JSR 326 Post mortem JVM Diagnostics API

First Early Draft Review

User manual and specification - 2010-01-22

JSR 326 Post mortem JVM Diagnostics API:

First Early Draft Review

User manual and specification - 2010-01-22

Published 2010-01-22

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Warning

This is a living document. Make sure you have the latest version. Other versions are available from the Apache Kato Wiki page http://cwiki.apache.org/KATO/jsr326specification.html or by checking out and building the Apache Kato project.

Note also that this is an early draft document and as such there are many areas open to improvement. There are also areas that are specifically not completed. These areas cover parts of the API or its usage that still need to be defined.

Audience

This document is intended to be used by consumers and implementors alike. It is expected that the reader will have good knowledge of the Java™ programming language.

Related Documentation

More information about the Apache Kato incubator can be found by visiting the main website at http://incubator.apache.org/kato/site/index.html

How to comment on the specification

Comments on this document and its contents can be made using the kato-spec mailing list hosted by the Apache Software Foundation.

Subscribe to the mailing list by sending an email to `<kato-spec-subscribe@incubator.apache.org>` Once subscribed you can send an email to the kato-spec mailing list by addressing your email to`<kato-spec@incubator.apache.org>`

Trademarks

References

Home Page for Apache Kato Incubator Project *http://incubator.apache.org/kato/site/index.html*

JSR 326 at JCP.org *http://jcp.org/en/jsr/detail?id=326*

Chapter 1. Introduction

What is JSR 326?

JSR 326 is intended to be a Java™ API specification for standardising how and what can be retrieved from the contents of post-mortem artefacts - typically process and JVM dumps.

Unusually for new APIs, this project will endeavour to encompass the old and the new, since diagnostic solution that only works when users move to the latest release does little to improve diagnosability in the short term.

This project will consume existing dump artefacts as well as possible while developing an API that can address the emerging trends in JVM and application directions. The most obvious of these trends are the exploitation of very large heaps, alternative languages and, paradoxically for Java™, the increased use of native memory through vehicles such as NIO.

What is Apache Kato?

Project Kato is intended to be the place where the Specification, Reference implementation (RI) and Technology Compatibility Kit (TCK) are openly created. The intention is that the Specification and RI will be developed in tight unison, guided by a user-story-focused approach to ensure that real-world problems drive the project from the beginning.

This project is about bringing together people and ideas to create a common, cross industry API, and we can't think of a better place to do that than in Apache.

IBM developed an API called DTFJ ("Diagnostic Tool Framework for Java") as a means of providing its support teams a basis on which to write tools to diagnose Java SDK and Java application faults. It consists of a native JVM-specific component and the DTFJ API, which was written in pure Java.

In 2009 IBM donated the implementation independent portions of DTFJ to the Apache Kato project

Rationale

JSR 326 exists because of the widely acknowledged limitations in diagnosing Java™ application problems after the fact. There are many good ways to understand and diagnose problems while they happen, but few credible or pervasive tools exist for helping resolve problems when it has all gone suddenly and horribly wrong.

Outside of "live monitoring" there is no standard way to provide diagnostics information, and hence no standard tools. Each tool writer has to figure out how to access the data individually and specifically for each JVM vendor and operating system. This sparsity of tools has meant that users have limited options in diagnosing their own problems, especially unexpected or intermittent failures.

These users turn to the providers of their software to work out what is happening. Consequently application, middleware, and JVM vendors are spending increasing time supporting customers in problem diagnosis.

Emerging trends indicate that this is going to get worse.

Today JVM heap sizes are measured in small numbers of gigabytes, processors on desktops come in twos or fours, and most applications running on a JVM are written in Java™. To help analyse problems in these configurations, we use a disparate set of diagnostic tools and artefacts.

If the problem can't be reproduced in a debugger, then things quickly get complicated. There are point tools for problems like deadlock analysis or the ubiquitous Java[tm] out-of-memory problems, but overall the Java[tm] diagnostic tools arena is fragmented and JVM- or OS-specific. Tool writers have to choose their place in this matrix.

We want to change that by removing the need for tool writers to make a choice.

By enabling tool writers to easily target all the major JVM vendors and operating systems, we expect the number and capability of diagnostic tools to greatly increase.

Tomorrow it gets harder; heap sizes hit 100's of gigabytes, processors come packaged in powers of 16, and the JVM commonly executes a wide range of language environments.

We can't tackle tomorrow's problems until we have a platform to address today's.

Post Mortem versus Live Monitoring

It's important to understand what the term "post mortem" means as far as JSR 326 is concerned and how it fits within the general Post-mortem versus Live Monitoring space.

For JSR 326 the term "post mortem" is used loosely: it does not just imply dead Java Virtual Machines and applications; JSR 326 also covers living, breathing applications where the dump artefacts are deliberately produced as part of live monitoring activity.

Live monitoring generally means tracing, profiling, debugging, or even bytecode monitoring and diagnosis by agents via the java.lang.instrument API. It can be a surprise to understand that it can also mean analysis of dumps to look for trends and gather statistics.

The live-monitoring diagnostic space is well served except for this last area. The mutation speed of modern applications under load can sometimes mean that monitoring systems cannot keep pace since they need to do complex analysis *on-the-fly* The obvious solution is to take a snapshot of a running system and analyse the results *off-line*

JSR 326 can help with this "Snapshot monitoring" by providing a standard mechanism for generating a snapshot and for reading the contents of the snapshot later.

For JSR 326 "Post Mortem" just means "after the fact"

Tell me more about "Diagnostic Artifacts"

Simply put, when something goes wrong you'd like to know why. A diagnostic artifact is whatever material is left when your application or JVM fails. Sometimes it's a message to the console, or a record in a log file. Hopefully you'll get enough information to figure out what happened and fix the problem.

Unfortunately there are many cases where you don't get to see the obvious “smoking gun.”

In those situations you need access to more information so you can dig into the causes of your problem. Historically the sorts of artifact you need are split into two types: those which show a time element and those that are a snapshot of working system. The former of these types is of course a trace, while the latter comes under the term “dump” or “core file” .

It's these latter type that JSR 326 is designed to consume.

What types of Dump are supported by this API?

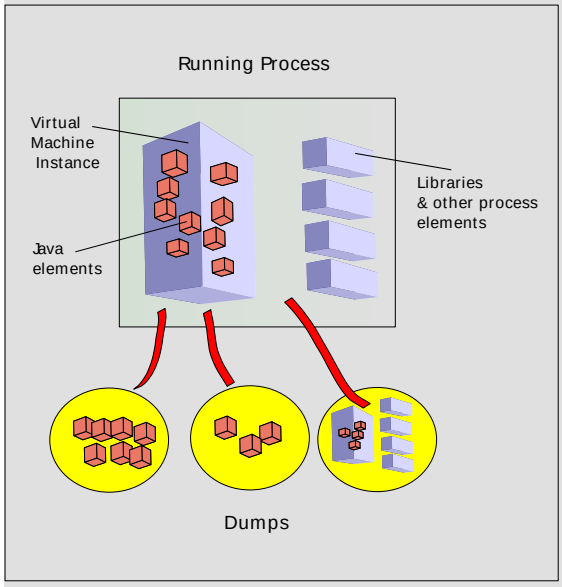
The Apache Kato incubator project is developing the reference implementation for JSR 326. That work includes the development of an implementation that can read standard binary HPROF files and an experimental new dump that uses JVMTI to expose more information than is currently in a HPROF file.

Other JVM vendors can develop implementations to support the API for their own dumps.

What data can I expect to find in a Dump?

Dumps come in all shapes and sizes. There is no definitive statement of contents. The API is designed to accomodate this fact by providing a mechanism to signal that data is not available. Note that the design of the API to handle data optionality is still not completed.

Figure 1.1. Dump contents



Its still possible to determine broad categories for the contents of a dump. In Figure 1.1, “Dump contents” you can see that its reasonable to have three categories - from the dump which has all process information down to the dump which only contains objects from the Java heap.

Chapter 2. Application Programming Interface Architecture

Introduction

This chapter describes the conceptual behaviour and structure of the API

Structure

JSR 326 consists of a series of Java packages listed below

Table 2.1. Packages

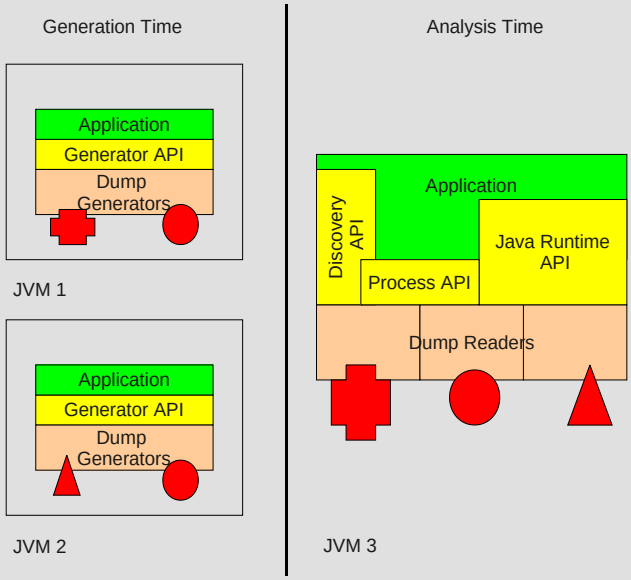
Name	Title
javax.tools.diagnostics	Top level of the API, provides access to API implementations.
javax.tools.diagnostics.image	Package of interfaces representing a snapshot of a program.
javax.tools.diagnostics.runtime	Definition of an abstract view of a managed runtime.
javax.tools.diagnostics.runtime.java	Definition of an abstract view of a Java runtime.
javax.tools.diagnostics.vm	Dump Creation API
javax.tools.diagnostics.vm.spi	
javax.tools.diagnostics.vm.spi.delegates	

These packages are separated into into four logical groups.

- Diagnostic Artifact creation
- Abstraction handler discovery
- Runtime abstraction
- Process abstraction

These logical separations are themselves grouped into two further distinctions: those available at creation time and those available at analysis time.

Figure 2.1. Summary Architecture



This separation can be seen in the Figure 2.1, “Summary Architecture”. In this diagram the green layer represents the user application, the yellow layer represents the API and the orange layer represents implementations in support of the API. The red shapes represent the different types of diagnostic artifact that can be created.

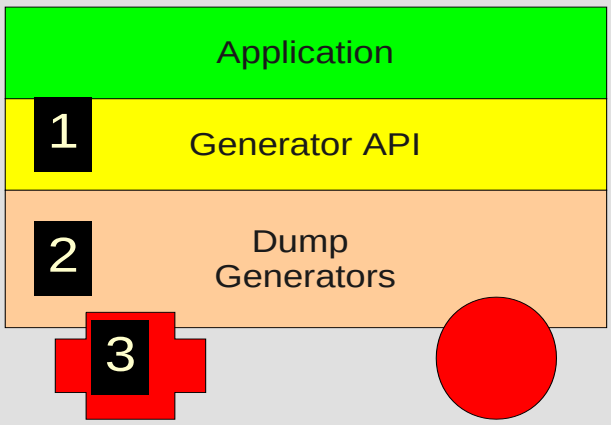
There is a vertical separation in this diagram which shows the two phases of the API. The left hand side shows two distinct JVM instances which is intended to indicate the capability of the API to be used on multiple implementations in a standard way. The right hand side shows a single application consuming multiple diagnostic artifacts.

In the following sections we'll explore the API in more detail.

Creation Time Architecture

Creation time is the term used for the phase when dumps are generated. Dump creation can be via the creation time API as outlined here, or via a JVM implementation specific mechanism.

Figure 2.2. Creation Time Architecture



In Figure 2.2, “Creation Time Architecture” the green layer represents the application, the yellow boxes represent the API and the orange boxes represent implementations that translate the dump creation request to the specific actions required to create a diagnostic artifact.

The numbered parts of the diagram can explained as followed

1. Creation API: This part of the API provides a standard mechanism for an application to initiate the creation of a diagnostic artifact

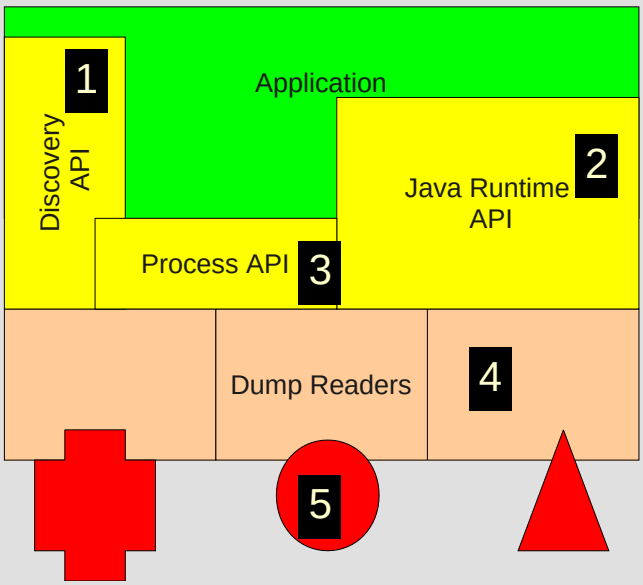
2. Dump Creators: The specific implementations that are registered within the JVM that provide the mechanisms to create the diagnostic artifact

3. Diagnostic Artifacts: The artifacts produced

The relationship between implementations and the discovery mechanism is not shown in this diagram. See the section on the discovery process for more information

Analysis Time architecture

Figure 2.3. Analysis Time Architecture



In Figure 2.3, “Analysis Time Architecture” the green layer represents the API, the yellow boxes represent implementations that translate from specific dump formats to the API structure and the red boxes represent the various types of dump that can be found.

The relationship between implementations and the discovery mechanism is not shown in this diagram. See the section on the discovery process for more information

Chapter 3. Design Principles

Introduction

This chapter describes the principles that are common across the whole API.

Lists

Methods use `java.util.List` to return multiple objects. All lists are immutable and unsynchronised.

For example, an `ImageProcess` may contain multiple instances of `ManagedRuntime`. To retrieve the `ManagedRuntime` instances, a call would be made to: `List<ManagedRuntime> ImageProcess.getRuntimes()`.

Lists are sometimes used in the API to access a larger number of objects than would be used in most Java applications. For this reason, use of the `java.util.List.toArray()` method is discouraged in situations where there would be a large number of array elements. For example:

```
JavaObject[] heapAsArray(JavaHeap heap) {
 return heap.getObjects().toArray(new JavaObject[0]);
}
```

would return an array with all of the objects in the heap, perhaps numbering in the hundreds of millions. This should be considered when implementing or calling the API.

Type names

Type names and signatures use the same format as JNI. Please see the JavaDoc for each method for the exact formats.

The class `java.util.Map.entry` would be formatted like so:

```
java/util/Map$Entry
```

An multidimensional array `java.util.Map.entry[][]` is formatted like:

```
[[Ljava/util/Map$Entry;
```

A primitive array class for `int[][]` is formatted like so:

```
[[I
```

Memory and Identification

Address space

Memory in the API consists of a collection of flat (i.e. not segmented) address spaces represented by the `ImageAddressSpace` class. Implementations do not have to report any memory as being present in the snapshot.

Memory sections

The `ImageSection` interface is used to describe arbitrary areas of memory by returning a pointer (represented by a `ImagePointer` instance) and a size along with a name. `ImageAddressSpace` instances use them to describe what memory is mapped in a snapshot. They are also used to describe the memory layout of the entities that the API interfaces represent. This is normally done with methods such as `List<ImageSection> getSections()`. The actual `ImageSection` instances returned are implementation specific and it is acceptable for there to be none. It is expected that they will be used for:

- determining the memory occupancy of items. For example, the heap size could be derived from `JavaHeap.getSections()`.
- accessing structures in memory. For example, `JavaObject.getSections()` will return all of the `ImageSection` instance representing the memory a `JavaObject` occupies. With knowledge of how an object is laid out on the heap, it would be possible to retrieve more information than is retrieved presented by the API.

Addresses and Identification

The API uses the `ImagePointer` interface to identify objects returned by the API. `ImagePointer` represents an address in memory, and enables programs to access memory at that address and at offsets from that address. Given that not all implementations of the API allow access to memory, the addresses returned could be entirely artificial.

When an `ImagePointer` is used as an address of an object from the API, it is up to the implementation to decide what it is actually pointing at. It is important though that objects of the same type have unique addresses. For example, `JavaObject` instances much each return an `ImagePointer` different from all other, but there may be instances of `JavaClass` that share the same address.

`ImagePointer` allows access to memory using Java types, corrected for endianness. This means that only twos-complement values can be returned, apart from the `char` which is unsigned in Java. There is no conversion from the native platform's floating point formats. Floating point values are assumed to be stored as Java floating point types.

Package Separation

While aspects of the `image` API are used by the `runtime.java` API, the reverse will never occur. The `image` API should, in principle, be implementable standalone. For example, the following is allowed as it refers to the image package:

```
ImageThread JavaThread.getImageThread()
```

But the converse is not:

```
JavaThread ImageThread.getJavaThread()
```

Error Handling

Implementations of the API should present data as accurate and complete as reasonably possible under any circumstances. The purpose of the API includes presenting the state of a running process, a Java Virtual Machine, when it encountered abnormal conditions. The most extreme of these situations is when the native code implementing the JVM itself has crashed. As such, there will be situations where information cannot be retrieved

or may be incorrect. This should be regarded as normal. Both implementors of the API and those calling the API should code anticipating errors to be normal rather than exceptional conditions.

