Java Platform, Standard Edition

Flight Recorder API Programmer's Guide

Release 26
G49898-01
March 2026

ORACLE

Java Platform, Standard Edition Flight Recorder API Programmer’s Guide, Release 26

G49898-01

Contents

Preface

This document shows you how to use the Flight Recorder API for more comprehensive application monitoring; you can analyze in greater detail events generated by applications, the JVM, and the operating system. Also, you can create your own events, record your own data, and view and parse the recordings. In addition, this document shows you how to use the Flight Recorder event streaming API, which enables you to consume Flight Recorder data continuously.

Audience

This document is intended for experienced Flight Recorder users who want to monitor their applications in greater detail.

Related Documents

- The jfr Command in the *Java Development Kit Tool Specifications*
- The `jdk.jfr` module

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I
Why Use the API?

Use the Flight Recorder API for more comprehensive application monitoring; you can analyze in greater detail events generated by applications, the JVM, and the operating system. In addition, you can create your own events, record your own data, and view and parse the recordings.

For example, you might create events and recordings for the following scenarios:

- To identify slow HTTP GET requests: A client application sends a request to a web server and it takes a long time for the request to be processed. To troubleshoot this problem, you can create an event that triggers if the request takes more than five seconds to process. You can also correlate these requests with JVM events, such as garbage collection or thread contention, which might impact web server performance.
- To track slow-running SQL queries: Some SQL queries take a long time to execute in a database server. To identify the problem, you can create an event to log all the SQL queries, and then analyze the recordings to track the slowest-running queries.

Flight Recorder API or Java Logging API?

The Java Logging API (see the package `java.util.logging`) captures information such as security failures, configuration errors, performance bottlenecks, and bugs in the application or platform. However, compared with the Java Logging API, the Flight Recorder API provides you with more information in the recording it generates (and in the events it records), more information regarding the context in which an event occurs, and more control over the timing of events.

Part II
Create Events

This section shows you how to create and record events. It contains the following topics:

1

Creating and Recording Your First Event

The sample `HelloWorldSample.java` creates an event named `com.oracle.Hello`.

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

public class HelloWorldSample {

    @Name("com.oracle.Hello")
    @Label("Hello World!")
    static class Hello extends Event {
        @Label("Message")
        String message;
    }

    public static void main(String... args) {
        Hello event = new Hello();
        event.begin();
        event.message = "Hello world!";
        event.commit();
    }
}
```

Run `HelloWorldSample` with the following command:

```
java -XX:StartFlightRecording:filename=hw.jfr HelloWorldSample.java
```

It runs `HelloWorldSample` and creates a recording file named `hw.jfr`.

To view the contents of the recording file, run this command:

```
jfr print hw.jfr
```

It prints all events recorded by Flight Recorder.

If you only want to view the `Hello` event that you created, then run this command:

```
jfr print --events Hello hw.jfr
```

It prints output similar to the following:

```
com.oracle.Hello {
  startTime = 16:44:14.841
  duration = 0.0170 ms
  message = "Hello world!"
  eventThread = "main" (javaThreadId = 1)
```

```
  stackTrace = [
    HelloWorldSample.main(String[]) line: 18
    jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Method, Object,
Object[])
    jdk.internal.reflect.NativeMethodAccessorImpl.invoke(Object, Object[])
line: 64
    jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(Object,
Object[]) line: 43
    java.lang.reflect.Method.invoke(Object, Object[]) line: 564
    ...
  ]
}
```

2
Event Metadata

The example `SetMetadataSample.java` defines an event named `com.oracle.Hello` and sets the annotations `@Name`, `@Description`, `@Label`, and `@Category`. (Note that this sample is in the package `frexamples`.)

```
package frexamples;

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

public class SetMetadataSample {

    @Name("com.oracle.Hello")
    @Label("Set Metadata Example")
    @Description("Demonstrates how to set the annotations "
        + "@Name, @Description, @Label, and @Category")
    @Category({ "Demonstration", "Tutorial" })
    static class Hello extends Event {
        @Label("Message")
        String message;
    }

    public static void main(String... args) {
        Hello event = new Hello();
        event.begin();
        event.message = "Hello Event!";
        event.commit();
    }
}
```

Ensure that the example is in a directory named `frexamples`, then run `SetMetadataSample` from this directory with the following commands:

```
java -XX:StartFlightRecording:filename=sm.jfr SetMetadataSample.java
jfr print --events Hello sm.jfr
```

The last command prints output similar to the following:

```
com.oracle.Hello {
  startTime = 23:43:48.444
  duration = 0.0177 ms
  message = "Hello Event!"
  ...
}
```

You can also use the `jfr print` command to filter events that belong to one or more categories:

```
jfr print --categories Demonstration sm.jfr
```

The `@Name` annotation overrides the default name for an event type. For example, the default name for the event created in this example would have been `frexamples.SetMetadataSample$Hello` if the `@Name` annotation hadn't been set. See Guidelines for Naming Events.

The `@Description` and `@Label` annotations enable to you add additional information about an event type. Note that you shouldn't use `@Label` as an identifier; use the `@Name` annotation instead. See Guidelines for Labeling Events

The `@Category` annotation enables you to associate one or more categories with an event type. To specify one category, use a string. To specify more than one category, use a comma-separated list of strings surrounded by braces (`{}`). See Categories.

3 Guidelines for Naming and Labeling Events

You should name and label all of your events by setting the annotations `@Name` and `@Label`.

Guidelines for Naming Events

Use the following format for naming your events, where `www.example.com` is the domain of your organization and *`Name`* is the name of your event class:

```
com.example.Name
```

When naming your event class, omit the word "Event."

By default, an event gets its name from its fully qualified class name. For example, in the example `SetMetadataSample.java` (see Event Metadata), the default name of the event `Hello` is `frexamples.SetMetadataSample$Hello`.

This works well for experimentation, but avoid omitting the `@Name` annotation for production code. You might have to refactor your source code and move the event class to a different package. If you haven't specified the event's name with the `@Name` annotation, then refactoring an event class can break code or settings files that configure the event. It can also break code that parses recording files that use the default name to identify an event.

The fully qualified class name may also contain redundant or unnecessary strings such as `jfr`, `internal`, `events`, or `Events` that you should omit.

An event name should be short but not so short that it clashes with other organizations or products. The name should be easy to understand for users who want to configure the event. This is especially true if the event is part of a framework or library that is meant to be used by others. It's usually sufficient to put all the events for a library or product in the same namespace. For example, all the events for OpenJDK are in the `jdk` namespace. There are no sub-namespaces for `hotspot`, `gc`, or `compiler` as this would just complicate things. However, it's possible to divide events into categories with the `@Category` annotation, which you can freely change without disruption.

Guidelines for Labeling Events

For labels, use headline-style capitalization: Capitalize the first and last words and all nouns, pronouns, adjectives, verbs and adverbs. Do not include ending punctuation. As with event names, omit the word "Event." Note that you shouldn't use `@Label` as an identifier; use the `@Name` annotation instead.

Use labels to display events in user interfaces such as a custom visualization tool. For example, JDK Mission Control's Event Browser uses the label to display events in its Event Types Tree.

4
Categories

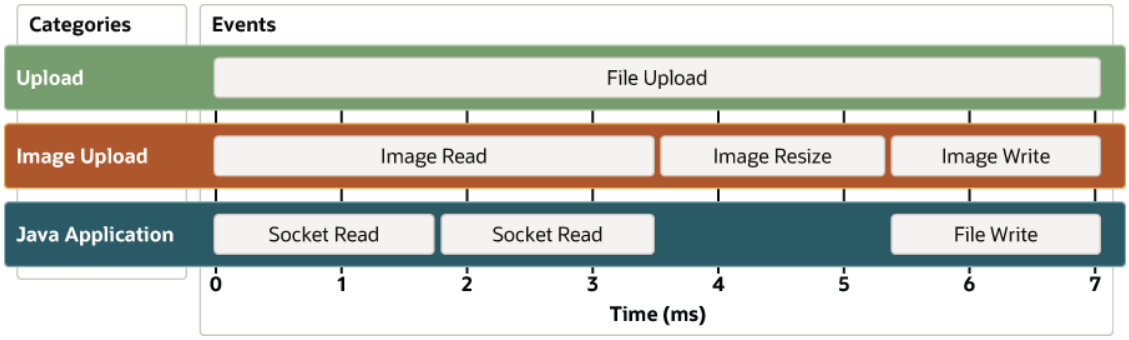
With the `@Category` annotation, you can assign any number of categories to an event. Categories enable you to identify similar events that should be displayed together, for example, in a graph or a tree. Although you can assign any category to an event (a category is just a string), it's best to first determine your categories.

If there's a chance that two or more events can happen at the same time and in the same thread, even if their start and end times might be different, then they should belong to different categories to prevent overlap when they're represented in a graphical user interface.

For example, suppose that you want to monitor image uploads to a web server. You create an event named File Upload that begins when a user uploads a file and ends when an upload completes. For advanced diagnostics about image uploads, you create more detailed events named Image Read, Image Resize, and Image Write. During these detailed events, other low-level events occur, for example, Socket Read during Image Read and File Write during Image Write. In this example, the event File Upload would overlap the events Image Read, Image Resize, and Image Write, which means that the File Upload event might hide the detailed events in some event visualizers. The same issue might happen for Image Read and Socket Read, and Image Write and File Write.

To prevent event overlap, make sure that events that might overlap belong to different categories. The following diagram illustrates one categorization scheme that prevents event overlaps and how an event visualizer could display them:

Figure 4-1 Categorizing Concurrent Events to Prevent Overlaps



File Upload belongs to the category Upload. Image Read, Image Resize, and Image Write belong to the category Image Upload. Socket Read and File Write belong to the category Java Application.

The example `CategoriesSample.java` implements this categorization scheme and simulates the creation of events as illustrated in the figure:

