on** in Oracle Fusion applications
Several large companies use JFR **extensively**
But what about performance...?
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();
    e.begin();
    // do important stuff here
    e.commit();
}
```
But what about performance...?

// Warning: pseudo-code – this is NOT what commit() actually looks like!!
void Event::commit() {
    if (isEnabled()) {
        // now() reads CPU clock register
        long duration = now() - startTime;
        if (duration > THRESHOLD) {
            if (shouldCommit()) {
                // Cheap – Thread local writes
                actuallyCommit();
            }
        }
    }
}
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();
    e.begin();
    // do important stuff here
    e.commit();
}
```
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();
    e.startTime = now();
    // do important stuff here
    e.commit();
}
```
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();

    e.startTime = <JVM intrinsic>;

    // do important stuff here

    e.commit();
}
```
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();
    e.startTime = <JVM intrinsic>;

    // do important stuff here

    if (e.isEnabled()) {
        // perform additional checks and possibly call actuallyCommit()
    }
}
```
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();

    e.startTime = <JVM intrinsic>;

    // do important stuff here

    if (false) {
        // perform additional checks and possibly call actuallyCommit()
    }
}
```
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();

    e.startTime = <JVM intrinsic>;

    // do important stuff here
}
```
But what about performance...?

```java
void doThing() {
    MyEvent e = new MyEvent();
    long startTime = <JVM intrinsic>;
    // do important stuff here
}
```
But what about performance...?

```java
void doThing() {
    long startTime = <JVM intrinsic>;

    // do important stuff here
}
```
But what about performance...?

```java
void doThing() {
    // do important stuff here
}
```
Performance (YMMV)

Logging Cost (ns/operation, lower is better)

<table>
<thead>
<tr>
<th>Setting</th>
<th>ns/operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFR Disabled</td>
<td>0</td>
</tr>
<tr>
<td>JFR Enabled (stackdepth=1)</td>
<td>1,400</td>
</tr>
<tr>
<td>log4j OFF</td>
<td>4</td>
</tr>
<tr>
<td>log4j INFO</td>
<td>154,000</td>
</tr>
<tr>
<td>j.u.logging OFF</td>
<td>1,800</td>
</tr>
<tr>
<td>Redirected System.out</td>
<td></td>
</tr>
</tbody>
</table>
Performance (YMMV)

Logging Cost (ns/operation, lower is better)

<table>
<thead>
<tr>
<th>Setting</th>
<th>ns/operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFR Disabled</td>
<td>0</td>
</tr>
<tr>
<td>JFR Enabled (stackdepth=1)</td>
<td>1,400</td>
</tr>
<tr>
<td>log4j OFF</td>
<td>4</td>
</tr>
<tr>
<td>log4j INFO</td>
<td>154,000</td>
</tr>
<tr>
<td>j.u.logging OFF</td>
<td>1,800</td>
</tr>
<tr>
<td>j.u.logging INFO</td>
<td>43,000,000</td>
</tr>
<tr>
<td>Redirected System.out</td>
<td>120,000,000</td>
</tr>
</tbody>
</table>
Performance Considerations

Default configuration designed to have less than 1% overhead
Other configurations can have more overhead

Stack depth (default: 64)
Deep call stacks can impact performance
Using JFR
Using JFR (JDK 11+)

# Start a recording
java -XX:StartFlightRecording ...

# Start a recording, and store it to file
java -XX:StartFlightRecording:filename=/tmp/foo.jfr ...

# Enable recording in an already running VM (pid 4711)
# jcmd <pid | main class name> JFR.start [options]
jcmd 4711 JFR.start OR jcmd MyApplication JFR.start

# Dump a recording from running VM (pid 4711), at most 50MB of data
jcmd 4711 JFR.dump maxsize=50MB
Demo: Looking at JFR recordings
Using bin/jfr

# Print summary of recording
jfr summary myrecording.jfr

# Print events
jfr print myrecording.jfr

# Print events in JSON format
jfr print --json myrecording.jfr

# Print GC related events
jfr print --categories "GC" myrecording.jfr
JFR: Use Cases

Production
   Troubleshooting / Root-cause analysis

Development
   Optimizing hot methods
   Allocation profiling

Testing
   Regression testing/monitoring execution profile changes
   Allocation, Lock Contention, …
Future Work
Future Work
Consuming Events **Today**

To access JFR data a recording must be
1. Started
2. Stopped
3. Dumped to a separate file

Reasonable for profiling
Not friendly to monitoring/continuous consumption
   Copying data out from disk repository creates overhead
   Recordings have same (redundant) information
Enter: JFR Event Streaming (JEP 349)

Goal: Make it trivial to consume and act on events continuously

API to read data directly from the disk repository
Even when recordings are in progress
Data flushed to repository continuously
Default: once a second
Simple Event Stream Consumer

```java
try (var rs = new RecordingStream()) {
    rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
    rs.onEvent("jdk.JavaMonitorEnter", event -> {
        System.out.println(event.getClass("monitorClass"));
    });

    rs.start();
}
```
Simple Event Stream Consumer
	ry (var rs = new RecordingStream()) {
    rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
    rs.onEvent("jdk.JavaMonitorEnter", event -> {
        System.out.println(event.getClass("monitorClass"));
    });

    rs.start();
}
Simple Event Stream Consumer

```java
try (var rs = new RecordingStream()) {
    rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
    rs.onEvent("jdk.JavaMonitorEnter", event -> {
        System.out.println(event.getClass("monitorClass"));
    });
    rs.start();
}
```
Simple Event Stream Consumer

```java
try (var rs = new RecordingStream()) {
    rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
    rs.onEvent("jdk.JavaMonitorEnter", event -> {
        System.out.println(event.getClass("monitorClass"));
    });
    rs.start();
}
```
Simple Event Stream Consumer

```java
try (var rs = new RecordingStream()) {
    rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
    rs.onEvent("jdk.JavaMonitorEnter", event -> {
        System.out.println(event.getClass("monitorClass"));
    });

    rs.start(); // “Blocking” call, will process events until stream ends/is closed
}
```
Simple Event Stream Consumer

```java
try (var rs = new RecordingStream()) {
    rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
    rs.onEvent("jdk.JavaMonitorEnter", event -> {
        System.out.println(event.getClass("monitorClass"));
    });
    rs.enable("jdk.CPULoad").withPeriod(Duration.ofSeconds(1));
    rs.onEvent("jdk.CPULoad", event -> {
        System.out.println(event.getFloat("machineTotal"));
    });

    rs.start();
}
```
Demo: Continuous Monitoring
Other

Access event stream over JMX
Additional JDK events
OpenJDK Project “Loom”: Fibers Support
Improve command line configuration
Event throttling – Record every n:th event
Deep tracing – Record *Everything* for a short period
JFR Integration Opportunities

Development
  IntelliJ, VisualVM, …

Monitoring
  APM, …

Frameworks
  Kafka, RxJava, Open Tracing, …
Life on the (not so) Bleeding Edge

Please **help us** by trying out the new features!

JDK 14 Early-Access builds: [http://jdk.java.net/14/](http://jdk.java.net/14/)

Feedback: [hotspot-jfr-dev@openjdk.java.net](mailto:hotspot-jfr-dev@openjdk.java.net)
Summary

JFR = JDK Flight Recorder
Available now, in a JDK near you!

An event based tracing framework
Built into the Java Runtime
Extremely low overhead, suitable for production environments
Allows correlation of data from different subsystems/software layers
With APIs for
  Producing application level events
  Consuming event streams
Thank You!

Questions?

@MikaelVidstedt

Director, Java Virtual Machine
Java Platform Group, Oracle