Data is retrieved from the API from two types of methods. Those returning multiple items using generisised Lists and those returning items directly. The lists that are returned are expected to return all items that they can.

When implementing lists, implementors should take care to:

- ensure that lists have a finite number of items. For example, a corrupted linked list may be corrupt or terminated incorrectly. The API implementation should detect this and terminate the list.
- process as little of the items being returned as possible. Better to continue reading the collection of items rather than fail on one item that is slightly wrong. For example, if the objects are being retrieved using the list from the following method call:

  ```
  List<JavaObject> JavaHeap.getObjects()
  ```

  ... then if one object fails to identify its type properly, it is expected that the list would return the `JavaObject`. Calls to that `JavaObject` would fail appropriately, such as to `JavaClass JavaObject.getJavaClass()`.

Errors are reported on methods returning single items (i.e. not lists) using exceptions. There are two exceptions, both subclasses of `DiagnosticException`. The exceptions are:

- `MemoryAccessException`. This is thrown when an attempt has been made to access memory that is not present in the snapshot.
- `CorruptDataException`. This is thrown when the data used to form a response to the method call is incorrect.

Optional and missing data

There are circumstances where information cannot be supplied. Methods that throw the exception `DataUnavailable` will do so if the information is either not presentable by the implmentation of the API, or if it is not available for that particular snapshot.

There are circumstances where `null` is returned by a method. These circumstances will be explicitly documented in the Javadoc.

`DataUnavailable` is thrown when the API *cannot* return the requested data. `null` is returned when the data was never there to be returned.

Methods that return `java.util.List` instances will always do so under all circumstances.

Faked objects

There are many possible implementations of a Java™ Virtual Machine each of which can have various and different optimisations. Mapping a particular implementation to this API may require the creation of synthetic objects for entities which do not actually exist in the diagnostic artifact.

An example of this would be array classes. These classes are never loaded by a Java™ Virtual Machine, they are constructed as and when they are necessary. It is conceivable that there would be no actual entities that could directly correspond with `JavaClass` instances. In circumstances like these the API implementor would have to create a `JavaClass` for the array class, as that is the only means the API has for identifying that objects type.

Faked objects should be implemented carefully. For instance, if a faked `JavaMethod` is created, then the class it declares itself as belonging to should report it, otherwise inconsistencies can arise that could cause calling programs to fail.

There should be no collisions between real and faked objects.

Implementations should not be misleading. If a faked object has been created, then related fack objects should be kept to a minimum. For instance a faked `JavaObject` should not return `ImageSection` instances.

Object identities

All implementations should override the `java.lang.Object.equals(Object)` method when objects are not permanently cached and may need to be recreated. All API's should use `equals` to test object identity.

The quantity of objects held within a diagnostic artifact normally means that it is impractical to keep an in-memory instance for everything. Therefore the API does not require that repeated calls to return a specific object will in fact return the same instance. *The API allows for recreation of already requested objects*

The behaviour of `equals` against objects from different snapshots is not defined.

Chapter 4. Topics not yet resolved

This chapter outlines the areas of the specification and implementation etc that need to be resolved before the API can be completed. In some cases the API definition *so far* has a solution that is not optimal and may change in the future.

Optionality

Diagnostic artifacts are not equal. Generally they do not contain the same set of information. Since it's likely that no single artifact will ever satisfy all the data requirements of the API the design of the API must therefore be amenable to missing data. In fact there are two cases of data being missing. The case where that data is *never* in the artifact (for instance an HPROF dump does not contain any information what interfaces a class may implement) and the case where in a particular instance of a the artifact that data is just missing (or corrupted).

Currently the API signals missing or corrupt data by throwing the relevant exception. There is no mechanism for the user of the API to discover what types of data are present or missing from the artifact. This is a problem for the user and for the Test Compliance Kit since it needs to have predictable contents.

Native and Java Frame interleaving

The API presents native stack frames and Java stack frames in different places in the API - in ImageThread and JavaThread. This makes it difficult to understand the order in which Java methods and native functions have been called. Exposing the interleaving of native and Java frames would help, for example, debugging complex JNI functions.

Optimisation of data access (query support)

The current programmatic means of accessing data is not open to optimisation. Having a query language would enable the creation of useful indexes to speed up queries.

No support for identifying or handing generics in the Java Runtime

Currently it is impossible to determine any of the generic type information that is available in JVMs implementing Java 5.0 or above.

No support for thread groups in the Java Runtime

The API has no consistent means of reporting the ThreadGroups the JavaThreads belong to.

Lack of consistency in accessing JavaObjects

In order to access the contents of objects in most cases it is often necessary to implement, as accesses via the API, at least a subset of the functionality of the methods in the class of the object being accessed. For example, to access all of the objects in a java.util.HashMap, knowledge of how HashMap is implemented is necessary. As HashMap can be implemented differently on different implementations of the Java SDK, it is difficult to write programs using the API that are truly Java SDK agnostic.

Need defined behaviour on what toString offers on each part of the API

The toString() method's behaviour is not specified in sufficient details across the whole API. For example, JavaStackFrame.toString() should return a string describing the stack location like in a Java stacktrace, so this should be specified so it can be implemented consistently.

No definitions about snapshoting

It is desirable that there be a consistent means for generating dumps for later analysis by the API. An API to be used during runtime would enable applications to generate dumps when it suits them. This would not be to the exclusion of other means of generating dumps peculiar to particular JVM implementations.

Chapter 5. Examples

Introduction

This chapter contains snippets of code demonstrating how to call the API. The appendices contain the unedited samples.

Opening Images

`Image` instances are obtained using the `javax.tools.diagnostics.FactoryRegistry` class. See Appendix B, *Opening Images example* for a complete example.

Programs using the API can obtain an Image like so:

Example 5.1. Opening an Image

```
Image image = FactoryRegistry.getDefaultRegistry().
        getImage(new File("example.file"));
```

`FactoryRegistry` uses javax.imageio.spi.ServiceRegistry as a registry of all API implementations known to the JVM. Implementors should ensure that `FactoryRegistry` is able to see their `ImageFactory` by placing their implementation in a jar file that contains a file called `META-INF/javax.tools.diagnostics.image.ImageFactory` that contains a line of text that is the name of the ImageFactory implementation, such as

```
com.example.dump.ImageFactoryImpl
```

Determining Snapshot Cause

Snapshots can be generated for a variety of reasons. The API can report that a snapshot has been generated because the JVM received a POSIX type signal, whether it was synchronous or asynchronous. For example, if a JNI library causes a `SIGSEGV` when running, this might be detectable through the API. In most cases, it should be possible through `ImageAddressSpace.getCurrentProcess()` and `ImageProcess.getCurrentThread()` to determine which process and which thread caused the snapshot.

If a snapshot was generated for a reason other than a POSIX signal being received, then the reason has to be derived through knowledge of the JVM implementation. For example, if an option was passed to the JVM to generate a snapshot on entry to a particular method, all of the stack traces could be searched to determine if that method was present, and therefore probably causing the snapshot. Likewise, detecting a call to `abort()` in the native stack would suggest that the snapshot was caused by a synchronous SIGABRT.

The following examples are extracts from the example program in Appendix C, Snapshot Cause Example.

The examples implement the `ImageAnalyzer` interface, they just need to implement the `analyze(Image)` method.

Example 5.2. CauseAnalyzer Class declaration

```
public class CauseAnalyzer implements ImageAnalyzer {

 @Override
 public void analyze(Image image) {
```

The containing `ImageAddressSpace` instances are searched for a current process. It is probable that there will be one address space and one process. If `getCurrentProcess()` returns `null`, there was no current process. Because `List` is returned, we can use a for-each loop.

Example 5.3. Find Current Process

```
for (ImageAddressSpace as : image.getAddressSpaces()) {
 ImageProcess process = as.getCurrentProcess();

 if (process != null) {
```

Once found, the process can be queried for signal information and thread information. If a signal was raised, `ImageProcess.getSignalName()` will not be null. The example reports the signal name and number to the user.

Example 5.4. Reporting signal information

```
    int signum = process.getSignalNumber();
    String signame = process.getSignalName();

    if (signame != null) {
     System.out.println("Snapshot caused by signal " + signame+"("+signum+")");
    }
```

The `ImageProcess` can report the command line. This is the command name and arguments that were used to start the process. The command and arguments are returned in a single string, separated by spaces.

The process ID returned by `getID()` in a String. This is implementation specific, and so could be in any format, whether that be hexadecimal, decimal or some other arbitrary string.

Example 5.5. Process ID and commandline

```
    System.out.println("Process "+process.getID()+
      " was started with `" +
      process.getCommandLine()+"'");
```

The program determines which thread caused the snapshot to be generated. If the current thread isn't null, the thread is identified by ID and its properties (implementation dependent).

Example 5.6. Thread identification

```
    ImageThread thread = process.getCurrentThread();
    if (thread != null) {
     System.out.println("\nSnapshot caused by thread "+
       thread.getID()+
       ", "+thread.getProperties());
```

Next, the thread's stack frames are printed out. The code relies on `java.lang.Object.toString()` being implemented correctly. It is expected that the most recent frame will be returned first.

Example 5.7. ImageThread stack trace

```
     for(ImageStackFrame frame: thread.getStackFrames()) {
      System.out.println("\t" + frame);
     }
```

This section of code relies on the relationship between `JavaThread` instances and `ImageThread` instances to determine which Java[tm] thread caused the snapshot. As the relationship is one-way, from Java[tm] to Image, all of the `JavaThread` instances have to be queried. Because `JavaThread.getImageThread()` might be `null`, `java.lang.Object.equals` is executed against the `ImageThread` which we know to be not null. Once the `JavaThread` is found, its name can be printed.

Example 5.8. JavaThread/ImageThread correlation

```
RUNTIME:    for(ManagedRuntime runtime : process.getRuntimes() ) {
       if (runtime instanceof JavaRuntime) {
        JavaRuntime jr = (JavaRuntime) runtime;

        for(JavaThread jthread : jr.getThreads()) {
         if (thread.equals(jthread.getImageThread())) {
          System.out.println("\nSnapshot caused by JavaThread "+
            jthread.getName());

          for(JavaStackFrame frame : jthread.getStackFrames()) {
           System.out.println("\t" + frame);
          }
          break RUNTIME;
         }}}}
```

Identifying Java[tm] VM

This example demonstrates how the JVM that generated a snapshot might be identified. The following information is reported using the image and java APIs:

- hostname of the machine the snapshot was generated on.
- The process ID.
- The command line.
- The executable that was running the Java[tm] program (e.g. "/usr/bin/java").
- The command line.
- The loaded native libraries.
- The version reported by the JVM.
- The options passed to the JVM.

The complete listing is in Appendix D, *Identifying Java VM Example*.

Like the previous example, this implements `ImageAnalyzer`. There is some functionality that is also present in the previous example - it is not repeated here.

Example 5.9. Declaration of WhatAnalyzer class

```
public class WhatAnalyzer implements ImageAnalyzer {

 @Override
 public void analyze(Image image) {
```

The hostname of the machine where the snapshot was generated is printed out. This isn't information that is necessarily available in most core dump formats, instead this would normally be recorded by the program. As it is important to get out as much information as possible, calls to the API are made with try/catch blocks around each individual method. This is necessary as exceptions should be expected to be raised under most circumstances. The stack traces for `DataUnavailable` are not usually reported as this is not an error condition. Instead, a message is inserted to indicate that the information is not known.

Example 5.10. Getting the hostname

```
  // Report the hostname.
  String hostname;
  try {
   hostname = image.getHostName();
  } catch (DataUnavailable e) {
   hostname = "<Could not retrieve hostname";
  } catch (CorruptDataException e) {
   hostname = "<Error retrieving hostname>";
   e.printStackTrace();
  }
  System.out.println("Snapshot was generated on " + hostname);
```

Here the executable that started the process is reported. This is the executable that launched the JVM - typically this is the "java" program. Alternatives include "javac" and "appletviewer". This is the name of the executable the operating system loaded into memory when creating the process.

Example 5.11. Executable name

```
    String executable;
    executable = process.getExecutable().getName();
    System.out.println("Process Executable "+ executable);
```

This code prints out the names of the libraries that were loaded by the process. This should include any JNI libraries that were configured. Note that it is also possible to determine where in memory these libraries have been loaded into memory using the `getSections()` method. This can be used to identify where a thread might have crashed.

Example 5.12. Process libraries

```
    System.out.println("Loaded Libraries:");
    for(ImageModule module : process.getLibraries()) {
     System.out.println("\t" + module.getName());
    }
```

The following code reports the Java[tm] VM version. This is implementation dependent, but is expected to contain more than just the version of Java[tm] that is supported, but actually identify which implementation of the JVM it is.

Example 5.13. Java[tm] VM Version

```
public void analyzeRuntime(JavaRuntime jr) {
 try {
  System.out.println("Java VM version:`"+jr.getVersion()+"'");
 } catch (CorruptDataException e1) {
  System.out.println("Error retrieving Java VM version");
  e1.printStackTrace();
 }
```

The following code reports the options that were passed to the JVM when it was created. The options are generated and passed on by the executable the launches Java. Some of these might be passed on the command line (such as by the "java" executable), but might also include options taken in from configuration files, as well as being generated by the launcher itself.

Example 5.14. Java[tm] VM Options

```
 System.out.println("VM options:");
 for (JavaVMOption option : jr.getJavaVMInitArgs().getOptions()) {
  String optionString = "\t\t\""+option.getOptionString()+"\"";

  ImagePointer extra = option.getExtraInfo();
  if (extra != null) {
   optionString += ", extraInfo=0x"+Long.toHexString(extra.getAddress());
  }

  System.out.println(optionString);
 }}
```

Retrieving Object Fields

This example demonstrates how object instance fields and array elements are accessed using the API. The complete listing is in Appendix E, *Retrieving Object Fields Example*.

The `analyzeRuntime` method walks over the heaps within the JVM. While Java[tm] programmers will be used to the concept of the heap, the API allows a number of heaps to be accessed in a single JVM. It is expected the different heaps will have different garbage collection policies and that each heap will be identified with a descriptive name through `JavaHeap.getName()`. The number of heaps and their names is implementation specific, but there must be at least one in a running JVM.

Example 5.15. Iterate over heaps

```
public void analyzeRuntime(JavaRuntime jr) {
 for (JavaHeap heap : jr.getHeaps()) {
   walkHeap (heap);
 } }
```

This section of code retrieves each object from a heap. For API implementations backed by a core file, the objects will probably be retrieved in order from lowest address in memory to the highest, but there is no relationship between `JavaObject` list indexes and the results of `JavaObject.getID()` that can be relied upon by callers of the API.

The `JavaObject` retrieved has to be tested to see if it is an array or an ordinary object as they are handled differently.

Example 5.16. Iterate over Objects

```
public void walkHeap(JavaHeap heap) {
 for (JavaObject jObject : heap.getObjects()) {
  if (jObject.isArray()) {
   walkArray (jObject);
  } else {
   walkObject (jObject);
  }
 }
```

This method takes a `JavaObject` and prints out the values of all of the instance fields (not the static fields). To identify each object, its ID is used. This is turned into a hex string using the `pointerToHexString(ImagePointer)` that is included in this example.

Example 5.17. Print object fields

```
public void walkObject(JavaObject jObject) {
 System.out.println("JavaObject @ " + pointerToHexString(jObject.getID()));
```

Each object in an instance of a class, so here the `JavaClass` is retrieved. This is equivalient to the following in Java:

```
Class java.lang.Object.getClass();
```

Implementors should ensure that the API returns the equivalent `JavaClass`.

Example 5.18. Get the object's type

```
 JavaClass clazz;
 clazz = jObject.getJavaClass();
```

A class will only report its fields, the superclasses must be retrieved in order to retrieve *their* fields. This `while` loop retrieves each superclass until the superclass is null, which will be returned by the `java.lang.Object` class. The class name is printed out, which should match what `java.lang.Class.getName()` would return, except for "." characters being replaced by "/".

Example 5.19. Iterate up class hierarchy

```
 while (clazz != null) {
  System.out.println( prefix + clazz.getName() +":");
  prefix += "  ";
```

This code retrieves each field from a `JavaClass`. This is equivalent to the following method in Java[tm] reflection:

```
Field[] java.lang.Class.getDeclaredFields()
```

This should return all fields, even synthetic fields. The `DiagnosticException` is a superclass of `CorruptDataException` and `DataUnavailable`.

Example 5.20. Print out each field

```
   for (JavaField nextField : clazz.getDeclaredFields()) {
    printField(prefix, nextField, jObject);
   }
  } catch (DiagnosticException e) {
   System.err.println("Error printing out fields.");
   e.printStackTrace();
  }
```

Here the next superclass is retrieved. This will return `null` if the class has no superclass, such as `java.lang.Object`. The loop is terminated by the `break` if the superclass couldn't be retrieved.

Example 5.21. get next superclass

```
  clazz = clazz.getSuperclass();
 } // while (class != null)
```

This method demonstrates how to print out an instance field. Note that the `JavaObject` is passed as it must passed on to the `JavaField` for it to retrieve the value of the field in that instance.

Example 5.22. Print fields class

```
private void printField(String prefix, JavaField field, JavaObject object)
```

It is not worth printing out the class fields for each instance of the class on the heap, so the field is tested to see if it is static. The following method call retrieves the modifiers (`public`, `static`, `protected`, etc.) from the `JavaField` and then uses reflection to test for `static` being set. Callers of the API should not assume that `JavaField.getModifiers()` only returns the bits defined in `java.lang.reflect.Modifier` - always test with the appropriate bitmasks or use the functions provided in `Modifier`.