```
import jdk.jfr.Category;
import jdk.jfr.DataAmount;
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;
```

```
import jdk.jfr.Percentage;

public class CategoriesSample {

    public static final String PROGRAMMERS_GUIDE_SAMPLES =
        "Programmer's Guide Samples";
    public static final String UPLOAD = "Upload";
    public static final String IMAGE_UPLOAD = "Image Upload";
    public static final String JAVA_APPLICATION = "Java Application";

    @Name("com.oracle.FileUpload")
    @Label("File Upload")
    @Category({PROGRAMMERS_GUIDE_SAMPLES, UPLOAD})
    private static class FileUpload extends Event { }

    @Name("com.oracle.ImageRead")
    @Label("Image Read")
    @Category({PROGRAMMERS_GUIDE_SAMPLES, IMAGE_UPLOAD})
    private static class ImageRead extends Event {
        @DataAmount(DataAmount.BYTES)
        long bytesUploaded;
    }

    @Name("com.oracle.ImageResize")
    @Label("Image Resize")
    @Category({PROGRAMMERS_GUIDE_SAMPLES, IMAGE_UPLOAD})
    private static class ImageResize extends Event {
        @Percentage
        double scale;
    }

    @Name("com.oracle.ImageWrite")
    @Label("Image Write")
    @Category({PROGRAMMERS_GUIDE_SAMPLES, IMAGE_UPLOAD})
    private static class ImageWrite extends Event {
        @DataAmount(DataAmount.BYTES)
        long bytesWritten;
    }

    @Name("com.oracle.SocketRead")
    @Label("Socket Read")
    @Category({PROGRAMMERS_GUIDE_SAMPLES, JAVA_APPLICATION})
    private static class SocketRead extends Event {
        @DataAmount(DataAmount.BYTES)
        long bytesRead;
    }

    @Name("com.oracle.FileWrite")
    @Label("File Write")
    @Category({PROGRAMMERS_GUIDE_SAMPLES, JAVA_APPLICATION})
    private static class FileWrite extends Event {
        @DataAmount(DataAmount.BYTES)
        long bytesWritten;
    }

    public static void main(String... args) {
```

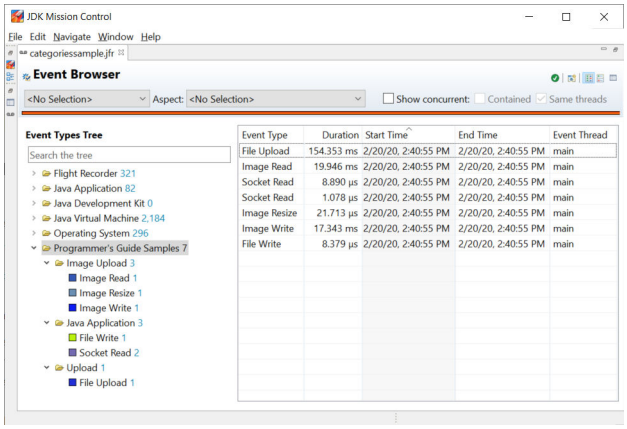
```
        FileUpload fu = new FileUpload();
        fu.begin();

        ImageRead ir = new ImageRead();
        ir.begin();
        ir.bytesUploaded = 2048;

        SocketRead sr1 = new SocketRead();
        sr1.begin();
        sr1.bytesRead = 1024;
        sr1.commit();

        SocketRead sr2 = new SocketRead();
        sr2.begin();
        sr2.bytesRead = 1024;
        sr2.commit();

        ir.commit();

        ImageResize irs = new ImageResize();
        irs.begin();
        irs.scale = 0.5;
        irs.commit();

        ImageWrite iw = new ImageWrite();
        iw.begin();
        iw.bytesWritten = 1024;

        FileWrite fw = new FileWrite();
        fw.begin();
        fw.bytesWritten = 1024;
        fw.commit();

        iw.commit();
        fu.commit();
    }
}
```

Run `CategoriesSample` with the following command:

```
java -XX:StartFlightRecording:filename=categoriessample.jfr
CategoriesSample.java
```

Then, open `categoriessample.jfr` in JDK Mission Control. From the **Event Types Tree** in the **Event Browser**, find the events created by this sample in the category **Programmer's Guide Samples**:

Figure 4-2 JDK Mission Control Browser Displaying Events from CategoriesSample



JDK Mission Control categorizes events based on their `@Category` attribute and lists them by their `@Label` attribute.

5
Measuring Time

The example `MeasureTimeSample.java` shows you how to measure the time of an operation by calling the `Event.begin` and `Event.commit` methods.

```
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;
public class MeasureTimeSample {

    @Name("com.oracle.MeasureDuration")
    @Label("Measure Duration")
    static class MeasureMyDuration extends Event { }

    public static void main(String... args) throws Exception {
        MeasureMyDuration event = new MeasureMyDuration();
        event.begin();
        Thread.sleep(42);
        event.commit();
    }
}
```

Note that the `commit` method ends the timing of an event without the need of an explicit call to the `end` method.

Run `MeasureTimeSample` with the following commands:

```
java -XX:StartFlightRecording:filename=mt.jfr MeasureTimeSample.java
jfr print --events MeasureDuration mt.jfr
```

The last command prints output similar to the following:

```
com.oracle.MeasureDuration {
  startTime = 12:26:43.169
  duration = 45.3 ms
  ...
}
```

6
Data Types

The example `PersistFieldTypesSample.java` shows which field types you can persist in an event, which are the following:

- `java.lang.String`, which may be null
- `java.lang.Thread`, which may be null
- `java.lang.Class`, which may be null
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `char`
- `boolean`

> **Note**
>
> Events don't support arrays.

The following is the `PersistFieldTypesSample.java` example:

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

public class PersistFieldTypesSample {

    @Name("com.oracle.FieldTypes")
    @Label("Allowed Field Types")
    static class FieldTypes extends Event {
        @Label("Class Value")
        Class<?> classValue;

        @Label("Thread Value")
        Thread threadValue; // thread must be started

        @Label("String Value")
        String stringValue;

        @Label("Byte Value")
        byte byteValue;
```

```
        @Label("Short Value")
        short shortValue;

        @Label("Int Value")
        int intValue;

        @Label("Long Value")
        long longValue;

        @Label("Float Value")
        float floatValue;

        @Label("Double Value")
        double doubleValue;

        @Label("Character Value")
        char characterValue;

        @Label("Boolean Value")
        boolean booleanValue;
    }

    public static void main(String... args) {
        FieldTypes event = new FieldTypes();
        event.classValue = Math.class;
        event.threadValue = Thread.currentThread();
        event.stringValue = "Hello";
        event.byteValue = 42;
        event.shortValue = 4711;
        event.intValue = Integer.MAX_VALUE;
        event.longValue = Long.MAX_VALUE;
        event.doubleValue = Math.PI;
        event.floatValue = Float.NaN;
        event.characterValue = '!';
        event.booleanValue = true;
        event.commit();
    }
}
```

Run `PersistFieldTypesSample` with the following commands:

```
java -XX:StartFlightRecording:filename=pft.jfr PersistFieldTypesSample.java
jfr print --events FieldTypes pft.jfr
```

The last command prints output similar to the following:

```
com.oracle.FieldTypes {
  startTime = 12:33:12.434
  classValue = java.lang.Math (classLoader = bootstrap)
  threadValue = "main" (javaThreadId = 1)
  stringValue = "Hello"
  byteValue = 42
  shortValue = 4711
  intValue = 2147483647
  longValue = 9223372036854775807
```

```
  floatValue = N/A
  doubleValue = 3.141592653589793
  characterValue = !
  booleanValue = true
  ...
}
```

7

Dynamic Events

Dynamic events enable you to define events at run time, including their annotations and fields.

> **Note**
>
> Only use dynamic events if you won't know the layout of an event until you run your application.

The example `DynamicSample.java` creates a dynamic event named `com.oracle.RandomString`, which includes a field whose name is a random string:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import jdk.jfr.AnnotationElement;
import jdk.jfr.Category;
import jdk.jfr.Description;
import jdk.jfr.Event;
import jdk.jfr.EventFactory;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.ValueDescriptor;

public class DynamicSample {

    private static String randomString(int n) {

        var ALPHA_NUMERIC_STRING = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        var builder = new StringBuilder();
        while (n-- != 0) {
            int character = (int) (Math.random()
                * ALPHA_NUMERIC_STRING.length());
            builder.append(ALPHA_NUMERIC_STRING.charAt(character));
        }
        return builder.toString();
    }

    public static void main(String[] args) {

        String[] category = { "Demonstration", "Tutorial" };
        var eventAnnotations = new ArrayList<AnnotationElement>();
        eventAnnotations
            .add(new AnnotationElement(
                Name.class, "com.oracle.RandomString"));
        eventAnnotations.add(new AnnotationElement(Label.class,
            "Field Named with Random String"));
        eventAnnotations.add(new AnnotationElement(Description.class,
```

```
            "Demonstrates how to create a dynamic event"));
        eventAnnotations.add(new AnnotationElement(
            Category.class, category));

        var fields = new ArrayList<ValueDescriptor>();
        var messageAnnotations = Collections
            .singletonList(new AnnotationElement(Label.class, "Message"));
        var randomFieldName = DynamicSample.randomString(8);
        fields.add(new ValueDescriptor(String.class, randomFieldName,
            messageAnnotations));
        var numberAnnotations = Collections
            .singletonList(new AnnotationElement(Label.class, "Number"));
        fields.add(new ValueDescriptor(
            int.class, "number", numberAnnotations));

        var f = EventFactory.create(eventAnnotations, fields);

        Event event = f.newEvent();
        event.set(0, "hello, world!");
        event.set(1, 100);
        event.commit();
    }
}
```

Run `DynamicSample` with the following commands:

```
java -XX:StartFlightRecording:filename=d.jfr DynamicSample.java
jfr print --events RandomString d.jfr
```

The last command prints output similar to the following:

```
com.oracle.RandomString {
  startTime = 12:56:32.782
  ZZEIUMTG = "hello, world!"
  number = 100
  ...
}
```

To create a dynamic event, call the static method `EventFactory.create<List<AnnotationElement>, List<ValueDescriptor>)`:

```
var f = EventFactory.create(eventAnnotations, fields);
```

The first argument is a list of your event's annotations, which may include built-in annotations such as `@Name` and `@Description`.

The second argument is a list of your event's fields. Define them with the `ValueDescriptor` class.

8
Custom Annotations

Creating custom annotations for events is the same as creating Java annotations. The example `CustomAnnotationSample.java` demonstrates this.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import jdk.jfr.Description;
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.MetadataDefinition;
import jdk.jfr.Name;
import jdk.jfr.Relational;

public class CustomAnnotationSample {

    @MetadataDefinition
    @Name("com.oracle.Severity")
    @Label("Severity")
    @Description("Value between 0 and 100 that indicates " +
        "severity. 100 is most severe.")
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ ElementType.TYPE })
    public @interface Severity {
       int value() default 50;
    }

    @MetadataDefinition
    @Name("com.oracle.TransactionId")
    @Label("Transaction ID")
    @Relational
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ ElementType.FIELD })
    public @interface TransactionId { }

    @Name("com.oracle.TransactionBlocked")
    @Severity(80)
    @Label("Transaction Blocked")
    static class TransactionBlocked extends Event {
        @TransactionId
        @Label("Transaction")
        long transaction;

        @TransactionId
        @Label("Transaction Blocker")
        long transactionBlocker;
    }
```

```
    public static void main(String... args) {
        TransactionBlocked event = new TransactionBlocked();
        event.begin();
        event.transaction = 1;
        event.transactionBlocker = 2;
        event.commit();
    }
}
```

Run `CustomAnnotationSample` with the following command:

```
java -XX:StartFlightRecording:filename=ca.jfr CustomAnnotationSample.java
```

To view annotations, categories, field layouts, and other information about all the events in `customannotationsample.jfr`, run the following command:

```
jfr metadata ca.jfr
```

The output of the previous command includes the following:

```
@Name("com.oracle.Severity")
@Label("Severity")
@Description("Value between 0 and 100 that indicates severity. 100 is most
severe.")
class Severity extends java.lang.annotation.Annotation {
  int value;
}

@Name("com.oracle.TransactionId")
@Label("Transaction ID")
@Relational
class TransactionId extends java.lang.annotation.Annotation {
}
...
@Name("com.oracle.TransactionBlocked")
@Severity(80)
@Label("Transaction Blocked")
class TransactionBlocked extends jdk.jfr.Event {
  @Label("Start Time")
  @Timestamp("TICKS")
  long startTime;

  @Label("Duration")
  @Timespan("TICKS")
  long duration;

  @Label("Event Thread")
  @Description("Thread in which event was committed in")
  Thread eventThread;

  @Label("Stack Trace")
  @Description("Stack Trace starting from the method the event was committed
in")
```

```
  StackTrace stackTrace;

  @TransactionId
  @Label("Transaction")
  long transaction;

  @TransactionId
  @Label("Transaction Blocker")
  long transactionBlocker;
}
```

To access values of custom annotations, use the `EventType.getAnnotation` method, which takes one argument, the `Class` object that corresponds to the annotation type. For example, the following code prints the events whose severity is greater than 50:

```
for (var e : RecordingFile.readAllEvents(file)) {
    EventType t = e.getEventType();
    Severity s = t.getAnnotation(Severity.class);
    if (s != null && s.getValue() > 50) {
        System.out.println(e);
    }
}
```

See Declaring an Annotation Type in *The Java Tutorials*.