Example 5.23. Testing JavaField.getModifiers()

```
if (java.lang.reflect.Modifier.isStatic(field.getModifiers()))
 return;
```

There are a number of methods provided by `JavaField` to retrieve the field value. The most generic is `JavaField.get(JavaObject)` which returns an `Object`.

Example 5.24. Getting the value of a field

```
Object fieldValue = field.get(object);
```

Object references that were `null` in the running program are also returned as `null` by the API.

Example 5.25. JavaField.get() returns null

```
// Format the field's value.
if(fieldValue == null) {
 valueString = "<null reference>";
```

As `JavaField.get(Object)` can return any type, primitive fields values are returned in instances of `Number` or `Character`. For instance, an `int` would be returned as an instance of `java.lang.Integer`. This can't be confused with fields that are references to `java.lang.Integer` instances as they would be represented by `JavaObject`.

Example 5.26. Boxed numbers

```
} else if (fieldValue instanceof Number) {
 valueString = fieldValue.toString();
} else if (fieldValue instanceof Character) {
 valueString = "`" + (Character)fieldValue + "'";
} else if (fieldValue instanceof Boolean) {
 valueString = ((Boolean) fieldValue).booleanValue() ? "true" : "false";
```

`JavaField.get()` is the means by which references to other objects are also retrieved. This program just retrieves the referred object's class name and its ID. It is important to remember that the signature of the field is expected to be an appropriate type for the objects that can be retrieved from it. A field signature would be either the same type as an object retrieved from it, an interface or super-interface, or a super class.

Example 5.27. Retrieving an object reference

```
} else if (fieldValue instanceof JavaObject) {
 JavaObject reference = (JavaObject) fieldValue;

 valueString = reference.getJavaClass().getName() + ": @ " + pointerToHexString(reference.getID());
```

The following code tests the object type to see if it is a Java[tm] String instance. The `JavaClass` representing `java.lang.String` could be cached and compared against the objects classes, but instead we compare against the name of the object's class. The method `JavaField.getString()` is used to retrieve the `JavaObject` as an instance of `java.lang.String` in the running JVM.

Example 5.28. Retrieving a string field

```
 if ("java/lang/String".equals(reference.getJavaClass().getName())) {
  valueString += valueString +" = \"" + field.getString(object) + "\"";
 }

}

System.out.println(prefix + field.getSignature() + " " +
  field.getName() + " = " + valueString);
```

This method deals only with arrays, which are treated differently from ordinary objects when retrieving their contents. Note that arrays don't have a field called "length".

Example 5.29. Method for printing out array contents

```
public void walkArray(JavaObject object) {
 System.out.println("JavaObject @ " + pointerToHexString(object.getID()));
```

All instances of `JavaObject` have a `JavaClass` with a name. For arrays, this follows the JNI conventions. An integer array would be called `"[I"`, whereas an array of strings would be called `"[Ljava/lang/String;"`.

Example 5.30. All objects have classes

```
 JavaClass clazz;
 clazz = object.getJavaClass();

 className = clazz.getName();
```

Each array describes the number of elements it contains. It is important to call `getArraySize()` and not `getSize()` as the latter returns the size of the object on the heap.

Example 5.31. Get number of array elements

```
 int arraySize = 0;
 arraySize = object.getArraySize();
```

An array's class should be able to report the type of its elements. This call is used to determine the type of array to receive the contents of the array.

Example 5.32. Getting the type of the array elements

```
 String componentName;
 componentName = clazz.getComponentType().getName();
```

Arrays elements are not accessed on an individual basis. Instead, their contents are copied to real arrays. This code demonstrates that there are `JavaClass` for primitive types, in the same way there is in reflection. These names are used to create primitive arrays of the correct type. Java[tm] reflection functions in the same way.

Example 5.33. Creating array of correct type

```
Object arrayCopy;

if ("boolean".equals(componentName)) {
 arrayCopy = new boolean[arraySize];
} else if ("byte".equals(componentName)) {
 arrayCopy = new byte[arraySize];
} else if ("char".equals(componentName)) {
 arrayCopy = new char[arraySize];
} else if ("short".equals(componentName)) {
 arrayCopy = new short[arraySize];
} else if ("int".equals(componentName)) {
 arrayCopy = new int[arraySize];
} else if ("long".equals(componentName)) {
 arrayCopy = new long[arraySize];
} else if ("float".equals(componentName)) {
 arrayCopy = new float[arraySize];
} else if ("double".equals(componentName)) {
 arrayCopy = new double[arraySize];
} else {
```

If an array is not an array of primitives, it must be an array of objects. As there is no means of converting a `JavaObject` into a "real" object, an array of `JavaObject` instance is returned. Multidimensional arrays are returned as arrays of `JavaObject` instances that are themselves arrays. The example code shows how a array to receive object arrays is allocated.

Example 5.34. Array of JavaObjects as destination

```
arrayCopy = new JavaObject[arraySize];
```

The method `JavaObject.arraycopy` is used to copy array elements in the same way as `java.lang.System.arraycopy()`. Implementors and those writing applications using this method call should take care as arrays can be extremely large, potentially larger than the JVM's heap size. If a fraction of an array is asked for, that is all that should be allowed in memory.

Example 5.35. Copying array contents

```
 object.arraycopy(0, arrayCopy, 0, arraySize);
```

This code prints out the contents of the array elements. The `java.lang.Array.get()` method is used to retrieve elements from the array in a generic fashion. If `null` is retrieved, that is printed, otherwise if it is an object the type and address of the object is printed and failing that it must be an autoboxed primitive that can be printed out using its `toString()`. The `CorruptDataException` is caught within the loop to allow the printing to continue even if some of the elements can't be located.

Example 5.36. Printing out array elements

```
 System.out.println("\t" + className + "[" + arraySize +"] = {");
 for (int cnt=0; cnt < arraySize; cnt++) {
  Object obj = Array.get(arrayCopy, cnt);

  if (obj == null) {
   System.out.println("\t\tnull,");
  } else if (obj instanceof JavaObject) {
   JavaObject refObj = (JavaObject) obj;
   try {
    System.out.println("\t\t" + refObj.getJavaClass().getName()+ " @ "+
      pointerToHexString(refObj.getID())+",");
   } catch (CorruptDataException e) {
    System.err.println("\t\t CorruptDataException while printing out array element");
    e.printStackTrace();
   }
  } else {
   System.out.println("\t\t"+obj+",");
  }
 }
 System.out.println("\t};");
}
```

Chapter 6. API Reference

This chapter contains the details of the Java classes that comprise the API.

package javax.tools.diagnostics

Top level of the API, provides access to API implementations.

Common semantics within javax.tools.diagnostics
Collections

1. Unless specifically declared all Collection classes and their associated Iterators are read only. Attempts to add ,remove or replace items within the collection will result in a java.lang.UnsupportedOperationException being thrown

2. Ordered Collections returned by this API are required to have a consistent and repeatable ordering across calls.

Object equality

Instances of classes defined by this API should only be tested for equality by using the java.lang.Object#equals() method.

Table 6.1. Class Summary

Name	Summary
FactoryRegistry	Experimental addition to the API.

Details

class FactoryRegistry

public FactoryRegistry extends java.lang.Object

Experimental addition to the API.

This class provides a central registry for image factories

Image factories can be registered directly using the addFactory() method.

The default registry obtained by calling getDefaultRegistry() uses javax.imageio.spi.ServiceRegistry to discover ImageFactory implementations.

To register an ImageFactory implementation that can be discovered by the registry do the following :

- Create a 'services' directory as a child of the manfest directory 'META-INF'
- Within this 'services' directory create a text file called 'javax.tools.diagnostics.image.ImageFactory'
- This text file should contain a single line which is the package qualified name of the ImageFactory implementation to be registered

Table 6.2. FactoryRegistry Constructor Summary

Constructor
public FactoryRegistry() Creates an empty registry. To obtain an populated registry use the getDefaultRegistry() method

Table 6.3. FactoryRegistry Methods

Methods	
getDefaultRegistry	*public static synchronized FactoryRegistry getDefaultRegistry()* Returns the default registry. This registy is preloaded with ImageFactory implementations discovered using the javax.imageio.spi.ServiceRegistry **Returns** default image factory
iterator	*public Iterator iterator()* Returns an java.util.Iterator of ImageFactories registered to this registry.

Methods	
addFactory	*public boolean addFactory(ImageFactory factory)* Adds an ImageFactory to the registry instance If the factory instance is already in the registry it is not added again. **Returns** true if factory added **Parameters** factory factory to add to registry **Throws** IllegalArgument if factory is null
getFactories	*public ImageFactory getFactories()* Always returns an array even if the registry is empty **Returns** Returns the factories in the registry as an array
getImage	*public Image getImage(File file)* Returns an appropriate javax.tools.diagnostics.image.Image for the provide file by locating the first registered image factory that can handle the case insensitive file name extension of the provided file. If no factory can be found to handle the file then null is returned. File names without extension will always return null **Returns** Image or null **Parameters** file to create image from **Throws** IllegalArgumentException if file is null IOException if errors occur during image creation

Methods	
getJavaRuntime	*public JavaRuntime getJavaRuntime(File file)* Returns an appropriate javax.tools.diagnostics.image.Image for the provide file by locating the first registered image factory that can handle the case insensitive file name extension of the provided file. If no factory can be found to handle the file then null is returned. File names without extension will always return null **Returns** Image or null **Parameters** file to create image from **Throws** IllegalArgumentException if file is null IOException if errors occur during image creation

package javax.tools.diagnostics.image

Package of interfaces representing a snapshot of a program.

In order to accommodate most dump formats, the API allows the possibility of having multiple processes in the same dump. Each process is capable of having multiple ManagedRuntime. The package contains information on:

- The Image of the snapshot.
- The ImageThread threads including information on their ImageRegister registers and ImageStackFrame stacks.
- Loaded ImageModule libraries and their ImageSymbol symbols.
- Information on the running ImageProcess process.

Table 6.4. Interface Summary

Name	Summary
ImageProcess	This class represents a Process running in a given Address Space.
Image	This class represents an entire operating system image (e.g.
ImageAddressSpace	This class represents a single Address Space within the image.
ImageSection	Represents a range of memory used for a specific purpose.
ImageStackFrame	Represents a native stack frame
ImageFactory	This interface is used for classes which can produce instances of Image implementors.

Name	Summary
ImageThread	A low-level thread instance
ImagePointer	Represents an address in image memory.
CorruptData	This class is used to indicate that corruption has been detected in the image.
ImageModule	Represents a shared library loaded into the image, or the executable module itself
ImageSymbol	Represents a symbol defined in an ImageModule
ImageRegister	Represents the state of a CPU or FPU register

Table 6.5. Class Summary

Name	Summary
MemoryAccessException	Indicates that an attempt was made to access memory which is not included within the image
DiagnosticException	This class is the superclass of all exceptions thrown by javax.tools.diagnostics classes
CorruptDataException	Used to indicate that corruption has been detected in the image.
DataUnavailable	This exception is used to indicate that data was requested which is not available on this system, or in this image.

Details

interface ImageProcess

public interface ImageProcess

This class represents a Process running in a given Address Space.

Table 6.6. ImageProcess Methods

Methods	
getCommandLine	*public String getCommandLine()* Fetch the command line for this process. This may be the exact command line the user issued, or it may be a reconstructed command line based on argv[] and argc. **Returns** the command line for the process **Throws** DataUnavailable if the information cannot be provided CorruptDataException

Methods	
getEnvironment	*public Properties getEnvironment()* Get the environment variables for this process. **Returns** the environment variables for this process **Throws** DataUnavailable if the information cannot be provided CorruptDataException
getID	*public String getID()* Get the system-wide identifier for the process. **Returns** a system-wide identifier for the process (e.g. a process id (pid) on Unix like systems) **Throws** DataUnavailable if the information cannot be provided CorruptDataException
getLibraries	*public List getLibraries()* Get the set of shared libraries which are loaded in this process. **Returns** an iterator to iterate over the shared libraries which are loaded in this process **Throws** DataUnavailable if the information cannot be provided CorruptDataException

Methods	
getExecutable	*public ImageModule getExecutable()* Get the module representing the executable within the image. **Returns** the module representing the executable within the image (as opposed to modules representing libraries) **Throws** DataUnavailable if the information cannot be provided CorruptDataException
getThreads	*public List getThreads()* Get the set of image threads in the image. There is not necessarily any relationship between JavaThreads and ImageThreads. A JVM implementation may use an n:m mapping of JavaThreads to ImageThreads, and not all ImageThreads are necessarily attached. **Returns** an iterator to iterate over each ImageThread in the image
getCurrentThread	*public ImageThread getCurrentThread()* Find the thread which triggered the creation of the image **Returns** the ImageThread which caused the image to be created, or null if the image was not created due to a specific thread **Throws** CorruptDataException
getRuntimes	*public List getRuntimes()* Get the set of the known ManagedRuntime environments in the image. In a typical image, there will be only one runtime, and it will be an instance of JavaRuntime. However any user of this API should be aware that there is a possibility that other runtimes may exist in the image **Returns** an iterator to iterate over all of the known ManagedRuntime environments in the image.

Methods	
getSignalNumber	*public int getSignalNumber()* Get the OS signal number in this process which triggered the creation of this image. **Returns** the OS signal number in this process which triggered the creation of this image, or 0 if the image was not created because of a signal in this process **Throws** DataUnavailable if the information cannot be provided CorruptDataException
getSignalName	*public String getSignalName()* Get the name of the OS signal in this process which triggered the creation of this image. **Returns** the name of the OS signal in this process which triggered the creation of this image, or null if the image was not created because of a signal in this process **Throws** DataUnavailable if the information cannot be provided CorruptDataException
getPointerSize	*public int getPointerSize()* Determine the pointer size used by this process. Currently supported values are 31, 32 or 64. In the future, other pointer sizes may also be supported. **Returns** the size of a pointer, in bits

interface Image

public interface Image

This class represents an entire operating system image (e.g. a core file). There are methods for accessing information about the architecture of the machine on which the image was running - hardware and operating system. The major feature, however, is the ability to iterate over the Address Spaces contained within the image.

Table 6.7. Image Methods

Methods	
getAddressSpaces	*public List getAddressSpaces()* Get the set of address spaces within the image - typically one but may be more on some systems such as z/OS. **Returns** an Iterator which iterates over all of the address spaces described by this Image
getProcessorType	*public String getProcessorType()* Get the family name for the processor on which the image was running. **Returns** the family name for the processor on which the image was running. This corresponds to the value you would find in the "os.arch" System property. **Throws** DataUnavailable — if this data cannot be inferred from this core type CorruptDataException — if expected data cannot be read from the core
getProcessorSubType	*public String getProcessorSubType()* Get the precise model of the CPU. **Returns** the precise model of the CPU (note that this can be an empty string but not null). e.g. getProcessorType() will return "x86" where getProcessorSubType() may return "Pentium IV step 4" Note that this value is platform and implementation dependent. **Throws** DataUnavailable CorruptDataException

Methods	
getProcessorCount	*public int getProcessorCount()* Get the number of CPUs running in the system on which the image was running. **Returns** the number of CPUs running in the system on which the image was running **Throws** DataUnavailable — if the information cannot be provided
getSystemType	*public String getSystemType()* Get the family name for the operating system. **Returns** the family name for the operating system. This should be the same value that would be returned for the "os.name" system property **Throws** DataUnavailable — if this data cannot be inferred from this core type CorruptDataException — if expected data cannot be read from the core
getSystemSubType	*public String getSystemSubType()* Get the detailed name of the operating system. **Returns** the detailed name of the operating system, or an empty string if this information is not available (null will never be returned). This should be the same value that would be returned for the "os.version" system property **Throws** DataUnavailable CorruptDataException

Methods	
getInstalledMemory	*public long getInstalledMemory()* Get the amount of physical memory (in bytes) installed in the system on which the image was running. **Returns** the amount of physical memory installed in the system on which the image was running. The return value is specified in bytes. **Throws** DataUnavailable — if the information cannot be provided
getCreationTime	*public long getCreationTime()* Determines when the image was created **Returns** the time in milliseconds since 1970 **Throws** DataUnavailable
getHostName	*public String getHostName()* Get the host name of the system where the image was running. **Returns** The host name of the system where the image was running. This string will be non-null and non-empty **Throws** DataUnavailable — If the image did not provide this information (would happen if the system did not know its host name or if the image predated this feature). CorruptDataException

Methods	
getIPAddresses	*public Iterator getIPAddresses()* The set of IP addresses (as InetAddresses) which the system running the image possessed. **Returns** An Iterator over the IP addresses (as InetAddresses) which the system running the image possessed. The iterator will be non-null (but can be empty if the host is known to have no IP addresses). **Throws** DataUnavailable — If the image did not provide this information (would happen if the system failed to look them up or if the image pre-dated this feature).
getSource	*public File getSource()* Experimenal

interface ImageAddressSpace

public interface ImageAddressSpace

This class represents a single Address Space within the image. On some operating systems (e.g. z/OS), there can be more than one Address Space per core file (but generally with only one process per ImageAddressSpace).

Table 6.8. ImageAddressSpace Methods

Methods	
getCurrentProcess	*public ImageProcess getCurrentProcess()* Get the process within this address space which caused the image to be created. **Returns** the process within this address space which caused the image to be created, if any. Return null if no individual process triggered the creation of the image.
getProcesses	*public List getProcesses()* Get the set of processes within the address space. **Returns** an iterator which provides all of the processes within a given address space. In most images, there will only be one process within an ImageAddressSpace

Methods	
getPointer	*public ImagePointer getPointer(long address)* A factory method for creating pointers into this address space **Returns** an ImagePointer for the specified address **Parameters** address the address to point to
getImageSections	*public List getImageSections()* Get the raw memory in the address space. **Returns** An iterator of all the ImageSections in the address space. Their union will be the total process address space

class MemoryAccessException

public MemoryAccessException extends javax.tools.diagnostics.image.DiagnosticException

Indicates that an attempt was made to access memory which is not included within the image

Table 6.9. MemoryAccessException Constructor Summary

Constructor
public MemoryAccessException(ImagePointer badPointerString description) Build exception for the given location and description
public MemoryAccessException(ImagePointer badPointer) Build exception for the given location and description

Table 6.10. MemoryAccessException Methods

Methods	
getPointer	*public ImagePointer getPointer()* Get a pointer into the image where the access failed. **Returns** The pointer into the image where the access failed

class DiagnosticException

public DiagnosticException extends java.lang.Exception

This class is the superclass of all exceptions thrown by javax.tools.diagnostics classes

Table 6.11. DiagnosticException Constructor Summary

Constructor
public DiagnosticException(String description) Build exception with the given description
public DiagnosticException() Build exception with no description

interface ImageSection

public interface ImageSection

Represents a range of memory used for a specific purpose.

Table 6.12. ImageSection Methods

Methods	
getBaseAddress	*public ImagePointer getBaseAddress()* Get the lowest address of memory in this section. **Returns** the lowest address of memory in this section
getSize	*public long getSize()* Get the size of this contiguous image section as measured in bytes. **Returns** the size of this contiguous image section as measured in bytes
getName	*public String getName()* Get the name of this section (e.g. ".text"). **Returns** the name of this section (e.g. ".text"). Note that sections of the image which have no specific name will receive a name synthesised by the implementation. This will never be null.

Methods	
isExecutable	*public boolean isExecutable()* Does this section have permission to allow the processor to attempt to execute code? **Returns** true if this section is executable, false otherwise **Throws** DataUnavailable
isReadOnly	*public boolean isReadOnly()* Is this section read-only ? **Returns** true if write access to this section was disabled **Throws** DataUnavailable
isShared	*public boolean isShared()* Is this section shared with other processes.. **Returns** true if this section is shared between processes **Throws** DataUnavailable

interface ImageStackFrame

public interface ImageStackFrame

Represents a native stack frame

Table 6.13. ImageStackFrame Methods

Methods	
getProcedureAddress	*public ImagePointer getProcedureAddress()* Get the address of the current instruction within the procedure being executed. **Returns** the address of the current instruction within the procedure being executed, or null if not available. Use this address with caution, as it is provided only as a best guess. It may not be correct, or even within readable memory **Throws** CorruptDataException
getBasePointer	*public ImagePointer getBasePointer()* Get the base pointer of the stack frame. **Returns** the base pointer of the stack frame **Throws** CorruptDataException

Methods	
getProcedureName	*public String getProcedureName()* Returns a string describing the procedure at this stack frame. Implementations should use the following template so that procedure names are reported consistently: • `libname(sourcefile)::entrypoint±<node></node>offset` Any portion of the template may be omitted if it is not available • e.g. • `system32(source.c)::WaitForSingleObject+14` • `system32::WaitForSingleObject-4` • `(source.c)::WaitForSingleObject` • `::WaitForSingleObject+14` • `system32+1404` • `system32::TWindow::open(int,void*)+14` **Returns** a string naming the function executing in this stack frame. If the name is not known for legitimate reasons, a synthetic name will be returned.. **Throws** CorruptDataException

interface ImageFactory

public interface ImageFactory

This interface is used for classes which can produce instances of Image implementors.

Note that this interface forms the contract between the javax.tools.diagnostics.FactoryRegistry and an implementation. The methods on this interface are only intended to be called by the javax.tools.diagnostics.FactoryRegistry

Table 6.14. ImageFactory Methods

Methods	
getImage	*public Image getImage(File imageFile)* Creates a new Image object based on the contents of imageFile. Note that this method is only intended to be called by the javax.tools.diagnostics.FactoryRegistry **Returns** an instance of Image **Parameters** imageFile — a file with Image information, typically a core file **Throws** IOException — if unable to create an image from the provided file
getImage	*public Image getImage(File imageFileFile metadata)* Creates a new Image object based on the contents of imageFile and metadata Note that this method is only intended to be called by the javax.tools.diagnostics.FactoryRegistry **Returns** an instance of Image **Parameters** imageFile — a file with Image information, typically a core file metadata — a file with additional Image information. This is an implementation defined file **Throws** IOException — if unable to create an image from the provided file

Methods	
getMajorVersion	*public int getMajorVersion()* Fetch the major version number **Returns** An integer corresponding to the API major version number
getMinorVersion	*public int getMinorVersion()* Fetch the minor version number **Returns** An integer corresponding to the API minor version number
getModificationLevel	*public int getModificationLevel()* Fetch the modification level **Returns** An integer corresponding to the API modification level
getValidFileExtensions	*public String getValidFileExtensions()* Returns an array of file extensions that the javax.tools.diagnostics.FactoryRegistry can use to determine if an file can be processed by this Image Factory implementation. File extensions are the part of a file name after the last '.'. The returned array is treated as a case insensitive collection of file extensions. The returned array is expected to contain at least one entry. Note that this method is only intended to be called by the javax.tools.diagnostics.FactoryRegistry **Returns** an array of case insensitive file extension names.

Methods	
getJavaRuntime	*public JavaRuntime getJavaRuntime(File file)* Creates a new JavaRuntime object based on the contents of the file; Note that this method is only intended to be called by the javax.tools.diagnostics.FactoryRegistry **Returns** an instance of JavaRuntime **Parameters** file — a file with JavaRuntime information **Throws** IOException — if unable to create a runtime from the provided file

class CorruptDataException

public CorruptDataException extends javax.tools.diagnostics.image.DiagnosticException

Used to indicate that corruption has been detected in the image.

Table 6.15. CorruptDataException Constructor Summary

Constructor
public CorruptDataException(CorruptData data) Construct a new CorruptDataException for the specified corrupt data

Table 6.16. CorruptDataException Methods

Methods	
getCorruptData	*public CorruptData getCorruptData()* Get more info about the corrupted data **Returns** the CorruptData object

interface ImageThread

public interface ImageThread

A low-level thread instance

Table 6.17. ImageThread Methods

Methods	
getID	*public String getID()* Fetch a unique identifier for the thread. In many operating systems, threads have more than one identifier (e.g. a thread id, a handle, a pointer to VM structures associated with the thread). In this case, one of these identifiers will be chosen as the canonical one. The other identifiers would be returned by getProperties() **Returns** a process-wide identifier for the thread (e.g. a tid number) **Throws** CorruptDataException
getStackFrames	*public List getStackFrames()* Get the set of stack frames on this thread. **Returns** an iterator to walk the native stack frames in order from top-of-stack (that is, the most recent frame) to bottom-of-stack. Throws DataUnavailable if native stack frames are not available on this platform. **Throws** DataUnavailable — If native stack frames are not available on this platform
getStackSections	*public List getStackSections()* Get the set of image sections which make up the stack. **Returns** a collection of ImageSections which make up the stack. On most platforms this consists of a single entry, but on some platforms the thread's stack may consist of non-contiguous sections

Methods	
getRegisters	*public List getRegisters()* Get the register contents. **Returns** an iterator to iterate over the state of the CPU registers when the image was created. The collection may be empty if the register state is not available for this thread. If the CPU supports partial registers (e.g. AH, AL, AX, EAX, RAX on AMD64), only the largest version of the register will be included
getProperties	*public Properties getProperties()* Get the OS-specific properties for this thread. **Returns** a table of OS-specific properties for this thread. Values which are commonly available include "priority" -- the priority of the thread "policy" -- the scheduling policy of the thread

class DataUnavailable

public DataUnavailable extends javax.tools.diagnostics.image.DiagnosticException

This exception is used to indicate that data was requested which is not available on this system, or in this image.

Table 6.18. DataUnavailable Constructor Summary

Constructor
public DataUnavailable(String description) Build exception with the given description
public DataUnavailable() Build exception with no description

interface ImagePointer

public interface ImagePointer

Represents an address in image memory.

Table 6.19. ImagePointer Methods

Methods	
getAddress	*public long getAddress()* Get the unwrapped address, represented as a 64-bit integer. **Returns** the unwrapped address, represented as a 64-bit integer Use caution when comparing addresses, as some addresses may be negative. Note that on segmented memory architectures, it may not be possible to represent all addresses accurately as integers
getAddressSpace	*public ImageAddressSpace getAddressSpace()* Get the address space to which this pointer belongs. **Returns** the address space to which this pointer belongs
add	*public ImagePointer add(long offset)* Build a new image pointer offset from this one by the given amount. **Returns** a new ImagePointer based at getAddress() + offset **Parameters** offset
isExecutable	*public boolean isExecutable()* Is the referenced location executable ? **Returns** true if this memory address is within an executable page **Throws** DataUnavailable

Methods	
isReadOnly	*public boolean isReadOnly()* Is the referenced location read only ? **Returns** true if write access to this memory address was disabled in the image **Throws** DataUnavailable
isShared	*public boolean isShared()* Is the referenced location shared ? **Returns** true if this memory address is shared between processes **Throws** DataUnavailable
getPointerAt	*public ImagePointer getPointerAt(long index)* Get the value at the given offset from this pointer. To determine the number of bytes to skip after this call to read the next value, use `ImageProcess.getPointerSize()` . **Returns** the 32 or 64-bit pointer stored at getAddress() + index in the same address space. **Parameters** index an offset (in bytes) from the current position **Throws** MemoryAccessException if the memory cannot be read CorruptDataException if the memory should be in the image, but is missing or corrupted

Methods	
getLongAt	*public long getLongAt(long index)* Get the value at the given offset from this pointer. **Returns** the 64-bit long stored at getAddress() + index **Parameters** index — an offset (in bytes) from the current position **Throws** MemoryAccessException — if the memory cannot be read CorruptDataException — if the memory should be in the image, but is missing or corrupted
getIntAt	*public int getIntAt(long index)* Get the value at the given offset from this pointer. **Returns** the 32-bit int stored at getAddress() + index **Parameters** index — an offset (in bytes) from the current position **Throws** MemoryAccessException — if the memory cannot be read CorruptDataException — if the memory should be in the image, but is missing or corrupted

Methods	
getShortAt	*public short getShortAt(long index)* Get the value at the given offset from this pointer. **Returns** the 16-bit short stored at getAddress() + index **Parameters** index — an offset (in bytes) from the current position **Throws** MemoryAccessException — if the memory cannot be read CorruptDataException — if the memory should be in the image, but is missing or corrupted
getByteAt	*public byte getByteAt(long index)* Get the value at the given offset from this pointer. **Returns** the 8-bit byte stored at getAddress() + index **Parameters** index — an offset (in bytes) from the current position **Throws** MemoryAccessException — if the memory cannot be read CorruptDataException — if the memory should be in the image, but is missing or corrupted

Methods	
getFloatAt	*public float getFloatAt(long index)* Get the value at the given offset from this pointer. **Returns** the 32-bit float stored at getAddress() + index **Parameters** index — an offset (in bytes) from the current position **Throws** MemoryAccessException — if the memory cannot be read CorruptDataException — if the memory should be in the image, but is missing or corrupted
getDoubleAt	*public double getDoubleAt(long index)* Get the value at the given offset from this pointer. **Returns** the 64-bit double stored at getAddress() + index **Parameters** index — an offset (in bytes) from the current position **Throws** MemoryAccessException — if the memory cannot be read CorruptDataException — if the memory should be in the image, but is missing or corrupted
equals	*public boolean equals(Object obj)* **Returns** True obj refers to the same Image Pointer in the image **Parameters** obj
hashCode	*public int hashCode()*

interface CorruptData

public interface CorruptData

This class is used to indicate that corruption has been detected in the image. It may indicate corruption of the image file, or it may indicate that inconsistencies have been detected within the image file, perhaps caused by a bug in the runtime or application. It may be encountered in two scenarios:

- within a CorruptDataException
- returned as an element from an Iterator

Any iterator in javax.diagnostics may implicitly include one or more CorruptData objects within the list of objects it provides. Normal data may be found after the CorruptData object if the javax.diagnostics implementation is able to recover from the corruption.

Table 6.20. CorruptData Methods

Methods	
toString	*public String toString()* Provides a string which describes the corruption **Returns** a descriptive string
getAddress	*public ImagePointer getAddress()* Return an address associated with the corruption. If the corruption is not associated with an address, return null. If the corruption is associated with more than one address, return the one which best identifies the corruption. **Returns** the address of the corrupted data

interface ImageModule

public interface ImageModule

Represents a shared library loaded into the image, or the executable module itself

Table 6.21. ImageModule Methods

Methods	
getName	*public String getName()* Get the file name of the shared library. **Returns** the file name of the shared library **Throws** CorruptDataException — If the module is corrupt and the original file cannot be determined
getSections	*public List getSections()* Get the collection of sections that make up this library. **Returns** a collection of sections that make up this library
getSymbols	*public List getSymbols()* Provides a collection of symbols defined by the library. This list is likely incomplete as many symbols may be private, symbols may have been stripped from the library, or symbols may not by available in the image. **Returns** a collection of symbols which are defined by this library.
getProperties	*public Properties getProperties()* Get the table of properties associated with this module. **Returns** a table of properties associated with this module. Values typically defined in this table include "version" -- version information about the module **Throws** CorruptDataException

interface ImageSymbol

public interface ImageSymbol

Represents a symbol defined in an ImageModule

Table 6.22. ImageSymbol Methods

Methods	
getAddress	*public ImagePointer getAddress()* Get the address of this symbol in the image. **Returns** the address of this symbol in the image
getName	*public String getName()* Get the name of the symbol. **Returns** the name of the symbol

interface ImageRegister

public interface ImageRegister

Represents the state of a CPU or FPU register

Table 6.23. ImageRegister Methods

Methods	
getName	*public String getName()* Fetch the name of a register. On some CPUs registers may have more than one conventional name. Recommended names for some CPUs are provided in the user guide. **Returns** the conventional name of the register
getValue	*public Number getValue()* Get the value for the register. **Returns** an integral or floating point type which contains the value for the register. The returned value may be an instance of any subclass of Number. For instance, on x86 architectures with MMX, the XMM registers will be returned as BigInteger instances **Throws** CorruptDataException

package javax.tools.diagnostics.runtime

Definition of an abstract view of a managed runtime.

Runtimes are collections of software services that together, provide a environment where an application program can be executed. Most computer languages provide some sort of runtime. This package contains definitions that are common to all runtime environments.

Table 6.24. Interface Summary

Name	Summary
ManagedRuntime	A generic managed runtime instance.

Details

interface ManagedRuntime

public interface ManagedRuntime

A generic managed runtime instance. A Managed Runtime as against an "Unmanaged Runtime" is one where the runtime takes an active role in the program execution. Common examples of managed runtimes are the Java Virtual Machine or the .NET Common Language Runtime.

No class should implement this interface directly. This is an marker interface which is extended by specific runtime interfaces. See javax.tools.diagnostics.runtime.java.JavaRuntime as an example of such a case.

Table 6.25. ManagedRuntime Methods

Methods	
getVersion	*public String getVersion()* Returns version data available for this runtime instance. The version information is never null. The format of the version data is implementation specific. **Returns** a string representing the available version information specific to the implementation **Throws** CorruptDataException — If the ManagedRuntime implementation is unable to retrieve version data

Methods	
getFullVersion	*public String getFullVersion()* **Returns** a string representation of the version information for this runtime instance **Throws** CorruptDataException

package javax.tools.diagnostics.runtime.java

Definition of an abstract view of a Java runtime.

Implementations of the API expose information about Java virtual machines with the

`JavaRuntime`

interface. The information about the following can be retrieved from JavaRuntimes:

- JavaHeap Heaps.
- JavaObject Objects.
- JavaClassLoader Classloaders.
- JavaClass Classes, including their JavaField fields and JavaMethod methods.
- JavaMonitor Monitors.
- JavaThread Threads, including their JavaStackFrame stacks.

Table 6.26. Interface Summary

Name	Summary
JavaStackFrame	Represents a Java stack frame.
JavaLocation	Represents a point of execution within a Java method
JavaHeap	Represents a single heap of managed objects.
JavaReference	Represents a Java reference.
JavaRuntime	Represents an instance of a Java Virtual Machine This interface defines attributes and features common across real implementation of the Java Virtual Machine.
JavaObject	Represents a Java object or array.
JavaClassLoader	Represents an internal ClassLoader structure within a Java Virtual Machine instance.
JavaVMInitArgs	This class models the JavaVMInitArgs C structure passed to JNI_CreateJavaVM to create this Java Virtual Machine Typically the options passed to the JVM are similar but necessarily identical to these used to invoke the Java Virtual Machine from a command line.

Name	Summary
JavaMember	Abstract interface which both JavaField and JavaMethod inherit from.
JavaMonitor	Represents the underlying monitor used by a Java Virtual Machine to manage locking and synchronization of a Java object.
JavaMethod	Represents a method or constructor in a class
JavaField	Represents a field declaration.
JavaVMOption	This class models the JavaVMOption C structures passed to the JNI invocation API entry point JNI_CreateJavaVM used to create a Java Virtual Machine.
JavaClass	Represents a Java class.
JavaVariable	Representation of a Java Variable
JavaThread	Represents a Java thread.