9

Inheritance of Annotations, Settings, and Fields

When a class extends an event, it inherits the event's annotations, settings, and fields. However, a class doesn't inherit private fields or annotations that lack the `@java.lang.Inherited` meta-annotation.

The example `InheritanceSample.java` demonstrates this. It defines three events: `FileAction`, `FileUpload`, and `ImageUpload`.

```
import jdk.jfr.Category;
import jdk.jfr.Description;
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.StackTrace;

public class InheritanceSample {

    @Category("Files")
    @StackTrace(false)
    abstract static class FileAction extends Event {
        @Label("In Progress")
        boolean inProgress;
    }

    @Name("com.oracle.FileUpload")
    @Description("Uploaded file that might be a text file")
    @Label("File Upload")
    static class FileUpload extends FileAction {
        @Label("Text file")
        private boolean isText;
    }

    @Name("com.oracle.ImageUpload")
    @Label("Image Upload")
    static class ImageUpload extends FileUpload {
    }

    public static void main(String... args) {
        FileUpload fu = new FileUpload();
        fu.inProgress = true;
        fu.isText = false;
        fu.commit();

        ImageUpload iu = new ImageUpload();
        iu.inProgress = false;
        iu.commit();
```

```
    }
}
```

Run `InheritanceSample` with the following commands:

```
java -XX:StartFlightRecording:filename=i.jfr InheritanceSample.java
jfr print --events FileUpload,ImageUpload i.jfr
```

The last command prints output similar to the following:

```
com.oracle.FileUpload {
  startTime = 15:22:28.794
  isText = false
  inProgress = true
  ...
}

com.oracle.ImageUpload {
  startTime = 15:22:28.822
  inProgress = false
  ...
}
```

Abstract event classes, such as `FileAction` are not registered, so their metadata is never available for inspection.

Classes don't inherit annotations that lack the `@java.lang.Inherited` annotation, such as `@Name` and `@Description`.

Because the field `isText` is private, `ImageUpload` doesn't inherit it.

Part III
Configure Events and Flight Recorder

This section describes how to configure events and Flight Recorder to optimize their performance and control their behavior.

Each event has the following predefined settings:

- `@Enabled`: Specifies whether the event is recorded. The default value is `true`. See Enable and Disable Events.
- `@Threshold`: Specifies the duration below which an event is not recorded. The default is `0` (no limit). See Event Threshold.
- `@Period`: Specifies the interval at which the event is emitted, if it is periodic. The default value is `everyChunk`, which means that the periodic event will be emitted at least once in the recording. See Periodic Events
- `@StackTrace`: Specifies whether the stack trace from the `Event::commit()` method is recorded. The default value is `true`. See Printing Event Stack Trace.

Flight Recorder provides various options to filter events; see Filter Events with SettingDefinition.

It's recommended that you specify a preconfigured configuration, which contains a collection of settings that control how much information Flight Recorder generates; see Flight Recorder Configurations. If you don't specify a preconfigured configuration, then Flight Recorder records information about all events; it monitors the running system at an extremely high level of detail and produces enormous amounts of data.

10

Enable and Disable Events

You can enable and disable events with the `@Enabled` annotation. The example `EnablementSample.java` demonstrates this.

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

public class EnablementSample {

    @Name("com.oracle.WontSeeMe")
    @Label("Won't See Me")
    @Enabled(false)
    static class WontSeeMe extends Event {
    }

    @Name("com.oracle.WillSeeMe")
    @Label("Will See Me")
    @Enabled(true)
    static class WillSeeMe extends Event {
    }

    public static void main(String... args) throws Exception {
        WontSeeMe event1 = new WontSeeMe();
        event1.commit();

        WillSeeMe event2 = new WillSeeMe();
        event2.commit();
    }
}
```

11

Event Threshold

Setting a threshold on an event means that Flight Recorder won't record it if its duration is less than the threshold. This enables you to limit the number of events that Flight Recorder records. By default, events have a threshold of 0 ms. It's recommended to set a threshold if an operation occurs frequently and outliers are of greatest concern.

The `SetThresholdSample.java` example creates ten events with a random duration. Flight Recorder records only those events whose duration is greater than 50 ms.

```
import java.util.Random;
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.Threshold;

public class SetThresholdSample {

    @Name("com.oracle.RandomSleep")
    @Label("Random Sleep")
    @Threshold("50 ms")
    static class RandomSleep extends Event {
        @Label("Event number")
        int eventNumber;
        @Label("Random Value")
        int randomValue;
    }

    public static void main(String... args) throws Exception {
        Random randNum = new Random();
        for (int i = 0; i < 10; i++) {
            RandomSleep event = new RandomSleep();
            event.begin();
            event.eventNumber = i;
            event.randomValue = Math.abs(randNum.nextInt() % 100);
            System.out.println("Event #" + i + ": " + event.randomValue);
            Thread.sleep(event.randomValue);
            event.commit();
        }
    }
}
```

Note that the `commit` method ends the timing of an event without the need of an explicit call to the `end` method.

Run `SetThresholdSample` with the following commands:

```
java -XX:StartFlightRecording:filename=st.jfr SetThresholdSample.java
jfr print --events RandomSleep st.jfr
```

The first command prints output similar to the following:

```
Event #0: 97
Event #1: 15
Event #2: 25
Event #3: 73
Event #4: 38
Event #5: 11
Event #6: 5
Event #7: 28
Event #8: 42
Event #9: 37
```

The last command prints output similar to the following:

```
com.oracle.RandomSleep {
  startTime = 23:17:42.050
  duration = 103.813 ms
  eventNumber = 0
  randomValue = 97
  ...
}

com.oracle.RandomSleep {
  startTime = 23:17:42.197
  duration = 77.726 ms
  eventNumber = 3
  randomValue = 73
  ...
}
```

The shouldCommit Method

You can reduce the overhead of expensive operations with the `Event.shouldCommit` method, which only commits an event if its duration is within a specified threshold.

The example `ShouldCommit.java` creates ten events with a random duration. Flight Recorder commits only those events whose duration is greater than 20 ms.

```
import java.util.Random;

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

public class ShouldCommitSample {

    @Name("com.oracle.RandomSleep")
    @Label("Random Sleep")
    @Threshold("20 ms")
    static class RandomSleep extends Event {
        @Label("ID")
        int id;
```

```
        @Label("Value Kind")
        String valueKind;
    }

    public static void main(String... args) throws Exception {
        Random randNum = new Random();
        for (int i = 0; i < 10; i++) {
            RandomSleep event = new RandomSleep();
            event.begin();
            event.id = i;
            int value = randNum.nextInt(40);
            System.out.println("ID " + i + ": " + value);
            Thread.sleep(value);
            event.end();
            if (event.shouldCommit()) {
                // Format message outside timing of event
                if (value < 10) {
                    event.valueKind = "It was a low value of " +
                        value + "!";
                } else if (value < 20) {
                    event.valueKind = "It was a normal value of " +
                        value + "!";
                } else {
                    event.valueKind = "It was a high value of " +
                        value + "!";
                }
                event.commit();
            }
        }
    }
}
```

Run this example with the following commands:

```
java -XX:StartFlightRecording:filename=shouldcommit.jfr ShouldCommit.java
jfr print --events RandomSleep shouldcommit.jfr
```

The first command prints output similar to the following:

```
ID 0: 8
ID 1: 2
ID 2: 34
ID 3: 0
ID 4: 11
ID 5: 2
ID 6: 14
ID 7: 28
ID 8: 27
ID 9: 11
```

The last command prints output similar to the following:

```
com.oracle.RandomSleep {
  startTime = 23:27:10.642
```

```
  duration = 36.711 ms
  id = 2
  valueKind = "It was a high value of 34!"
  ...
}

com.oracle.RandomSleep {
  startTime = 23:27:10.711
  duration = 29.390 ms
  id = 7
  valueKind = "It was a high value of 28!"
  ...
}

com.oracle.RandomSleep {
  startTime = 23:27:10.741
  duration = 28.475 ms
  id = 8
  valueKind = "It was a high value of 27!"
  ...
}
```

12
Periodic Events

The example `PeriodicSample.java` creates a periodic event named `StartedThreadCount` that records the total number of threads that have been created and started every second.