Details

interface JavaStackFrame

public interface JavaStackFrame

Represents a Java stack frame.

Table 6.27. JavaStackFrame Methods

Methods	
getBasePointer	*public ImagePointer getBasePointer()* Get a pointer to the base of this stack frame within memory. The layout of a `JavaStackFrame` is implementation specific. **Returns** the base pointer of the stack frame **Throws** CorruptDataException

Methods	
getLocation	*public JavaLocation getLocation()* Returns the JavaLocation that represents the location of this `JavaStackFrame` within the Java program. **Returns** a location object describing where the frame is executing. **Throws** CorruptDataException
getHeapRoots	*public List getHeapRoots()* A list of references to objects and classes from this stack frame. Returns the references to object and classes this Java Virtual Machine's implementation considers as being kept alive by this Java stack frame. As well as references from local variables and operations stack entries, this may also include a reference to the Java frame's class or to an object this stack frame is keeping alive through holding its monitor. **Returns** a list of JavaReferences
getVariable	*public Object getVariable(int slot)* Gets the value of a variable from a stack frame. Returns a JavaObject for an object reference, `null` for a null object reference. Primitives are returned as boxed primitives. CorruptDataException is thrown if object reference is incorrect, or if the float or double are set to invalid values. **Parameters** slot - the numerical local variable slot number to retrieve. **Throws** DataUnavailable if this method is not supported or if stack not in correct state to return variables. IndexOutOfBoundsException if an invalid slot number is passed.

Methods	
getVariables	*public List getVariables()* Gets all variables from the stack frame. Returns a java.util.List containing the available JavaVariable JavaVariables. While the method JavaMethod#getVariables() will return all local variables for a method, this method will return only those variables that are visible at the point of execution for this stack frame.
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same JavaStackFrame in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaLocation

public interface JavaLocation

Represents a point of execution within a Java method

Table 6.28. JavaLocation Methods

Methods	
getAddress	*public ImagePointer getAddress()* Fetches the absolute address of the code which this location represents. This pointer will be contained within one of the segments returned by getBytecodeSections() or getCompiledSections() of the method returned by getMethod(). null may be returned, particularly for methods with no bytecode or compiled sections (e.g. some native methods) Although an offset into the method may be calculated using this pointer, caution should be exercised in attempting to map this offset to an offset within the original class file. Various transformations may have been applied to the bytecodes by the VM or other agents which may make the offset difficult to interpret. For native methods, the address may be meaningless. **Returns** the address in memory of the managed code **Throws** CorruptDataException — if the underlying data is in an unexpected state
getLineNumber	*public int getLineNumber()* Get the line number. **Returns** the line number, if available, or throws DataUnavailable if it is not available Line numbers are counted from 1 **Throws** DataUnavailable — if the line number data is not available for this location CorruptDataException — if the underlying data is in an unexpected state

Methods	
getFilename	*public String getFilename()* Get the source file name. **Returns** the name of the source file, if available, or throws DataUnavailable if it is not available **Throws** DataUnavailable — if the source file name is unavailable in the core CorruptDataException — if the underlying data is in an unexpected state
getCompilationLevel	*public int getCompilationLevel()* Get the compilation level for this location. This is an implementation defined number indicating the level at which the current location was compiled. 0 indicates interpreted. Any positive number indicates some level of JIT compilation. Typically, higher numbers indicate more aggressive compilation strategies For native methods, a non-zero compilation level indicates that some level of JIT compilation has been applied to the native call (e.g. a custom native call stub). To determine if the method is native, use getMethod().getModifiers(). **Returns** the compilation level **Throws** CorruptDataException — if the underlying data is in an unexpected state
getMethod	*public JavaMethod getMethod()* Get the method which contains the point of execution. **Returns** the method which contains the point of execution **Throws** CorruptDataException — if the underlying data is in an unexpected state

Methods	
toString	*public String toString()* **Returns** A string representing the location as it would be seen in a Java stack trace
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same Java Location in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaHeap

public interface JavaHeap

Represents a single heap of managed objects. The heap can be viewed as an unordered collection of JavaObjects or as a region of storage within the Java Virtual Machine instance. The heap commonly contains JavaObject instances that are reachable by navigating chains of JavaReference These references can be obtained from the JavaRuntime#getHeapRoots() method. A heap can contain instances which cannot be reached by the use of JavaReference

Table 6.29. JavaHeap Methods

Methods	
getSections	*public List getSections()* Get the set of memory regions that represent the memory layout of the heap. The actual make up of this list is implementation specific. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned value is never null but can be an empty list. **Returns** a list of ImageSection instances
getName	*public String getName()* Get a brief textual description of this heap. The value returned is implementation specific. The returned value is never null. **Returns** a brief textual description of this heap

Methods	
getObjects	*public List getObjects()* Get the set of objects which are stored in this heap. **Returns** a list of JavaObject objects which are stored in this heap The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned value is never null but can be an empty list.
equals	*public boolean equals(Object obj)* **Returns** true if the given object refers to the same Java Heap in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaReference

public interface JavaReference

Represents a Java reference.

A Java reference is a traceable relationship between two objects or between a root and a Java object.

References are used by Garbage Collection systems to identify objects that can be reclaimed.

Table 6.30. JavaReference Methods

Methods	
getRootType	*public int getRootType()* Get the root type, as defined in the JVMTI specification. **Returns** an integer representing the root type, see HEAP_ROOT_ statics above.
getReferenceType	*public int getReferenceType()* Get the reference type, as defined in the JVMTI specification. **Returns** an integer representing the reference type, see REFERENCE_ statics above.

Methods	
getReachability	*public int getReachability()* Get the reachability of the target object via this specific reference. **Returns** an integer representing the reachability, see REACHABILITY_ statics above.
getDescription	*public String getDescription()* Get a string describing the reference type. Users should not depend on the contents or identity of this string. e.g. "JNI Weak global reference", "Instance field 'MyClass.value'", "Constant pool string constant" **Returns** a String describing the reference type
isObjectReference	*public boolean isObjectReference()* Check to see if this reference points to an object in the heap **Returns** true if the target of this root is an object **Throws** DataUnavailable — if the requested information is not available CorruptDataException — is the underlying data is in an unexpected state
isClassReference	*public boolean isClassReference()* Check to see if this reference points to a class. **Returns** true if the target of this root is a class **Throws** DataUnavailable — if the requested information is not available CorruptDataException — is the underlying data is in an unexpected state

Methods	
getTarget	*public Object getTarget()* Get the object referred to by this reference. **Returns** a JavaObject or a JavaClass **Throws** DataUnavailable — if the requested information is not available CorruptDataException — is the underlying data is in an unexpected state
getSource	*public Object getSource()* Get the source of this reference if available. **Returns** a JavaClass, JavaObject, JavaStackFrame, JavaThread or null if unknown **Throws** DataUnavailable — if the requested information is not available CorruptDataException — is the underlying data is in an unexpected state

interface JavaRuntime

public interface JavaRuntime

Represents an instance of a Java Virtual Machine This interface defines attributes and features common across real implementation of the Java Virtual Machine. Not all of these characteristics are defined by the Java Virtual Machine Specification Notable additions beyond the JVM specification include Garbage Collection and access to the contents of the Heap or Heaps. Since this interface defines a view of the Java Runtime that is beyond that seen by the Java programmer during program execution it is necessarily more detailed.

Table 6.31. JavaRuntime Methods

Methods	
getJavaVM	*public ImagePointer getJavaVM()* Get the object that represents the virtual machine **Returns** the address of the JavaVM structure which represents this JVM instance in JNI **Throws** CorruptDataException
getJavaVMInitArgs	*public JavaVMInitArgs getJavaVMInitArgs()* Fetch the JavaVMInitArgs which were used to create this VM. See JNI_CreateJavaVM in the JNI Specification for more details. A valid object is returned or an exception is thrown. **Returns** the JavaVMInitArgs which were used to create this VM. **Throws** DataUnavailable — if the arguments are not available CorruptDataException — if the implementation was unexpectedly unable to retrieve the data

Methods	
getJavaClassLoaders	*public List getJavaClassLoaders()* Get the set of class loaders available in this Java Virtual Machine instance. Available in this context means class loaders that are participating in the class loader hierarchy. All class loaders are returned including any defined by the Java Virtual machine instance itself. Any structural relationships between class loaders in this list is not exposed. Recreation of the class loader graph within a Java Virtual Machine instance is beyond the scope of the API. At least one class loader must be returned in the resulting list. **Returns** a java.util.List of all of the class loaders within this Java Virtual Machine instance
getThreads	*public List getThreads()* Get the set of Java Threads that have been started java.lang.Thread#start() in this Java Virtual Machine instance. This method does not return all instances of java.lang.Thread contained within the system. Only threads that have been started and have not yet stopped or exited are returned. Threads may not be in an active state when returned by this method. The returned list follows the standard semantics for javax.tools.diagnostics collections The returned list is never null although it can be empty. **Returns** a java.util.List of the JavaThreads in the runtime
getCompiledMethods	*public List getCompiledMethods()* Get the set of JavaMethod objects that have been compiled. Compiled methods are methods that have been converted into native code by the Java Virtual Machine or related Just In Time Compiler There is no expectation that any method has been compiled. The returned list could be empty. However any JavaMethod reachable though the API which would return a non empty list for calls to JavaMethod#getCompiledSections() must be contained within the list returned by this method. The returned list follows the standard semantics for javax.tools.diagnostics collections **Returns** a java.util.List of all of the JavaMethods in the JavaRuntime which have been compiled.

Methods	
getMonitors	*public List getMonitors()* Provides access to the collection of monitors used in the Java Virtual Machine. This collection can include monitors associated with managed objects (e.g. object monitors) and monitors associated with Java Virtual Machine internal control structures (e.g. raw monitors). Raw monitors are implementation specific. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of monitors
getHeaps	*public List getHeaps()* Get the set of heaps known by the Java Virtual Machine There may be multiple heaps within a Java Virtual Machine, for instance a generational heap and a class heap. Heaps may be specific to this Java Virtual Machine instance, or may be shared between multiple Java Virtual Machine instances. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null and will always contain at least one JavaHeap object **Returns** a list for all of the Java heaps within this runtime.
getHeapRoots	*public List getHeapRoots()* Get the complete set of object and class roots known to the Java Virtual Machine **Returns** a list of JavaReferences representing the known global heap roots within this runtime. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but can be empty.

Methods	
getTraceBuffer	*public Object getTraceBuffer(String bufferNameboolean formatted)* **Returns** an implementation specific result, depending on the parameters **Parameters** bufferName — a String naming the buffer to be fetched formatted — true if formatting should be performed on the buffer, or false if the raw buffer contents should be returned **Throws** CorruptDataException
getObjectAtAddress	*public JavaObject getObjectAtAddress(ImagePointer address)* Gets the object located at address `address` in the heap. **Returns** the JavaObject instance representing the located object. **Parameters** addr — the ImagePointer instance representing the start address of object in the heap; **Throws** IllegalArgumentException — if address is null, outside the heap's boundaries or if it doesn't point to the start location of an object; MemoryAccessException — if address is is in the heap but it's not accessible from the dump; CorruptDataException — if any data needed to build the returned instance of JavaObject is corrupt. DataUnavailable — if any data needed to build the returned instance of JavaObject is not available.

Methods	
equals	*public boolean equals(Object obj)* **Returns** true if the given object refers to the same Java Runtime in the image **Parameters** obj
hashCode	*public int hashCode()*
getSource	*public File getSource()* Returns the File used as source for the creation of this Runtime. This File will be equal to the file presented to the FactoryRegistry when this runtime (or its parent Image) was created. **Returns** File object

interface JavaObject

public interface JavaObject

Represents a Java object or array.

Array elements can be retrieved using the arraycopy() method. Object instance fields can be retrieved using the

`get*()`

methods in JavaField such as JavaField#get(JavaObject). The JavaField objects can be retrieved from an object's JavaClass using JavaClass#getDeclaredFields().

Table 6.32. JavaObject Methods

Methods	
getJavaClass	*public JavaClass getJavaClass()* Get the JavaClass instance which represents the class of this object. This method never returns null, all objects have a class. The JavaClass returned might be synthetic for array types. **Returns** the JavaClass instance which represents the class of this object. **Throws** CorruptDataException

Methods	
isArray	*public boolean isArray()* Returns true if this JavaObject represents an array. **Returns** true if this JavaObject represents an array. **Throws** CorruptDataException

getArraySize	*public int getArraySize()*
	
	Get the number of elements in this array.
	
	This is equivalent to calling
	`array.length`
	in Java, where
	`array`
	is an array reference.
	
	Returns
	the number of elements in this array.
	
	Throws
	CorruptDataException
	IllegalArgumentException if the object is not an array.

Methods	
arraycopy	*public void arraycopy(int srcStartObject dstint dstStartint length)* Copies data from the array this JavaObject represents into an array. The dst object must be an array of the appropriate type -- a primitive type array for base types, or a JavaObject array for reference arrays. **Parameters** srcStart — index in the receiver to start copying from. dst — the destination array. dstStart — index in the destination array to start copying into. length — the number of elements to be copied. **Throws** CorruptDataException MemoryAccessException NullPointerException — if dst is null. IllegalArgumentException — if the object is not an array, or if dst is not an array of the appropriate type. IndexOutOfBoundsException — if srcStart, dstStart, or length are out of bounds in either the JavaObject or the destination array.
getSize	*public long getSize()* Get the number of bytes of memory occupied by this object. **Returns** the number of bytes of memory occupied by this object. The memory may not necessarily be contiguous. **Throws** CorruptDataException

Methods	
getHashcode	*public long getHashcode()* Fetch the basic hash code for the object. This is the hash code which would be returned if a Java thread had requested it. Typically the hash code is based on the address of an object, and may change if the object is moved by a garbage collect cycle. **Returns** the basic hash code of the object in the image. **Throws** DataUnavailable — if the hash code cannot be determined. CorruptDataException
getPersistentHashcode	*public long getPersistentHashcode()* Fetch the basic hash code of the object in the image. This hash code is guaranteed to be persistent between multiple snapshots of the same Image. If the hash code cannot be determined, or if the hash code for this object could change between snapshots, an exception is thrown. *If the VM uses a 'hasBeenHashed' bit, the value of this bit can be inferred by calling getPersistentHashcode(). If the persistent hash code is not available, then the 'hasBeenHashed' bit has not been set, and the hash of the object could change if the object moves between snapshots* **Returns** the basic hash code of the object in the image **Throws** DataUnavailable — if a hash code cannot be determined, or if the hash code could change between successive snapshots CorruptDataException

Methods	
getID	*public ImagePointer getID()* The ID of an object is a unique address is memory which identifies the object. It is probable that an object's address will change during the lifetime of a Java Virtual Machine because of the operations of the garbage collector. Other mechanisms for uniquely identifying objects should be used when comparing dumps. The data at this memory is implementation defined. The object may be non-contiguous. Portions of the object may appear below or above this address. **Returns** the runtime-wide unique identifier for the object.
getSections	*public List getSections()* Returns the sections that this object occupies in memory. These sections include the object's header and the data in the object. In certain allocation strategies, an object's header and data may be allocated contiguously. In this case, this method may return an iterator for a single section. In other schemes, the header may be separate from the data or the data may be broken up into multiple regions. Additionally, this function does not guarantee that the memory used by this object is not also shared by one or more other objects. The contents of the image sections are implementation specific, as so are undefined here. **Returns** a collection of sections that make up this object.
getReferences	*public List getReferences()* Get the set of references from this object. These references will include at least the object's references to its class, and any references from instance fields to other objects and classes, or array elements references to other objects. **Returns** an List of JavaReferences.

Methods	
getHeap	*public JavaHeap getHeap()* Gets the heap where this object is located. A JavaHeap will always be returned if this object could be retrieved by JavaHeap#getObject(), otherwise DataUnavailable is thrown. **Returns** the JavaHeap instance representing the heap where this object is stored in memory. **Throws** CorruptDataException — if the heap information for this object is corrupt. DataUnavailable — if the heap information for this object is not available.
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same Java Object in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaClassLoader

public interface JavaClassLoader

Represents an internal ClassLoader structure within a Java Virtual Machine instance. For most ClassLoaders there is a corresponding java.lang.ClassLoader instance within with JavaRuntime. For primordial class loaders such as the bootstrap class loader, there may or may not be a corresponding java.lang.ClassLoader instance.

Since Java does not define any strict inheritance structure between class loaders, there are no APIs for inspecting 'child' or 'parent' class loaders. This information may be inferred by inspecting the corresponding java.lang.ClassLoader instance:

pseudo javacode example

```
JavaClassLoader  loader;
JavaObject instance=loader.getObject();
String classLoaderName=instance.getJavaClass().getName();
```

Table 6.33. JavaClassLoader Methods

Methods	
getDefinedClasses	*public List getDefinedClasses()* Get the set of classes which are defined in this JavaClassLoader. Calling the JavaClass#getClassLoader() method on objects returned in this list will return this JavaClassLoader The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned value is never null but can be an empty list. **Returns** an list of classes which are defined in this JavaClassLoader
getCachedClasses	*public List getCachedClasses()* When a ClassLoader successfully delegates a findClass() request to another ClassLoader, the result of the delegation must be cached within the internal structure so that the Java Virtual Machine does not make repeated requests for the same class. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned value is never null but can be an empty list. **Returns** a list of classes which are defined in this JavaClassLoader or which were found by delegation to other JavaClassLoaders

Methods	
findClass	*public JavaClass findClass(String name)* Find a class by name within this class loader. The class may have been defined in this class loader, or this class loader may have delegated the load to another class loader and cached the result. The form of the name presented to this method should be as follows `[ packagenamepart / ... ] (classname ) [ $innerclassname ...]` *Examples* • To find the JavaClass that represents "java.lang.String" use findClass("java/lang/String") • To find the JavaClass that represents "Foo.InnerClass.InnerInnerClass" in the default package use findClass("Foo$InnerClass$InnerInnerClass") • To find the JavaClass that represents "java.util.Map.Entry use findClass("java/util/Map$Entry") **Returns** the JavaClass instance, or null if it is not found **Parameters** name — of the class to find. Packages should be separated by '/' instead of '.' **Throws** CorruptDataException — if the underlying data is in an unexpected state
getObject	*public JavaObject getObject()* Get the java.lang.ClassLoader instance (represented by a JavaObject associated with this class loader. If there is no associated class loader, for example the system class loader , then null will be returned. Further examination of the returned object is implementation specific. **Returns** a JavaObject representing the java.lang.ClassLoader instance **Throws** CorruptDataException — if the underlying data is in an unexpected state

Methods	
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same Java Class Loader in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaVMInitArgs

public interface JavaVMInitArgs

This class models the JavaVMInitArgs C structure passed to JNI_CreateJavaVM to create this Java Virtual Machine Typically the options passed to the JVM are similar but necessarily identical to these used to invoke the Java Virtual Machine from a command line.

Table 6.34. JavaVMInitArgs Methods

Methods	
getVersion	*public int getVersion()* Fetch the JNI version from the JavaVMInitArgs structure used to create this Java Virtual Machine. See the JNI specification for the meaning for this field. **Returns** the JNI version **Throws** DataUnavailable CorruptDataException

Methods	
getIgnoreUnrecognized	*public boolean getIgnoreUnrecognized()* Fetch the ignoreUnrecognized field from the JavaVMInitArgs structure used to create this Java Virtual Machine. See the JNI specification for the meaning for this field. **Returns** true if ignoreUnrecognized was set to a non-zero value when the Java Virtual Machine was invoked **Throws** DataUnavailable CorruptDataException
getOptions	*public List getOptions()* Fetch the options used to start this Java Virtual Machine, in the order they were originally specified. The returned list follows the standard semantics for javax.tools.diagnostics collections The order of the options returned in the list is the same as that passed to the to JNI_CreateJavaVM function. A list is always returned but could be empty **Returns** an List of JavaVMOptions **Throws** DataUnavailable

interface JavaMember

public interface JavaMember

Abstract interface which both JavaField and JavaMethod inherit from. It defines APIs which are common to both types of members. It is modelled on java.lang.reflect.Member

Table 6.35. JavaMember Methods

Methods	
getModifiers	*public int getModifiers()* Get the set of modifiers for this field or method - a set of bits The values for the constants representing the modifiers can be obtained from java.lang.reflect.Modifier. **Returns** the modifiers for this field or method. **Throws** CorruptDataException — if the underlying data is in an unexpected state
getDeclaringClass	*public JavaClass getDeclaringClass()* Get the class which declares this field or method **Returns** the JavaClass which declared this field or method **Throws** CorruptDataException — if the underlying data is in an unexpected state DataUnavailable — if there is no declaring class available
getName	*public String getName()* Get the name of the field or method **Returns** the name of the field or method **Throws** CorruptDataException — if the underlying data is in an unexpected state

Methods	
getSignature	*public String getSignature()* Get the signature of the field or method **Returns** the signature of the field or method. e.g. "(Ljava/lang/String;)V" **Throws** CorruptDataException — if the underlying data is in an unexpected state
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same Java Member in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaMonitor

public interface JavaMonitor

Represents the underlying monitor used by a Java Virtual Machine to manage locking and synchronization of a Java object.

The underlying monitor is implementation specific. Some implementations may choose to use their monitor implementations to control access to Java Virtual Machine resources that are not objects. In such cases, getObject() will return null.

Java programmers use the synchronized modifier on methods and the synchronized block within methods to control simultaneous access to Java objects. Java uses monitors for this synchronization, which can be implemented using a variety of techniques. The JavaMonitor class presents the simple monitor abstraction that allows the caller to determine:

- Which thread currently owns the monitor
- Which threads are waiting to be woken after they have gotten ownship of the monitor and relinquished it, normally within Object.wait() within a synchronized block or method.
- The threads that waiting to get ownership of the monitor. These are typically threads waiting to enter a synchronized block or method.

This API presents only what exists at the Java Virtual Machine bytecode level. The locking facilities provided by the java.util.concurrent.lock package are expected to be implemented on top of ordinary Java monitors.

Table 6.36. JavaMonitor Methods

Methods	
getObject	*public JavaObject getObject()* Get the object associated with this monitor. Not all JavaMonitors will have objects, as there may be JavaMonitors that are used to control access to internal Java Virtual Machine resources ("Raw" monitors). **Returns** the Java object associated with this monitor, or null.
getName	*public String getName()* Get the name of a monitor. For monitors not associated with object ("raw" monitors), it is expected that this method will return a descriptive name that is meaningful to the Java Virtual Machine implementation. For example "Heap lock" might be a monitor controlling exclusive access to the Java heap. For objects, the expectation is that the name will uniquely identify the object the monitor is associated with. This is not expected to necessarily be consistent between different dumps of the same JVM. **Returns** the name of the monitor (never null) **Throws** CorruptDataException
getOwner	*public JavaThread getOwner()* Get the thread which currently owns the monitor. This may be null if the monitor is not owned. **Returns** the owner of the monitor, or null if the monitor is not owned **Throws** CorruptDataException

Methods	
getEnterWaiters	*public List getEnterWaiters()* Get the set of threads waiting to enter the monitor. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned value is never null but can be an empty list. **Returns** a list of threads waiting to enter this monitor
getNotifyWaiters	*public List getNotifyWaiters()* Get the set of threads waiting to be notified on the monitor. They are usually threads in the java.lang.Object#wait() method. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned value is never null but can be an empty list. **Returns** a list of threads waiting to be notified on this monitor.
getID	*public ImagePointer getID()* Get the identifier for this monitor. **Returns** The pointer which uniquely identifies this monitor in memory.
equals	*public boolean equals(Object obj)* **Returns** true if the given object refers to the same Java Monitor in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaMethod

public interface JavaMethod

Represents a method or constructor in a class

Table 6.37. JavaMethod Methods

Methods	
getBytecodeSections	*public List getBytecodeSections()* Get the set of ImageSections containing the bytecode of this method. Each ImageSection contains data (usually bytecodes) used in executing this method in interpreted mode. The collection may be empty for native methods, or pre-compiled methods. Typically, the collection will contain no more than one section, but this is not guaranteed. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of ImageSections.
getCompiledSections	*public List getCompiledSections()* Get the set of ImageSections containing the compiled code of this method. Each ImageSection contains data (usually executable code) used in executing this method in compiled mode. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of ImageSections.

Methods	
getVariables	*public List getVariables()* An experimental addition to the API. Get the set of JavaVariable objects The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** List of JavaVariable objects available
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same JavaMethod in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaField

public interface JavaField

Represents a field declaration. It is modelled on java.lang.reflect.Field

Table 6.38. JavaField Methods

Methods	
get	*public Object get(JavaObject object)* Get the contents of a field of an Object. **Returns** a JavaObject instance for reference type fields, an instance of a subclass of Number, Boolean, or Character for primitive fields, or null for null reference fields. **Parameters** object — to fetch the field from. Ignored for static fields. This field must be declared in the object's class or in a superclass **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field

Methods	
getBoolean	*public boolean getBoolean(JavaObject object)* Get the contents of a boolean field **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the field is not a boolean.
getByte	*public byte getByte(JavaObject object)* Get the contents of a byte field **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the type of the field cannot be converted to byte

Methods	
getChar	*public char getChar(JavaObject object)* Get the contents of a char field **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the type of the field cannot be converted to char
getDouble	*public double getDouble(JavaObject object)* Get the contents of a double field or of another primitive field whose type is convertible to double via a widening conversion. **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the type of the field cannot be converted to double via a widening conversion

Methods	
getFloat	*public float getFloat(JavaObject object)* Get the contents of a float field or of another primitive field whose type is convertible to float via a widening conversion. **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the type of the field cannot be converted to float via a widening conversion
getInt	*public int getInt(JavaObject object)* Get the contents of an int field or of another primitive field whose type is convertible to int via a widening conversion. **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the type of the field cannot be converted to int via a widening conversion.

Methods	
getLong	*public long getLong(JavaObject object)* Get the contents of a long field or of another primitive field whose type is convertible to long via a widening conversion. **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the type of the field cannot be converted to long via a widening conversion.
getShort	*public short getShort(JavaObject object)* Get the contents of a short field or of another primitive field whose type is convertible to short via a widening conversion. **Returns** the field contents **Parameters** object — to fetch the field from. Ignored for static fields. **Throws** CorruptDataException — if the underlying data is in an unexpected state MemoryAccessException NullPointerException — if the field is an instance field, and object is null IllegalArgumentException — if the specified object is not appropriate for this field, or if the type of the field cannot be converted to short via a widening conversion.

Methods	
getString	*public String getString(JavaObject object)* Get the contents of a string field **Returns** a String representing the value of the String field. Note that the instance returned can be null if the field was null in object. **Parameters** object to fetch the field from. Ignored for static fields. **Throws** CorruptDataException if the underlying data is in an unexpected state MemoryAccessException IllegalArgumentException if the specified field is not a String NullPointerException if the field is an instance field, and object is null
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same Java Field in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaVMOption

public interface JavaVMOption

This class models the JavaVMOption C structures passed to the JNI invocation API entry point JNI_CreateJavaVM used to create a Java Virtual Machine. Each JavaVMOption consists of two components :

1. an optionString string, used to identify the option.

2. an extraInfo pointer, used to pass additional information. This component is usually null.

Table 6.39. JavaVMOption Methods

Methods	
getOptionString	*public String getOptionString()* Fetch the optionString component of the option. **Returns** a string representing the optionString. This is never null. **Throws** DataUnavailable CorruptDataException
getExtraInfo	*public ImagePointer getExtraInfo()* Fetch the extraInfo component of this option. **Returns** the pointer value from the extraInfo (usually null). **Throws** DataUnavailable CorruptDataException

interface JavaClass

public interface JavaClass

Represents a Java class.

A Java Class can have fields and methods. It is a shallow model of a loaded class file or special types such as array types or primitive types in the Java Virtual Machine.

Table 6.40. JavaClass Methods

Methods	
getObject	*public JavaObject getObject()* Fetch the java.lang.Class object associated with this class. In some implementations this may be null if no object has been created to represent this class, or if the class is synthetic. **Returns** the java.lang.Class object associated with this class **Throws** CorruptDataException — if the underlying data is in an unexpected state
getClassLoader	*public JavaClassLoader getClassLoader()* Fetch the class loader associated with this class. Classes defined in the bootstrap class loader (including classes representing primitive types or void) will always return a JavaClassLoader representing the bootstrap class loader. This asymmetry with java.lang.Class#getClassLoader() is intentional. **Returns** the JavaClassLoader in which this class was defined **Throws** CorruptDataException — if the class loader for this class cannot be found (a class cannot exist without a loader so this implies corruption)

Methods	
getName	*public String getName()* Get the name of the class in a form that follows the java.lang.Class#getName() definition. This method will always return a valid class name. **Returns** the name of the class **Throws** CorruptDataException — if the underlying data is in an unexpected state
getSuperclass	*public JavaClass getSuperclass()* Get the super class of this class. Will return the superclass of this class or null if no superclass exists. For JavaClass instances representing interfaces, java.lang.Object, primitive types (int,boolean,char etc) and void, calling this method will return null. **Returns** the immediate superclass of this class, or null if this class has no superclass. **Throws** CorruptDataException — if the underlying data is in an unexpected state
getInterfaces	*public List getInterfaces()* Get the set of names of interfaces directly implemented by the class represented by this JavaClass. Some JVM implementations may choose to load interfaces lazily, so only the names are returned. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of the names of interfaces directly implemented by this class.

Methods	
getModifiers	*public int getModifiers()* Return the Java language modifiers for this class. The modifiers are defined by the JVM Specification. Note that, for inner classes, the actual modifiers are returned, not the synthetic modifiers. For instance, a class will never have its 'protected' modifier set, even if the inner class was a protected member, since 'protected' is not a legal modifier for a class file. **Returns** the modifiers for this class **Throws** CorruptDataException — if the underlying data is in an unexpected state
isArray	*public boolean isArray()* This method returns true if the class represented by this JavaClass is an array class. **Returns** true if this class is an array class **Throws** CorruptDataException — if the underlying data is in an unexpected state
getComponentType	*public JavaClass getComponentType()* For array classes, returns a JavaClass representing the component type of this array class. **Returns** a JavaClass representing the component type of this array class **Throws** CorruptDataException — if the underlying data is in an unexpected state java.lang.IllegalArgumentException — if this JavaClass does not represent an array class

Methods	
getDeclaredFields	*public List getDeclaredFields()* Get the set of fields declared in this class. Fields declared in any superclass of this class are not returned. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of fields declared in this class.
getDeclaredMethods	*public List getDeclaredMethods()* Get the set of methods declared in this class. Methods declared in any superclass of this class are not returned. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of methods declared in this class.

Methods	
getConstantPoolReferences	*public List getConstantPoolReferences()* Returns the list of constant pool references defined by this class. Java classes may refer to other classes and to String objects via the class's constant pool. These references are followed by the garbage collector, forming edges on the graph of reachable objects. This getConstantPoolReferences() may be used to determine which objects are referred to by the receiver's constant pool. Although Java VMs typically permit only Class and String objects in the constant pool, some esoteric or future virtual machines may permit other types of objects to occur in the constant pool. This API imposes no restrictions on the types of JavaObjects which might be included in the list. No assumption should be made about the order in which constant pool references are returned. Classes may also refer to objects through static variables. These may be found with the getDeclaredFields() API. Objects referenced by static variables are not returned by getConstantPoolReferences() unless the object is also referenced by the constant pool. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of JavaObjects which are referred to by the constant pool of this class.
getID	*public ImagePointer getID()* The ID of a class is a pointer to a section of memory which identifies the class. The contents of this memory are implementation defined. In some implementations getID() and getObject().getID() may return the same value. This implies that the class object is also the primary internal representation of the class. API users should not rely on this behaviour. In some implementations, getID() may return null for some classes. **Returns** a pointer to the class

Methods	
getReferences	*public List getReferences()* Get the set of references from this class. A reference is a object that represents the uni-directional relationship between objects and classes. Objects and classes cannot be reclaimed by the Java Virtual Machine garbage collector if references exist that can ultimately be traced back to root references. see JavaReference for more detailed information. Since this API can present entities that exist at any point in their lifecycle, it is possible to encounter an JavaClass that is eligible for collection and thus no JavaReference can be found that refers to it. The returned list follows the standard semantics for javax.tools.diagnostics collections. The returned list is never null but could be empty. **Returns** a list of JavaReferences
equals	*public boolean equals(Object obj)* **Returns** True if the given object refers to the same Java Class in the image **Parameters** obj
hashCode	*public int hashCode()*

interface JavaVariable

public interface JavaVariable

Representation of a Java Variable

This is an experimental addition and may be removed at a later date

Table 6.41. JavaVariable Methods

Methods	
getName	*public String getName()* The name of the variable. **Throws** DataUnavailable if the information is not available
getSignature	*public String getSignature()* The local variable's signature in JNI format.
getStart	*public int getStart()* The start of the local variable's scope within the bytecode.
getLength	*public int getLength()* The number of bytes this variables scope covers over the bytecode.
getSlot	*public int getSlot()* The local variable slot this variable occupies. Passed to JavaStackFrame.getVariable() to retrieve the contents.
getValue	*public Object getValue()* The value of the variable

interface JavaThread

public interface JavaThread

Represents a Java thread.

`JavaThread`

instances correspond with executing threads in the Java Virtual Machine, not java.lang.Thread instances on the heap.

`JavaThread`

provide information on what was running including the locations of all of the threads within the Java program when the dump was taken.

Table 6.42. JavaThread Methods

Methods	
getJNIEnv	*public ImagePointer getJNIEnv()* Get the address of the JNIEnv structure which represents this thread instance in JNI. **Returns** the address of the JNIEnv structure which represents this thread instance in JNI. **Throws** CorruptDataException
getPriority	*public int getPriority()* Get the Java priority of the thread. The value returned will be the same as what would have been returned by a call to java.lang.Thread#getPriority() within the Java Virtual Machine. **Returns** the Java priority of the thread (a number between 1 and 10 inclusive) **Throws** CorruptDataException

Methods	
getObject	*public JavaObject getObject()* Returns the JavaObject representing the instance of the class or subclass of java.lang.Thread that represents this thread in the Java Virtual Machine. The object returned is the java.lang.Thread instance the method java.lang.Thread#start() start() was executed against in order to create this Java thread. This method may return `null` when there is no java.lang.Thread instance associated with this Java thread. Some Java threads may be created for purposes other than for executing Java code (for example, for garbage collection). **Returns** a JavaObject representing the java.lang.Thread associated with this thread, or null. **Throws** CorruptDataException — if the reference to java.lang.Thread is not null and cannot be retrieved.
getState	*public int getState()* Get the state of the thread when the dump was generated. The result is a bit vector, and uses the states defined by the function GetThreadState in the JVMTI specification. **Returns** the state of the thread when the image was created. **Throws** CorruptDataException — If the thread state could not be successfully retrieved.