```
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadMXBean;

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

public class PeriodicSample {

    private static ThreadMXBean tBean =
        ManagementFactory.getThreadMXBean();

    @Name("com.oracle.StartedThreadCount")
    @Label("Total number of started threads")
    @Period("1 s")
    static class StartedThreadCount extends Event {
        long totalStartedThreadCount;
    }

    public static void main(String[] args) throws InterruptedException {

        Runnable hook = () -> {
            StartedThreadCount event = new StartedThreadCount();
            event.totalStartedThreadCount =
                tBean.getTotalStartedThreadCount();
            event.commit();
        };

        FlightRecorder.addPeriodicEvent(StartedThreadCount.class, hook);

        for (int i = 0; i < 4; i++) {
            Thread.sleep(1500);
            Thread t = new Thread();
            t.start();
        }

        FlightRecorder.removePeriodicEvent(hook);
    }
}
```

Run `PeriodicSample` with the following commands:

```
java -XX:StartFlightRecording:filename=periodic.jfr PeriodicSample.java
jfr print --events StartedThreadCount periodic.jfr
```

The last command prints output similar to the following:

```
com.oracle.StartedThreadCount {
  startTime = 00:59:40.769
  totalStartedThreadCount = 12
    ...
}

com.oracle.StartedThreadCount {
  startTime = 00:59:41.816
  totalStartedThreadCount = 12
    ...
}

com.oracle.StartedThreadCount {
  startTime = 00:59:42.866
  totalStartedThreadCount = 13
    ...
}

com.oracle.StartedThreadCount {
  startTime = 00:59:43.918
  totalStartedThreadCount = 14
    ...
}

com.oracle.StartedThreadCount {
  startTime = 00:59:44.962
  totalStartedThreadCount = 14
    ...
}
```

To create a periodic event, follow these two steps:

1. Specify how often the event should be emitted with the `@Period` annotation:

   ```
   @Name("com.oracle.StartedThreadCount")
   @Label("Total number of started threads")
   @Period("1 s")
   static class StartedThreadCount extends Event {
       long totalStartedThreadCount;
   }
   ```

   Valid units for a period are: `ns`, `us`, `ms`, `s`, `m`, `h`, and `d`.

   You can also specify one of the following:

   - `everyChunk`: A periodic event will be emitted at least once in the recording.
   - `beginChunk`: A periodic event will be emitted in the beginning of a recording.

- `endChunk`: A periodic event will be emitted in the end of a recording.

2. Add the periodic event with the `FlightRecorder.addPeriodicEvent(Class<? extends Event>, Runnable)` static method. The first argument is the name of the periodic event's class. The second argument is a callback method that's represented by a lambda expression that creates and commits the event:

```
Runnable hook = () -> {
    StartedThreadCount event = new StartedThreadCount();
    event.totalStartedThreadCount =
        tBean.getTotalStartedThreadCount();
    event.commit();
};

FlightRecorder.addPeriodicEvent(StartedThreadCount.class, hook);
```

The method `FlightRecorder.removePeriodicEvent(Runnable)` removes the lambda expression associated with a periodic event. In most cases, you won't need this method; if you want to disable a periodic event, you can call `Recording.disable(Class<? extends Event>)`. However, one reason to call `removePeriodicEvent` is to avoid memory leaks. For example, suppose you have an application server where data is loaded and unloaded. If the callback method references data that the server loads and unloads, then it may prevent that data from being garbage collected. You can avoid this by removing the callback method when the data is unloaded.

13
Printing Event Stack Trace

The example `StackTraceSample.java` prints information about an event's stack trace.

`StackTraceSample` uses the Event Streaming API (see Monitor Events with Flight Recorder Event Streaming API) to print stack trace information of `WithStackTrace` events. The sample recursively calls the method `firstFunc` six times. This method creates an event named `WithStackTrace`. Every time an `WithStackTrace` occurs, information about the event's stack trace is printed.

```
import java.util.List;
import java.util.function.Consumer;

import jdk.jfr.Event;
import jdk.jfr.EventType;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.StackTrace;
import jdk.jfr.consumer.RecordedEvent;
import jdk.jfr.consumer.RecordedFrame;
import jdk.jfr.consumer.RecordedStackTrace;
import jdk.jfr.consumer.RecordingStream;

public class StackTraceSample {

    @Name("com.oracle.WithStackTrace")
    @Label("With Stack Trace")
    @StackTrace(true)
    static class WithStackTrace extends Event {
        String message;
    }

    public static void main(String... args) throws Exception {
        Consumer<RecordedEvent> myCon = x -> {
            EventType et = x.getEventType();
            System.out.println("Label: " + et.getLabel());
            System.out.println("Message: " + x.getValue("message"));
            RecordedStackTrace rst = x.getStackTrace();
            if (rst != null) {
                List<RecordedFrame> frames = rst.getFrames();
                System.out.println(
                    "Number of frames: " + frames.size());
                for (RecordedFrame rf : frames) {
                    System.out.println("Method, line number: "
                        + rf.getMethod().getName() + ", "
                        + rf.getLineNumber());
                }
            }
            System.out.println("");
        };
```

```
        try (RecordingStream rs = new RecordingStream()) {
            rs.onEvent("com.oracle.WithStackTrace", myCon);
            rs.startAsync();
            firstFunc(5);
            rs.awaitTermination();
        }
    }

    static void firstFunc(int n) {
        if (n > 0) {
            secondFunc(n - 1);
        }
        WithStackTrace event = new WithStackTrace();
        event.message = "n = " + n;
        event.commit();

    }

    static void secondFunc(int n) {
        firstFunc(n);
    }
}
```

The example `StackTraceSample` prints output similar to the following:

```
Label: With Stack Trace
Message: n = 0
Number of frames: 12
Method, line number: firstFunc, 97
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: main, 86

Label: With Stack Trace
Message: n = 1
Number of frames: 10
Method, line number: firstFunc, 97
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: main, 86
```

```
Label: With Stack Trace
Message: n = 2
Number of frames: 8
Method, line number: firstFunc, 97
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: main, 86

Label: With Stack Trace
Message: n = 3
Number of frames: 6
Method, line number: firstFunc, 97
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: main, 86

Label: With Stack Trace
Message: n = 4
Number of frames: 4
Method, line number: firstFunc, 97
Method, line number: secondFunc, 102
Method, line number: firstFunc, 93
Method, line number: main, 86

Label: With Stack Trace
Message: n = 5
Number of frames: 2
Method, line number: firstFunc, 97
Method, line number: main, 86
```

An event's stack trace, an instance of `RecordedStackTrace`, consists of a list of `RecordedFrame` instances. You can obtain the following information from a `RecordedFrame` with these methods:

- `getMethod()`: Returns the method from which the event was run.
- `getLineNumber()`: Returns the line number from which the event was run.
- `isJavaFrame()`: Indicates whether the `RecordedFrame` is a Java frame.
- `getBytecodeIndex()`: Returns the bytecode index from which the event was run.
- `getType()`: Returs the frame type; possible values include `Interpreted`, `JIT compiled`, and `Inlined`.

Flight Recorder uses a default stack depth of 64 method calls, which is more than enough for this example. You can change this with the `stackdepth` command-line option:

```
-XX:FlightRecorderOptions:stackdepth=depth
```

Note that values greater than 64 could create significant overhead and reduce performance.

14

Filter Events with SettingDefinition

The example `FilteringSample.java` (along with `RegExpControl.java`) uses a `SettingDefinition` to filter which events Flight Recorder records. In this example, it records `Hello` events that have a value that starts with `g` in its `message` field.