Methods	

getImageThread	*public ImageThread getImageThread()*
	
	Returns the operating system level thread that executes the Java thread.
	
	This will return an ImageThread if an operating system level thread can be returned, otherwise the
	
	`DataUnavailable`
	
	exception is thrown. There is no guarantee that there is a 1:1 relationship between
	
	`JavaThreads`
	
	and ImageThread ImageThreads.
	
	Returns
	the ImageThread which this thread is currently bound to.
	
	Throws
	CorruptDataException — If the underlying resource describing the native representation of the thread is damaged.
	DataUnavailable — If no mapping is provided due to missing or limited underlying resources.

getStackSections	*public List getStackSections()* Get the List of ImageSection ImageSections which make up the Java Virtual Machine stack. Some Java Virtual Machine implementations may use parts of the ImageThread's stack for JavaStackFrames. **Returns** a collection of ImageSections which make up the Java stack.

Methods	
getStackFrames	*public List getStackFrames()* Get the set of stack frames. The start of the list will contain the top most stack frame, the last entry will contain the bottom most stack frame. The top contains the most recently executing stack frame. This method may return an empty list when there are no Java stack frames associated with this Java thread. `null` must never be returned. **Returns** a list of Java stack frames in order from top to bottom.
getName	*public String getName()* Return the name of the thread. Usually this is derived from the object associated with the thread, but if the name cannot be derived this way (e.g. there is no object associated with the thread) a name will be created for the thread. **Returns** the name of the thread **Throws** CorruptDataException If a name exists but cannot be retrieved.
equals	*public boolean equals(Object obj)* **Returns** true if the given object refers to the same Java Thread in the image **Parameters** obj
hashCode	*public int hashCode()*

package javax.tools.diagnostics.vm

Dump Creation API
Provides standard mechanisms for initiating a Dump programmatically

Table 6.43. Interface Summary

Name	Summary
DumpHandle	Triggers a dump.
DumpInitiatorDelegate	Interface that describes the required capabilities of specific dump initiator.

Table 6.44. Class Summary

Name	Summary
DumpFactory	Standard mechanism that allows a java application to trigger a dump for the executing JVM.
DumpDescriptor	Definition of the capabilities of a Dump that should be produced by passing an instance of this descriptor to the Dump class

Details

interface DumpHandle

public interface DumpHandle

Triggers a dump. The location and type of dump produced is implementation specific but is guided by the DumpDescriptor used to create this handle.

Table 6.45. DumpHandle Methods

Methods	
dump	*public boolean dump()*

interface DumpInitiatorDelegate

public interface DumpInitiatorDelegate

Interface that describes the required capabilities of specific dump initiator. Classes that implement this interface can be called by the Dump class to trigger a specific dump.

Table 6.46. DumpInitiatorDelegate Methods

Methods	
getCapabilities	*public DumpInitiatorCapabilities getCapabilities()* Returns an object that describes the capabilities offered by this initiator **Returns** a populated capabilities object
available	*public boolean available()* Called to check that the delegate has all available resources to proceed. This method should only be called by the Dump class This method will always be called before the first call to dump **Returns** true if the delegate is available
createDumpHandle	*public DumpHandle createDumpHandle()* Returns a Dump handle for a default configured dump **Returns** valid dump handle
createDumpHandle	*public DumpHandle createDumpHandle(DumpDescriptor descriptor)* Returns a Dump handle for a dump that will match the provided descriptor on a "best can do" basis. **Returns** valid dump handle

Methods	
getDumpType	*public String getDumpType()* Returns a literal that describes the dump being produced by this initiator. The value can be used by callers to the DumpFactory API to retrieve a specific dump initiator. **Returns** dump type literal

class DumpFactory

public DumpFactory extends java.lang.Object

Standard mechanism that allows a java application to trigger a dump for the executing JVM.

Table 6.47. DumpFactory Constructor Summary

Constructor
public DumpFactory() Instantiate a dump factory that can be used to trigger dumps

Table 6.48. DumpFactory Methods

Methods	
getDefaultInitiatorDelegate	*public DumpInitiatorDelegate getDefaultInitiatorDelegate()* Get the default dump initiator
setDefaultInitiatorDelegate	*public void setDefaultInitiatorDelegate(DumpInitiatorDelegate defaultInitiatorDelegate)* sets the default dump initiator. **Parameters** defaultInitiatorDelegate
dump	*public void dump()* Fastpath method allowing the user to trigger a dump using the default dump method **Throws** IOException
dump	*public void dump(String id)* Fastpath method allowing the user to trigger a standard dump for the given data format **Throws** IOException

Methods	
dump	*public void dump(DumpDescriptor desc)* Fastpath method to create a dump from a dump type that can support the data requested. **Returns** a valid dump handle
getDefault	*public static DumpFactory getDefault()* Get the default Dump factory. Returns a Dump factory populated with all available dump initiators **Returns** default dump factory
loadStandardInitiators	*public void loadStandardInitiators()* Looks for commonly available dump initiators and add them to this instances configuration
createDumpHandle	*public DumpHandle createDumpHandle()* Returns a Dump handle for the default dump type used by this JVM **Returns** a valid dump handle
createDumpHandle	*public DumpHandle createDumpHandle(DumpDescriptor desc)* Returns a Dump handle for a dump type that can support the data requested. **Returns** a valid dump handle
getInitiator	*public DumpInitiatorDelegate getInitiator(String format)* Returns an initiator that can produce a dump of the required format. If no initiator exists which can handle the format then null is returned **Returns** supporting initiator or null **Parameters** format
getAvailableInitiators	*public Collection getAvailableInitiators()* Returns the set of available Initiators. Always returns a set.

Methods	
instantiateOverrideDelegate	*public DumpInitiatorDelegate instantiateOverrideDelegate()* Instantiates a Dump delegate class to handle dump requests based on the presence of the override system property initiatorPropertyName It will be used in preference to any other auto discovered delegates. If the override does not exist or cannot be instantiated then null is returned **Returns** override delegate instance

class DumpDescriptor

public DumpDescriptor extends java.lang.Object

Definition of the capabilities of a Dump that should be produced by passing an instance of this descriptor to the Dump class

Table 6.49. DumpDescriptor Constructor Summary

Constructor
public DumpDescriptor()

Table 6.50. DumpDescriptor Methods

Methods	
setRecordClassLoaders	*public void setRecordClassLoaders()* Call to indicate that the dump generated by the initiator is required to contain data that will eventually be represented by the JavaClassLoader interface
setRecordClasses	*public void setRecordClasses()* Call to indicate that the dump generated by the initiator is required to contain data that will eventually be represented by the JavaClass interface
recordClassLoadersRequired	*public boolean recordClassLoadersRequired()* Returns true if the recording of class loaders in the generated dump is a required attribute. Returns false if it is optional.
recordClassesRequired	*public boolean recordClassesRequired()*
copy	*public DumpDescriptor copy()* Returns a copy of this descriptor. **Returns** new copy.
clone	*protected Object clone()*

package javax.tools.diagnostics.vm.spi

Table 6.51. Interface Summary

Name	Summary
DumpInitiatorCapabilities	Description of the capabilities offered by a particular Dump Initiator

Details

interface DumpInitiatorCapabilities

public interface DumpInitiatorCapabilities

Description of the capabilities offered by a particular Dump Initiator

package javax.tools.diagnostics.vm.spi.delegates

Table 6.52. Class Summary

Name	Summary
HProfSignalTriggeredDumpDelegate	
HProfMBeanDumpDelegate	Uses HotSpotDiagnostic MBean to generate hprof dump.
IBMSPIBasedHeapDumpDelegate	
JavaDumpDelegate	
XMLDumpWriter	
AbstractSignalBasedDumpInitiatorDelegate	Signal based Dump Initiator relies on the presence of org.apache.kato.common142.DumpTrigger in the class path
IBMSPIBasedSystemDumpDelegate	
AbstractIBMSPIBasedDumpInitiatorDelegate	Dump Delegate for IBM JVMs using the com.ibm.jvm.Dump API Note that reflection is used to trigger a dump as otherwise compilation of this code would be dependent on having an IBM JVM.

Details

class HProfSignalTriggeredDumpDelegate

public HProfSignalTriggeredDumpDelegate extends javax.tools.diagnostics.vm.spi.delegates.AbstractSignalBasedDumpInitiatorDelegate

Table 6.53. HProfSignalTriggeredDumpDelegate Constructor Summary

Constructor
public HProfSignalTriggeredDumpDelegate()

Table 6.54. HProfSignalTriggeredDumpDelegate Methods

Methods	
available	*public boolean available()*
getCapabilities	*public DumpInitiatorCapabilities getCapabilities()*
getDumpType	*public String getDumpType()*

class HProfMBeanDumpDelegate

public HProfMBeanDumpDelegate extends java.lang.Object

Uses HotSpotDiagnostic MBean to generate hprof dump. Only available on Sun's hotspot.

Table 6.55. HProfMBeanDumpDelegate Constructor Summary

Constructor
public HProfMBeanDumpDelegate()

Table 6.56. HProfMBeanDumpDelegate Methods

Methods	
available	*public boolean available()* Checks to see if the MBean is registered. On IBM JVMs, this will return false.
createDumpHandle	*public DumpHandle createDumpHandle()*
createDumpHandle	*public DumpHandle createDumpHandle(DumpDescriptor descriptor)*
getCapabilities	*public DumpInitiatorCapabilities getCapabilities()*
getDumpType	*public String getDumpType()*

class IBMSPIBasedHeapDumpDelegate

public IBMSPIBasedHeapDumpDelegate extends javax.tools.diagnostics.vm.spi.delegates.AbstractIBMSPIBasedDumpInitiatorDelegate

Table 6.57. IBMSPIBasedHeapDumpDelegate Constructor Summary

Constructor
public IBMSPIBasedHeapDumpDelegate()

Table 6.58. IBMSPIBasedHeapDumpDelegate Methods

Methods	
getMethodName	*protected String getMethodName()*
getCapabilities	*public DumpInitiatorCapabilities getCapabilities()*
getDumpType	*public String getDumpType()*

class JavaDumpDelegate

public JavaDumpDelegate extends java.lang.Object

Table 6.59. JavaDumpDelegate Constructor Summary

Constructor
public JavaDumpDelegate()

Table 6.60. JavaDumpDelegate Methods

Methods	
available	*public boolean available()*
createDumpHandle	*public DumpHandle createDumpHandle()*
createDumpHandle	*public DumpHandle createDumpHandle(DumpDescriptor descriptor)*
getCapabilities	*public DumpInitiatorCapabilities getCapabilities()*
getDumpType	*public String getDumpType()*

class XMLDumpWriter

public XMLDumpWriter extends java.lang.Object

Table 6.61. XMLDumpWriter Constructor Summary

Constructor
public XMLDumpWriter()

Table 6.62. XMLDumpWriter Methods

Methods	
write	*public void write(File output)*
write	*public void write(Writer writer)*

class AbstractSignalBasedDumpInitiatorDelegate

public abstract AbstractSignalBasedDumpInitiatorDelegate extends java.lang.Object

Signal based Dump Initiator relies on the presence of org.apache.kato.common142.DumpTrigger in the class path

Table 6.63. AbstractSignalBasedDumpInitiatorDelegate Constructor Summary

Constructor
public AbstractSignalBasedDumpInitiatorDelegate()

Table 6.64. AbstractSignalBasedDumpInitiatorDelegate Methods

Methods	
available	*public boolean available()*
createDumpHandle	*public DumpHandle createDumpHandle()*
createDumpHandle	*public DumpHandle createDumpHandle(DumpDescriptor descriptor)*

class IBMSPIBasedSystemDumpDelegate

public IBMSPIBasedSystemDumpDelegate extends javax.tools.diagnostics.vm.spi.delegates.AbstractIBMSPIBasedDumpInitiatorDelegate

Table 6.65. IBMSPIBasedSystemDumpDelegate Constructor Summary

Constructor
public IBMSPIBasedSystemDumpDelegate()

Table 6.66. IBMSPIBasedSystemDumpDelegate Methods

Methods	
getMethodName	*protected String getMethodName()*
getCapabilities	*public DumpInitiatorCapabilities getCapabilities()*
getDumpType	*public String getDumpType()*

class AbstractIBMSPIBasedDumpInitiatorDelegate

public abstract AbstractIBMSPIBasedDumpInitiatorDelegate extends java.lang.Object

Dump Delegate for IBM JVMs using the com.ibm.jvm.Dump API Note that reflection is used to trigger a dump as otherwise compilation of this code would be dependent on having an IBM JVM.

Table 6.67. AbstractIBMSPIBasedDumpInitiatorDelegate Constructor Summary

Constructor
public AbstractIBMSPIBasedDumpInitiatorDelegate()

Table 6.68. AbstractIBMSPIBasedDumpInitiatorDelegate Methods

Methods	
available	*public boolean available()*
getMethodName	*protected abstract String getMethodName()*
createDumpHandle	*public DumpHandle createDumpHandle()*
createDumpHandle	*public DumpHandle createDumpHandle(DumpDescriptor descriptor)*

Appendix A. Register tables

Common Register Names

Table A.1. IA32 Register Names

Register	Type
EDI	Integer
ESI	Integer
EAX	Integer
EBX	Integer
ECX	Integer
EDX	Integer
EIP	Integer
ESP	Integer
EBP	Integer

Table A.2. AMD64 Register Names

Register	Type
RDI	Long
RSI	Long
RAX	Long
RBX	Long
RCX	Long
RDX	Long
R8	Long
R9	Long
R10	Long
R11	Long
R12	Long
R13	Long
R14	Long
R15	Long
RIP	Long
RSP	Long
RBP	Long

Table A.3. PowerPC 32 Register Names

Register	Type
R0	Integer

Register	Type
R1	Integer
R2	Integer
R3	Integer
R4	Integer
R5	Integer
R6	Integer
R7	Integer
R8	Integer
R9	Integer
R10	Integer
R11	Integer
R12	Integer
R13	Integer
R14	Integer
R15	Integer
R16	Integer
R17	Integer
R18	Integer
R19	Integer
R20	Integer
R21	Integer
R22	Integer
R23	Integer
R24	Integer
R25	Integer
R26	Integer
R27	Integer
R28	Integer
R29	Integer
R30	Integer
R31	Integer
IAR	Integer
LR	Integer
MSR	Integer
CTR	Integer
CR	Integer
FPSCR	Integer
XER	Integer

Register	Type
TID	Integer
MQ	Integer

Table A.4. PowerPC 64 Register Names

Register	Type
R0	Long
R1	Long
R2	Long
R3	Long
R4	Long
R5	Long
R6	Long
R7	Long
R8	Long
R9	Long
R10	Long
R11	Long
R12	Long
R13	Long
R14	Long
R15	Long
R16	Long
R17	Long
R18	Long
R19	Long
R20	Long
R21	Long
R22	Long
R23	Long
R24	Long
R25	Long
R26	Long
R27	Long
R28	Long
R29	Long
R30	Long
R31	Long
IAR	Long

Register	Type
LR	Long
MSR	Long
CTR	Long
CR	Long
FPSCR	Long
XER	Long

Table A.5. z/Series 31 Register Names

Register	Type
gpr0	Integer
gpr1	Integer
gpr2	Integer
gpr3	Integer
gpr4	Integer
gpr5	Integer
gpr6	Integer
gpr7	Integer
gpr8	Integer
gpr9	Integer
gpr10	Integer
gpr11	Integer
gpr12	Integer
gpr13	Integer
gpr14	Integer
gpr15	Integer
psw0	Integer
psw1	Integer

Table A.6. z/Series 64 Register Names

Register	Type
gpr0	Long
gpr1	Long
gpr2	Long
gpr3	Long
gpr4	Long
gpr5	Long
gpr6	Long
gpr7	Long
gpr8	Long
gpr9	Long
gpr10	Long
gpr11	Long
gpr12	Long
gpr13	Long
gpr14	Long
gpr15	Long
psw0	Long
psw1	Long

Appendix B. Opening Images example

This class takes the name of at least one dump, and the name of a class that implements `ImageAnalyzer` listed in Appendix F, *ImageAnalyzer interface*.

```
/*******************************************************************************
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *******************************************************************************/
package org.apache.kato.examples;

import java.io.File;
import java.io.IOException;

import javax.tools.diagnostics.FactoryRegistry;
import javax.tools.diagnostics.image.Image;

/**
 * This example shows how an Image is opened, given a snapshot's file name.
 *
 */
public abstract class ReadImage {

 /**
  * Main method. Takes names of snapshot dumps and opens them with the
  * ImageFactory.
  *
  * @param args
  */
 public static void main(String[] args) {
  if (args.length < 2) {
   System.err.println("Usage: ReadImage <snapshot> [snapshot...] <Image analyzer>");
  }

  String analyzerName = "org.apache.kato.examples."+args[args.length-1];
  Class analyzerClass = null;
  try {
   analyzerClass = Class.forName(analyzerName);
  } catch (ClassNotFoundException e) {
   System.err.println("Unable to find test class `"+analyzerName+"'");
   e.printStackTrace();
   System.exit(1);
  }

  for(int i = 0; i < args.length-1; i++) {
   String filename = args[i];
   System.out.println("\nAnalysing `"+filename+"'");
   System.out.println("==========================");

   Image image;
   try {
    image = FactoryRegistry.getDefaultRegistry().getImage(new File(filename));
   } catch (IOException e) {
    System.out.println("Unable to open snapshot.");
    e.printStackTrace();
    continue;
   }
   try {
    ImageAnalyzer analyzerInstance =  (ImageAnalyzer) analyzerClass.newInstance();
```

```
     analyzerInstance.analyze(image);
    } catch (InstantiationException e) {
     e.printStackTrace();
    } catch (IllegalAccessException e) {
     e.printStackTrace();
    } catch (ClassCastException e) {
     System.out.println("Analzyer class should be instances of ImageAnalyzer.");
     e.printStackTrace();
     System.exit(2);
    }

   }
  }
 }
```

Appendix C. Snapshot Cause Example

This class implements the `ImageAnalyzer` interface in Appendix F, *ImageAnalyzer interface*.

```
/*******************************************************************************
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *******************************************************************************/
package org.apache.kato.examples;

import javax.tools.diagnostics.image.CorruptDataException;
import javax.tools.diagnostics.image.DataUnavailable;
import javax.tools.diagnostics.image.Image;
import javax.tools.diagnostics.image.ImageAddressSpace;
import javax.tools.diagnostics.image.ImageProcess;
import javax.tools.diagnostics.image.ImageStackFrame;
import javax.tools.diagnostics.image.ImageThread;
import javax.tools.diagnostics.runtime.ManagedRuntime;
import javax.tools.diagnostics.runtime.java.JavaRuntime;
import javax.tools.diagnostics.runtime.java.JavaStackFrame;
import javax.tools.diagnostics.runtime.java.JavaThread;

/**
 * This analyzer determines what process and thread
 * caused the dump, and what signal, if any.
 *
 */
public class CauseAnalyzer implements ImageAnalyzer {

 @Override
 public void analyze(Image image) {
  for (ImageAddressSpace as : image.getAddressSpaces()) {
   ImageProcess process = as.getCurrentProcess();

   // Only invoked if there is a "current" process,
   // This is a process that caused the dump to occur.
   if (process != null) {
    try {
     int signum = process.getSignalNumber();
     String signame = process.getSignalName();

     // Identify the process by number and command line.
     System.out.println("Process "+process.getID()+
       " was started with `" +
       process.getCommandLine()+"'");

     // The signals that cause the dump to be generated
     if (signame != null) {
      System.out.println("Dump caused by signal " + signame+"("+signum+")");
     }

     ImageThread thread = process.getCurrentThread();
     // Identify the thread, by id, various properties and a stack trace.
     if (thread != null) {
      System.out.println("\nDump caused by thread "+
        thread.getID()+
        ", "+thread.getProperties());
      for(ImageStackFrame frame: thread.getStackFrames()) {
       System.out.println("\t" + frame);
```

```
      }

      // Find JavaThread and then do stacktrace.
RUNTIME:    for(ManagedRuntime runtime : process.getRuntimes() ) {
       if (runtime instanceof JavaRuntime) {
        JavaRuntime jr = (JavaRuntime) runtime;

        for(JavaThread jthread : jr.getThreads()) {
         if (thread.equals(jthread.getImageThread())) {
          System.out.println("\nDump caused by JavaThread "+
            jthread.getName());

          for(JavaStackFrame frame : jthread.getStackFrames()) {
           System.out.println("\t" + frame);
          }
          break RUNTIME;
         }
        }
       }
      }
     }
    } catch (DataUnavailable e) {
     e.printStackTrace();
    } catch (CorruptDataException e) {
     e.printStackTrace();
    }
   }
  }
 }
}
```

Appendix D. Identifying Java VM Example

This class implements the `ImageAnalyzer` interface in Appendix F, *ImageAnalyzer interface*.

```
/*******************************************************************************
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *******************************************************************************/
package org.apache.kato.examples;

import javax.tools.diagnostics.image.CorruptDataException;
import javax.tools.diagnostics.image.DataUnavailable;
import javax.tools.diagnostics.image.Image;
import javax.tools.diagnostics.image.ImageAddressSpace;
import javax.tools.diagnostics.image.ImageModule;
import javax.tools.diagnostics.image.ImagePointer;
import javax.tools.diagnostics.image.ImageProcess;
import javax.tools.diagnostics.runtime.ManagedRuntime;
import javax.tools.diagnostics.runtime.java.JavaRuntime;
import javax.tools.diagnostics.runtime.java.JavaVMOption;
/**
 * This Analyzer generates a reports information useful for
 * identifying the JVM that was running when the snapshot was taken.
 * Elements from the Java and Image APIs are used.
 */
public class WhatAnalyzer implements ImageAnalyzer {

 @Override
 public void analyze(Image image) {
  // Report the hostname.
  String hostname;
  try {
   hostname = image.getHostName();
  } catch (DataUnavailable e) {
   hostname = "<Could not retrieve hostname";
  } catch (CorruptDataException e) {
   hostname = "<Error retrieving hostname>";
   e.printStackTrace();
  }

  System.out.println("Snapshot was generated on " + hostname);

  for (ImageAddressSpace as : image.getAddressSpaces()) {
   for(ImageProcess process : as.getProcesses()) {
    String processID;
    try {
     processID = process.getID();
    } catch (DataUnavailable e) {
     processID = "<Unknown>";
    } catch (CorruptDataException e) {
     processID = "<Error>";
     e.printStackTrace();
    }
    System.out.println("Process ID="+processID);

    String commandLine;
    try {
     commandLine = process.getCommandLine();
    } catch (DataUnavailable e) {
```

```
      commandLine = "<Unknown>";
     } catch (CorruptDataException e) {
      commandLine = "<Error>";
      e.printStackTrace();
     }
     System.out.println("Command line: "+commandLine);

     String executable;
     try {
      executable = process.getExecutable().getName();
     } catch (CorruptDataException e) {
      executable = "<Unknown>";
     } catch (DataUnavailable e) {
      executable = "<Error>";
      e.printStackTrace();
     }
     System.out.println("Process Executable "+ executable);

     System.out.println("Loaded Libraries:");
     try {
      for(ImageModule module : process.getLibraries()) {
       System.out.println("\t" + module.getName());
      }
     } catch (DataUnavailable e) {
      System.out.println("No libraries found.");
     } catch (CorruptDataException e) {
      System.out.println("Error retrieving libraries:");
      e.printStackTrace();
     }

     for (ManagedRuntime runtime : process.getRuntimes()) {
      if (runtime instanceof JavaRuntime) {
       analyzeRuntime ((JavaRuntime) runtime);
      }
     }
    }
   }

   public void analyzeRuntime(JavaRuntime jr) {

    try {
     System.out.println("Java VM version:`"+jr.getVersion()+"'");
    } catch (CorruptDataException e1) {
     System.out.println("Error retrieving Java VM version");
     e1.printStackTrace();
    }

    System.out.println("VM options:");
    try {
     for (JavaVMOption option : jr.getJavaVMInitArgs().getOptions()) {
      String optionString = "\t\t\""+option.getOptionString()+"\"";

      ImagePointer extra = option.getExtraInfo();
      if (extra != null) {
       optionString += ", extraInfo=0x"+Long.toHexString(extra.getAddress());
      }

      System.out.println(optionString);
     }
    } catch (DataUnavailable e) {
     System.out.println("Unable to report VM options");
    } catch (CorruptDataException e) {
     System.out.println("Error retrieving VM options");
     e.printStackTrace();
    }
   }
  }
```

Appendix E. Retrieving Object Fields Example

This class extends the `RuntimeAnalyzer` class in Appendix G, *Retrieval of all JavaRuntimes* to simplify obtaining a `JavaRuntime` instance.

```
/*******************************************************************************
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *******************************************************************************/
package org.apache.kato.examples;

import java.lang.reflect.Array;

import javax.tools.diagnostics.image.CorruptDataException;
import javax.tools.diagnostics.image.DiagnosticException;
import javax.tools.diagnostics.image.ImagePointer;
import javax.tools.diagnostics.image.MemoryAccessException;
import javax.tools.diagnostics.runtime.java.JavaClass;
import javax.tools.diagnostics.runtime.java.JavaField;
import javax.tools.diagnostics.runtime.java.JavaHeap;
import javax.tools.diagnostics.runtime.java.JavaObject;
import javax.tools.diagnostics.runtime.java.JavaRuntime;

/**
 * This examples walks over all objects on the heap and prints the values
 * of all of the fields, and the contents of all elements of all the arrays.
 */
public class ObjectFields extends RuntimeAnalyzer{

 /**
  * Given a JavaRuntime, print out all of the objects on the heap.
  * @param jr JavaRuntime to get objects from
  */
 public void analyzeRuntime(JavaRuntime jr) {
  for (JavaHeap heap : jr.getHeaps()) {
    walkHeap (heap);
  }
 }

 /**
  * Walks over all objects on the heap.
  * @param heap JavaHeap to iterate over.
  */
 public void walkHeap(JavaHeap heap) {
  for (JavaObject jObject : heap.getObjects()) {

   try {
    if (jObject.isArray()) {
     walkArray (jObject);
    } else {
     walkObject (jObject);
    }

    // Even determining whether or not the JavaObject is an array has difficulties.
   } catch (CorruptDataException e) {
    System.err.println("Corrupt data exception calling jObject.isArray() at "+
```

```
      pointerToHexString(jObject.getID()));
    e.printStackTrace();
   }
  }
 }

 /**
  * Prints out all of the values of the fields in an object, along with
  * identifying information of the type itself.
  *
  * @param jObject A JavaObject
  */
 public void walkObject(JavaObject jObject) {
  // Just identify the object by its ID - this would the address on the heap.
  System.out.println("JavaObject @ " + pointerToHexString(jObject.getID()));
  // Handle indentation.
  String prefix = "\t";

  // Get the type of this object.
  JavaClass clazz;
  try {
   clazz = jObject.getJavaClass();
  } catch (CorruptDataException e) {
   System.err.println(prefix+"Error getting JavaClass");
   e.printStackTrace();
   return;
  }

  while (clazz != null) {
   // print out the name of the class and the the fields.
   try {
    System.out.println( prefix + clazz.getName() +":");
    prefix += "  ";

    /* Print out all fields for this class.
     */
    for (JavaField nextField : clazz.getDeclaredFields()) {
     printField(prefix, nextField, jObject);
    }
   } catch (DiagnosticException e) {
    System.err.println("Error printing out fields.");
    e.printStackTrace();
   }

   // Get the next superclass.
   try {
    clazz = clazz.getSuperclass();
   } catch (CorruptDataException e) {
    e.printStackTrace();
    break;
   }

  } // while (class != null)
 }

 /**
  * Print out the content of one field.
  * Only prints out instance fields. Will return if a
  * static field is passed.
  *
  * @param prefix Pad out message
  * @param field The field to print
  * @param object The object to print out
  * @throws CorruptDataException if something goes wrong
  * @throws MemoryAccessException
  */
 private void printField(String prefix, JavaField field, JavaObject object) throws CorruptDataException, Mem

  if (java.lang.reflect.Modifier.isStatic(field.getModifiers())) {
   return;
  }
```

```
 Object fieldValue = field.get(object);

 String valueString = "";

 // Format the field's value.
 if(fieldValue == null) {
  valueString = "<null reference>";

  // Most of the primitive fields can be handled as Number instances.
 } else if (fieldValue instanceof Number) {
  valueString = fieldValue.toString();
 } else if (fieldValue instanceof Character) {
  valueString = "`" + (Character)fieldValue + "'";
 } else if (fieldValue instanceof Boolean) {
  valueString = ((Boolean) fieldValue).booleanValue() ? "true" : "false";
 } else if (fieldValue instanceof JavaObject) {
  // Note how we have to get an instance of the object to know anything about it.
  JavaObject reference = (JavaObject) fieldValue;

  // classname: @ 0xadddress
  valueString = reference.getJavaClass().getName() + ": @ " + pointerToHexString(reference.getID());

  if ("java/lang/String".equals(reference.getJavaClass().getName())) {
   valueString += valueString +" = \"" + field.getString(object) + "\"";
  }

 }

 System.out.println(prefix + field.getSignature() + " " +
   field.getName() + " = " + valueString);
}

/**
 * Print out the contents of an array.
 *
 * @param jObject JavaObject of an array.
 */
public void walkArray(JavaObject object) {
 // Just identify the object by its ID - this would the address on the heap.
 System.out.println("JavaObject @ " + pointerToHexString(object.getID()));
 // Handle indentation.
 String className;

 JavaClass clazz;
 try {
  clazz = object.getJavaClass();
 } catch (CorruptDataException e) {
  System.err.println("Unable to determine array's JavaClass. aborting");
  e.printStackTrace();
  return;
 }

 // The class name is needed to determine the element types
 try {
  className = clazz.getName();
 } catch (CorruptDataException e) {
  System.err.println("Error getting Array class name.");
  e.printStackTrace();
  return;
 }


 int arraySize = 0;

 // This gets the number of elements in the array.
 try {
  arraySize = object.getArraySize();
 } catch (CorruptDataException e) {
  System.err.println("Unable to get object size.");
  e.printStackTrace();
  return;
 }
```

```
String componentName;
try {
 componentName = clazz.getComponentType().getName();
} catch (CorruptDataException e) {
 System.err.println("Unable to determine component type name. Quitting.");
 e.printStackTrace();
 return;
}

Object arrayCopy;

if ("boolean".equals(componentName)) {
 arrayCopy = new boolean[arraySize];
} else if ("byte".equals(componentName)) {
 arrayCopy = new byte[arraySize];
} else if ("char".equals(componentName)) {
 arrayCopy = new char[arraySize];
} else if ("short".equals(componentName)) {
 arrayCopy = new short[arraySize];
} else if ("int".equals(componentName)) {
 arrayCopy = new int[arraySize];
} else if ("long".equals(componentName)) {
 arrayCopy = new long[arraySize];
} else if ("float".equals(componentName)) {
 arrayCopy = new float[arraySize];
} else if ("double".equals(componentName)) {
 arrayCopy = new double[arraySize];
} else {
 // Anything element that is not a primitive must be a reference type.
 arrayCopy = new JavaObject[arraySize];
}

/* Copy the contents of the JavaObject array into the array we prepared earlier.
 * This is the only way to get the values of an array in a dump.
 */
try {
 object.arraycopy(0, arrayCopy, 0, arraySize);
} catch (CorruptDataException e) {
 e.printStackTrace();
 return;
} catch (MemoryAccessException e) {
 e.printStackTrace();
 return;
} catch (IllegalArgumentException e){
 e.printStackTrace();
 return;
} catch (IndexOutOfBoundsException e) {
 e.printStackTrace();
 return;
}

/* Go through every element in the copy of the array and print out its contents.
 * This example uses java.lang.Array.get() to make this easy to handle generically.
 *
 * Object references - JavaObjects - are only printed out at the type name @ an address.
 * We don't print out subarrays - they will be encountered later in the heap anyhow.
 */
System.out.println("\t" + className + "[" + arraySize +"] = {");
for (int cnt=0; cnt < arraySize; cnt++) {
 Object obj = Array.get(arrayCopy, cnt);

 if (obj == null) {
  System.out.println("\t\tnull,");
 } else if (obj instanceof JavaObject) {
  JavaObject refObj = (JavaObject) obj;
  try {
   System.out.println("\t\t" + refObj.getJavaClass().getName()+ " @ "+
     pointerToHexString(refObj.getID())+",");
  } catch (CorruptDataException e) {
   System.err.println("\t\t CorruptDataException while printing out array element");
   e.printStackTrace();
```

```
    }
   } else {
    System.out.println("\t\t"+obj+",");
   }
  }
  System.out.println("\t};");
 }

 /**
  * Takes ImagePointer and returns it as a hex string. Perhaps this should be defined
  * behaviour for ImagePointer.toString().
  * @param pointer ImagePointer
  * @return Address of pointer as a hex string prefixed with "0x"
  */
 public static String pointerToHexString(ImagePointer pointer) {
  return "0x"+Long.toHexString(pointer.getAddress());
 }
}
```

Appendix F. ImageAnalyzer interface

```
/*****************************************************************************
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *****************************************************************************/
package org.apache.kato.examples;

import javax.tools.diagnostics.image.Image;

public interface ImageAnalyzer {
 public abstract void analyze(Image image);
}
```

Appendix G. Retrieval of all JavaRuntimes

This class allows subclasses to be given an `Image` and have their `analyzerRuntime(JavaRuntime)` methods invoked. This class implements the `ImageAnalyzer` interface in Appendix F, *ImageAnalyzer interface*.

```
/*****************************************************************************
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *****************************************************************************/
package org.apache.kato.examples;

import javax.tools.diagnostics.image.Image;
import javax.tools.diagnostics.image.ImageAddressSpace;
import javax.tools.diagnostics.image.ImageProcess;
import javax.tools.diagnostics.runtime.ManagedRuntime;
import javax.tools.diagnostics.runtime.java.JavaRuntime;

public abstract class RuntimeAnalyzer implements ImageAnalyzer {

 /**
  * Calls the analyzeRuntime(JavaRuntime jr) method against all JavaRuntime
  * instances found in the image.
  *
  * @param image Image to analyse
  */
 @Override
 public void analyze(Image image) {
  for(ImageAddressSpace as : image.getAddressSpaces()) {
   for(ImageProcess process : as.getProcesses()) {
    for(ManagedRuntime runtime : process.getRuntimes()) {
     if (runtime instanceof JavaRuntime) {
      analyzeRuntime((JavaRuntime) runtime);
     }
    }
   }
  }
 }

 /**
  * Override this method to analyze just the JavaRuntime.
  *
  * @param jr
  */
 public abstract void analyzeRuntime(JavaRuntime jr);
}
```