```
import java.io.IOException;

import jdk.jfr.Description;
import jdk.jfr.Event;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.Recording;
import jdk.jfr.SettingDefinition;

public class FilteringSample {

    @Name("com.oracle.FilteredHello")
    @Label("Hello With Message Filter")
    static class FilteredHello extends Event {
        @Label("Message")
        String message;

        @Label("Message Filter")
        @Description("Filters messages with regular expressions")
        @SettingDefinition
        protected boolean messageFilter(RegExpControl control) {
            return control.matches(message);
        }
    }

    public static void main(String[] args) throws IOException {

        try (Recording r = new Recording()) {
            r.enable(FilteredHello.class).with("messageFilter", "g.*");
            r.start();
            FilteredHello greenEvent = new FilteredHello();
            FilteredHello yellowEvent = new FilteredHello();
            FilteredHello redEvent = new FilteredHello();
            greenEvent.message = "green";
            yellowEvent.message = "yellow";
            redEvent.message = "red";
            greenEvent.commit();
            yellowEvent.commit();
            redEvent.commit();
        }
    }
}
```

The example `FilteringSample` requires `RegExpControl.java`:

```
import java.util.Set;
import java.util.regex.Pattern;

import jdk.jfr.SettingControl;

public class RegExpControl extends SettingControl {

    private Pattern pattern = Pattern.compile(".*");

    @Override
    public void setValue(String value) {
        this.pattern = Pattern.compile(value);
    }

    @Override
    public String combine(Set<String> values) {
        return String.join("|", values);
    }

    @Override
    public String getValue() {
        return pattern.toString();
    }

    public boolean matches(String s) {
        return pattern.matcher(s).find();
    }
}
```

Compile `FilteringSample.java` and `RegExpControl.java`, then run `FilteringSample` with the following commands:

```
java -XX:StartFlightRecording:filename=filteringsample.jfr FilteringSample
jfr print --events FilteredHello filteringsample.jfr
```

The last command prints output similar to the following:

```
com.oracle.FilteredHello {
  startTime = 23:38:28.364
  message = "green"
  ...
}
```

The annotation `@SettingDefinition` specifies which method Flight Recorder calls to determine whether it records a particular event. In this example, it calls `messageFilter(RegExpControl)`:

```
@SettingDefinition
protected boolean messageFilter(RegExpControl control) {
    return control.matches(message);
}
```

This method's parameter, `RegExpControl`, extends the class `SettingControl`. In this example, `RegExpControl.java` implements a regular expression setting control; the method `matches(String)` returns `true` when its string matches the field `pattern` (which an application can change with the `setValue(String)` method).

The methods `setValue()`, `getValue()` and `combine(Set<String>)` methods are invoked when a setting value changes, which typically happens when a recording is started or stopped. The `combine(Set<String>)` method is invoked to resolve what value to use when multiple recordings are running at the same time.

15

Exclude Fields from Being Persisted with the transient Keyword

You can exclude fields from being persisted with the `transient` keyword. The example `ExcludeFieldsSample.java` demonstrates this.

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

public class ExcludeFieldsSample {

    @Name("com.oracle.Message")
    @Label("Message")
    static class Message extends Event {
        String messageA;
        transient String messageB;
        String messageC;
    }

    public static void main(String... args) {
        Message event = new Message();
        event.messageA = "hello";
        event.messageB = "world"; // will not be persisted.
        event.messageC = "!";
        event.commit();
    }
}
```

Run `ExcludeFieldsSample` with the following commands:

```
java -XX:StartFlightRecording:filename=excludefieldssample.jfr
ExcludeFieldsSample.java
jfr print --events Message excludefieldssample.jfr
```

The last command prints output similar to the following:

```
com.oracle.Message {
  startTime = 23:41:15.425
  messageA = "hello"
  messageC = "!"
  ...
}
```

16

Manually Register and Unregister an Event

By default, an event is automatically registered when the event class is initialized. Alternatively, you can manually register an event with the `@Registered` annotation. One reason to do this is to take control of the security context in which the event is initialized.

The difference between the `@Enabled` annotation and the `@Registered` annotation is that when an event is unregistered, its metadata, such as the field layout, is not available for inspection. A call to `FlightRecorder::register` can ensure that an event class is visible for configuration, for example, to a Java Management Extensions (JMX) client.

The example `RegistrationSample.java` demonstrates this:

```
import jdk.jfr.Event;
import jdk.jfr.FlightRecorder;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.Registered;

public class RegistrationSample {

    @Name("com.oracle.Message")
    @Label("Message")
    @Registered(false)
    static class Message extends Event {
        String message;
    }

    public static void main(String... args) {

        Message event1 = new Message();
        event1.message = "Not registered, so you won't see this";
        event1.commit();

        FlightRecorder.register(Message.class);
        Message event2 = new Message();
        event2.message = "Now registered, so you will see this!";
        event2.commit();

        FlightRecorder.unregister(Message.class);

        Message event3 = new Message();
        event3.message = "Not registered again, so you won't see this";
        event3.commit();
    }
}
```

17

Flight Recorder Configurations

Flight Recorder configurations control the amount of data that is recorded.

Flight Recorder uses two preconfigured configurations, `default.jfc` and `profile.jfc`. These configurations have predefined settings for each event type. Both of these configurations are located in `<java_home>/lib/jfr`. By default, Flight Recorder uses the `default.jfc` configuration when you start a recording. The `default.jfc` configuration is recommended for continuous recordings. It gives a good balance between data and performance (typically, less than 1% overhead). The `profile.jfc` configuration records more events and is useful while profiling an application.

In most cases, the preconfigured configurations are sufficient. However, there might be a scenario while analyzing a recording that some events are disabled by default. To enable these events, create a custom configuration. Use JDK Mission Control to configure the event settings by using one of the existing configurations. Make a copy of an existing configuration and then modify it. Don't modify the default configurations. Specify which configuration to use with the `settings` command-line option when starting a recording. For example:

```
-
XX:StartFlightRecording:filename=recording.jfr,dumponexit=true,settings=default.j
fc
```

```
-
XX:StartFlightRecording:filename=recording.jfr,dumponexit=true,settings=mysetting
s.jfc
```

Part IV

Monitor Events with Flight Recorder Event Streaming API

The Flight Recorder event streaming API enables you to consume Flight Recorder data continuously. This section shows you three ways you can do this:

- Create Event Stream in Process, Active: Creates an event stream at the same time a recording is created
- Create Event Stream in Process, Passive: Creates a passive stream that listens for events, but what gets recorded is controlled by external means
- Create Event Stream from External Process: Creates an event stream from a separate Java process

18

Create Event Stream in Process, Active

The sample `StreamEventsSample.java` creates an event stream at the same time a recording is created. An event stream is a sequence of events.

The class `RecordingStream` starts a recording and creates an event stream at the same time. The sample calls `Thread.sleep(1000)` three times, which creates three `jdk.ThreadSleep` events. The Event Streaming API prints the `jdk.ThreadSleep` events when they occur:

```
import jdk.jfr.Configuration;
import jdk.jfr.consumer.RecordingStream;

public class StreamEventsSample {

    public static void main(String... args) throws Exception {
        Configuration c = Configuration.getConfiguration("profile");
        try (RecordingStream rs = new RecordingStream(c)) {
            rs.onEvent("jdk.ThreadSleep", System.out::println);
            System.out.println("Starting recording stream ...");
            rs.startAsync();
            for (int i = 0; i < 3; i++) {
                System.out.println("Sleeping for 1s...");
                Thread.sleep(1000);
            }
        }
    }
}
```

Run `StreamEventsSample` with the following command:

```
java -XX:StartFlightRecording StreamEventsSample.java
```

It prints output similar to the following:

```
Started recording 1. No limit specified, using maxsize=250MB as default.

Use jcmd 7400 JFR.dump name=1 filename=FILEPATH to copy recording data to
file.
Starting recording stream ...
Sleeping for 1s...
Sleeping for 1s...
jdk.ThreadSleep {
  startTime = 00:26:42.463
  duration = 2.14 s
  time = 1.00 s
  ...
}

Sleeping for 1s...
```

```
jdk.ThreadSleep {
  startTime = 00:26:44.602
  duration = 1.04 s
  time = 1.00 s
  ...
}
```

Follow these steps to create an event stream from a recording with the `RecordingStream` class:

1. Optionally specify a predefined configuration ("default" or "profile") with the `Configuration` class.
2. Create a `RecordingStream` instance with either the `Configuration.getConfiguration()` or `Configuration.getConfiguration(Configuration)` method.
3. Optionally enable events that you want to include in the event stream with the `RecordingStream::enable(String)` method.
4. Specify actions to perform on events in the stream. To specify an action to perform on all events, use the `onEvent(Consumer<RecordedEvent>)` method. For example, the following statement prints the name of all events in the stream to standard output:

   ```
   rs.onEvent(e -> { System.out.println(e.getEventType().getName()); });
   ```

   Use the `onEvent(String, Consumer<RecordedEvent>)` to specify an action to perform on a specific event. For example, the following statement prints events whose name matches `jdk.ThreadSleep`:

   ```
   rs.onEvent("jdk.ThreadSleep", System.out::println);
   ```

5. Start the event stream with either the `start()` or `startAsync()` method. This sample calls `startAsync()`, which runs the stream in a background thread. If you call the `start()` method, then the application will not proceed pass this method call until the stream is closed.

19

Create Event Stream in Process, Passive

The sample `PassiveEventStreamSample.java` starts a passive event stream with the method `EventStream.openRepository()`. As with any event stream, a passive event stream listens for events; in this example, it listens for `jdk.CPULoad` events. However, what gets recorded is controlled by external means, for example, by the command-line option `-XX:StartFlightRecording`, the `jcmd` command `JFR.start`, or an API (for example, `Recording::start()`).

The sample `PassiveEventStreamSample.java` creates an event stream not with `RecordingStream` but with `EventStream.openRepository()`. An event stream requires a recording; this sample obtains it from the command-line option `-XX:StartFlightRecording`.

```
import java.util.concurrent.atomic.AtomicInteger;

import jdk.jfr.consumer.EventStream;

public class PassiveEventStreamSample {

    static int NUMBER_CPULOAD_EVENTS = 3;

    public static void main(String... args) throws Exception {

        AtomicInteger timer = new AtomicInteger();

        try (EventStream es = EventStream.openRepository()) {
            es.onEvent("jdk.CPULoad", event -> {
                System.out.println("CPU Load " + event.getEndTime());
                System.out.println(" Machine total: "
                    + 100 * event.getFloat("machineTotal") + "%");
                System.out.println(
                    " JVM User: " + 100 * event.getFloat("jvmUser") +
                    "%");
                System.out.println(
                    " JVM System: " + 100 * event.getFloat("jvmSystem") +
                    "%");
                System.out.println();
                if (timer.incrementAndGet() == NUMBER_CPULOAD_EVENTS) {
                    System.exit(0);
                }
            });
            es.start();
        }
    }
}
```

Run `PassiveEventStreamSample` with the following command:

```
java -XX:StartFlightRecording PassiveEventStreamSample.java
```

It prints output similar to the following:

```
Started recording 1. No limit specified, using maxsize=250MB as default.

Use jcmd 12352 JFR.dump name=1 filename=FILEPATH to copy recording data to
file.
CPU Load 2020-01-24T05:34:36.265584686Z
 Machine total: 19.3799%
 JVM User: 5.2175264%
 JVM System: 1.8634024%

CPU Load 2020-01-24T05:34:37.310049859Z
 Machine total: 5.2533073%
 JVM User: 0.0%
 JVM System: 0.3899041%

CPU Load 2020-01-24T05:34:38.373796070Z
 Machine total: 7.242967%
 JVM User: 0.0%
 JVM System: 1.1451485%
```

20

Create Event Stream from External Process

The sample `StreamExternalEventsWithAttachAPISample.java` creates an event stream from a separate Java process, the sample `SleepOneSecondIntervals.java`.

`SleepOneSecondIntervals` repeatedly sleeps for 1 second intervals; as demonstrated in Create Event Stream in Process, Active, every time `Thread.sleep()` is called, a `jdk.ThreadSleep` event occurs.

```
public class SleepOneSecondIntervals {

    public static void main(String... args) throws Exception {
        long pid = ProcessHandle.current().pid();
        System.out.println("Process ID: " + pid);
        while(true) {
            System.out.println("Sleeping for 1s...");
            Thread.sleep(1000);
        }
    }
}
```

`StreamExternalEventsWithAttachAPISample` uses the Attach API to obtain the virtual machine in which `SleepOneSecondIntervals` is running. From this virtual machine, `StreamExternalEventsWithAttachAPISample` obtains the location of its Flight Recorder repository though the `jdk.jfr.repository` property. It then creates an `EventStream` with this repository through the `EventStream::openRepository(Paths)` method.

```
import java.nio.file.Paths;
import java.util.Optional;
import java.util.Properties;

import com.sun.tools.attach.VirtualMachine;
import com.sun.tools.attach.VirtualMachineDescriptor;

import jdk.jfr.consumer.EventStream;

public class StreamExternalEventsWithAttachAPISample {
    public static void main(String... args) throws Exception {

        Optional<VirtualMachineDescriptor> vmd =
            VirtualMachine.list().stream()
            .filter(v -> v.displayName()
                .contains("SleepOneSecondIntervals"))
            .findFirst();

        if (vmd.isEmpty()) {
            throw new RuntimeException("Cannot find VM for
SleepOneSecondIntervals");
        }
```

```
        VirtualMachine vm = VirtualMachine.attach(vmd.get());

        // Get system properties from attached VM

        Properties props = vm.getSystemProperties();
        String repository = props.getProperty("jdk.jfr.repository");
        System.out.println("jdk.jfr.repository: " + repository);

        try (EventStream es = EventStream
            .openRepository(Paths.get(repository))) {
            System.out.println("Found repository ...");
            es.onEvent("jdk.ThreadSleep", System.out::println);
            es.start();
        }
    }
}
```

Compile `SleepOneSecondIntervals.java` and `StreamExternalEventsWithAttachAPISample.java`. Then run `SleepOneSecondIntervals` with this command:

```
java -XX:StartFlightRecording SleepOneSecondIntervals
```

In a new command shell, run `StreamExternalEventsWithAttachAPISample`:

```
java StreamExternalEventsWithAttachAPISample
```

It prints output similar to the following:

```
jdk.jfr.repository: C:\Users\<your user
name>\AppData\Local\Temp\2019_12_08_23_32_47_5100
Found repository ...
jdk.ThreadSleep {
  startTime = 00:15:31.643
  duration = 1.04 s
  time = 1.00 s
  eventThread = "main" (javaThreadId = 1)
  stackTrace = [
    java.lang.Thread.sleep(long)
    SleepOneSecondIntervals.main(String[]) line: 8
  ]
}

jdk.ThreadSleep {
  startTime = 00:15:32.689
  duration = 1.05 s
  time = 1.00 s
  eventThread = "main" (javaThreadId = 1)
  stackTrace = [
    java.lang.Thread.sleep(long)
    SleepOneSecondIntervals.main(String[]) line: 8
  ]
```

```
}
...
```

The sample `StreamExternalEventsWithJcmdSample.java` is similar to `StreamExternalEventsWithAttachAPISample` except it starts Flight Recorder for `SleepOneSecondIntervals` with the Attach API. With this API, the sample runs the command `jcmd <PID> JFR.start` with the PID of `SleepOneSecondIntervals`:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.file.Paths;
import java.util.Properties;

import com.sun.tools.attach.VirtualMachine;

import jdk.jfr.consumer.EventStream;

public class StreamExternalEventsWithJcmdSample {
    public static void main(String... args) throws Exception {
        if (args[0] == null) {
            System.err.println("Requires PID of process as argument");
            System.exit(1);
        }

        String pid = args[0];

        Process p = Runtime.getRuntime().exec(
            "jcmd " + pid + " JFR.start");

        printOutput(p);

        // Wait for jcmd to start the recording
        Thread.sleep(1000);

        VirtualMachine vm = VirtualMachine.attach(pid);
        Properties props = vm.getSystemProperties();
        String repository = props.getProperty("jdk.jfr.repository");
        System.out.println("jdk.jfr.repository: " + repository);

        try (EventStream es = EventStream
            .openRepository(Paths.get(repository))) {
            System.out.println("Found repository ...");
            es.onEvent("jdk.ThreadSleep", System.out::println);
            es.start();
        }
    }

    private static void printOutput(Process proc) throws IOException {
        BufferedReader stdInput = new BufferedReader(
            new InputStreamReader(proc.getInputStream()));

        BufferedReader stdError = new BufferedReader(
            new InputStreamReader(proc.getErrorStream()));
```

```
            // Read the output from the command
            System.out.println(
                "Here is the standard output of the command:\n");
            String s = null;
            while ((s = stdInput.readLine()) != null) {
                System.out.println(s);
            }

            // Read any errors from the attempted command
            System.out.println(
                "Here is the standard error of the " + "command (if any):\n");
            while ((s = stdError.readLine()) != null) {
                System.out.println(s);
            }
        }
}
```

Compile `SleepOneSecondIntervals.java` and `StreamExternalEventsWithJcmdSample.java`. Then run `SleepOneSecondIntervals` with this command:

```
java -XX:StartFlightRecording SleepOneSecondIntervals
```

It prints output similar to the following:

```
Started recording 1. No limit specified, using maxsize=250MB as default.

Use jcmd 5100 JFR.dump name=1 filename=FILEPATH to copy recording data to
file.
Process ID: 5100
Sleeping for 1s...
Sleeping for 1s...
Sleeping for 1s...
...
```

Note the PID for `SleepOneSecondIntervals` (in this example, it's 5100). While this sample is running, in a new command shell, run `StreamExternalEventsWithJcmdSample` with this command.

```
java StreamExternalEventsWithJcmdSample <PID of SleepOneSecondIntervals>
```

It prints output similar to `StreamExternalEventsWithAttachAPISample`.

Part V
Parsing a Recording File

The example `ParseRecordingFileSample.java` describes various ways you can parse a recording file. It starts a recording to record several `Hello` and `Message` events.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

import jdk.jfr.Event;
import jdk.jfr.EventType;
import jdk.jfr.Label;
import jdk.jfr.Name;
import jdk.jfr.Recording;
import jdk.jfr.consumer.EventStream;
import jdk.jfr.consumer.RecordingFile;

public class ParseRecordingFileSample {

    @Name("com.oracle.Hello")
    @Label("Hello World!")
    static class Hello extends Event {
        @Label("Greeting")
        String greeting;
    }

    @Name("com.oracle.Message")
    @Label("Message")
    static class Message extends Event {
        @Label("Text")
        String text;
    }

    public static void main(String... args) throws IOException {

        try (Recording r = new Recording()) {
            r.start();
            for (int i = 0; i < 3; i++) {
                Message messageEvent = new Message();
                messageEvent.begin();
                messageEvent.text = "message " + i;
                messageEvent.commit();

                Hello helloEvent = new Hello();
                helloEvent.begin();
                helloEvent.greeting = "hello " + i;
                helloEvent.commit();
            }
            r.stop();
            Path file = Files.createTempFile("recording", ".jfr");
```

```
            r.dump(file);

            try (var recordingFile = new RecordingFile(file)) {
                System.out.println("Reading events one by one");
                System.out.println("=========================");
                while (recordingFile.hasMoreEvents()) {
                    var e = recordingFile.readEvent();
                    String eventName = e.getEventType().getName();
                    System.out.println("Name: " + eventName);
                }
                System.out.println();
                System.out.println("List of registered event types");
                System.out.println("==============================");
                for (EventType eventType : recordingFile.readEventTypes())
                {
                    System.out.println(eventType.getName());
                }
            }
            System.out.println();

            System.out.println("Reading all events at once");
            System.out.println("=========================");

            for (var e : RecordingFile.readAllEvents(file)) {
                String eventName = e.getEventType().getName();
                System.out.println("Name: " + eventName);
            }
            System.out.println();

            System.out.println("Reading events one by one, printing only "
                    + "com.oracle.Message events");
            System.out.println("========================================="
                    + "=========================");

            try (EventStream eventStream = EventStream.openFile(file)) {
                eventStream.onEvent("com.oracle.Message", e -> {
                    System.out.println(
                        "Name: " + e.getEventType().getName());
                });
                eventStream.start();
            }
        }
    }
}
```

Run `ParseRecordingFileSample` with this command:

```
java ParseRecordingFileSample.java
```

When running `ParseRecordingFileSample`, you don't have to start Flight Recorder with the command-line option `-XX:StartFlightRecording`; the method `Recording.start()` starts it. `ParseRecordingFileSample` prints the following:

```
Reading events one by one
=========================
```

```
Name: com.oracle.Message
Name: com.oracle.Hello
Name: com.oracle.Message
Name: com.oracle.Hello
Name: com.oracle.Message
Name: com.oracle.Hello

List of registered event types
==============================
jdk.ThreadStart
jdk.ThreadEnd
jdk.ThreadSleep
...
jdk.X509Validation
com.oracle.Message
com.oracle.Hello

Reading all events at once
==========================
Name: com.oracle.Message
Name: com.oracle.Hello
Name: com.oracle.Message
Name: com.oracle.Hello
Name: com.oracle.Message
Name: com.oracle.Hello

Reading events one by one, printing only com.oracle.Message events
===================================================================
Name: com.oracle.Message
Name: com.oracle.Message
Name: com.oracle.Message
```

Write Recording Data to a File

`ParseRecordingFileSample` demonstrates several ways you can parse a recording file. However, you first need a recording file, and this sample doesn't create one at the command line. Instead, it calls `Recording.dump(Path)` to write recording data to a temporary file:

```
Path file = Files.createTempFile("recording", ".jfr");
r.dump(file);
```

Note that the recording must be started but not necessarily stopped.

Read Events One by One

Use this technique for large recordings and if you need to access metadata.

The method `RecordingFile.readEvent()` reads the next event in the recording while `RecordingEvent.hasMoreEvents()` returns `true` if unread events exist in the recording file:

```
while (recordingFile.hasMoreEvents()) {
    var e = recordingFile.readEvent();
    String eventName = e.getEventType().getName();
```

```
    System.out.println("Name: " + eventName);
}
```

List Registered Event Types

The method `RecordingFile.readEventTypes()` returns a list of all event types in the recording.

Read All Events at Once

Use this technique for smaller recordings that fit in memory.

The method `RecordingFile.readAllEvents(Path)` returns a list of all events in the recording file. It's intended for small recording files where it's more convenient to read all events in a single operation. It's not intended for reading large recording files.

Read Only Specific Events with Event Streaming API

To process only specific events, you could read events one by one with `RecordingFile.readEvent()`, as described previously, then check the event's name. However, if you use the event streaming API, then event objects of the same type are reused to reduced allocation pressure.

This technique involves creating an event stream with `EventStream.openFile(Path)`, then calling `EventStream.onEvent(String eventName, Consumer)` to register an action that will be performed if `eventName` matches the event's name:

```
try (EventStream eventStream = EventStream.openFile(file)) {
    eventStream.onEvent("com.oracle.Message", e -> {
        System.out.println("Name: " +
            e.getEventType().getName());
    });
    eventStream.start();
}
```