Java Art Chapter 3. Tracing with JPDA

Java Art Chapters 1 and 2 looked at how to convert the source of a Java file into an image, an image that can be executed just like the original program. In this chapter and the next, I'll be explaining how to convert an *executing* program into a visual delight of ever-changing spirals, whorls, pinwheels, stars, tendrils, and other mesmerizing 'psychedelic' patterns (see Figure 1).

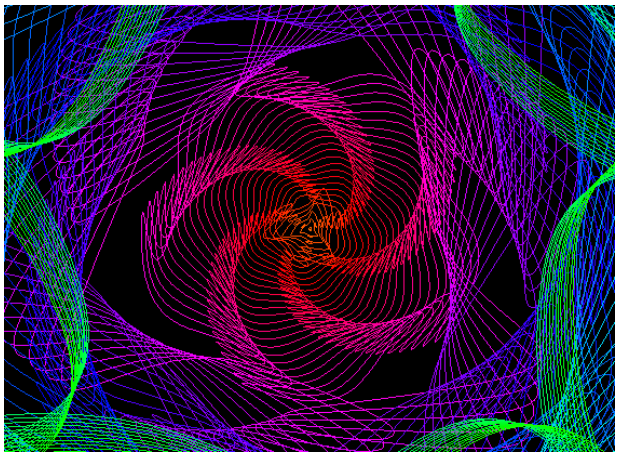

Figure 1. Psychedelic Java

I won't be doing this by adding Java 2D drawing code to the application. Instead, my aim is to visualize a running program without adding to, or changing, its code.

The solution utilizes a combination of the Java Platform Debugger Architecture (JPDA) and Java Sound API (yes, sound), together with an open-source visualizer called Whorld (http://www.churchofeuthanasia.org/whorld), which interprets MIDI music commands as animations like the one in Figure 1.

In this chapter I'll describe how to use JPDA to write a simple tracer which can monitor the execution of a program. In the next chapter I'll augment the tracer with the ability to transmit MIDI control commands to the Whorld visualizer, making it generate the required animation. The basic idea is illustrated by Figure 2.

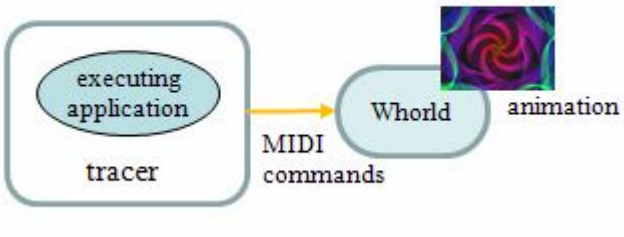

Figure 2. Generating Whorld Animations.

Although this example may seem a little silly, it does have a serious side. The visualization of executing code is a major research area, promising new ways to understand, debug, and monitor programs which are too complex to understand through purely textual means. The approach I use here (tracing with JPDA) has been employed in numerous research projects, although the visualization component has typically been much more complex: e.g. the generation of UML object and sequence diagrams, statecharts, cone trees, and call trees.

Research tools of this type include JIVE (http://www.cs.brown.edu/~spr/research/vizjive.html), JOVE (http://www.cs.brown.edu/~spr/research/visjove.html), JavaVis, JInsight and JinsightLive (http://www.alphaworks.ibm.com/tech/jinsightlive), and Jacot.

Many of these visualization techniques are starting to appear in IDEs, such as Eclipse, JGrasp (http://www.jgrasp.org/), BlueJ (http://www.bluej.org/), and NetBeans.

1. An Overview of JPDA

JPDA is a collection of APIs for the debugging, profiling, or tracing of Java code. It consists of three interfaces which fit together as shown in Figure 3.

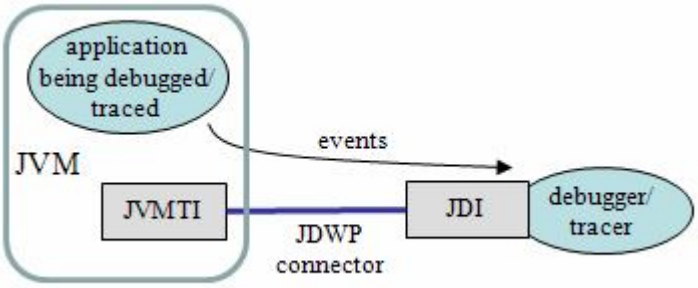

Figure 3. A JPDA Debugger/Tracer

The Java Virtual Machine Tools Interface (JVMTI) defines the JVM's debugging services, while the Java Debug Wire Protocol (JDWP) specifies the communication link between the JVM and the debugger/tracer. Fortunately, the person coding the debugger usually only needs to grapple with the Java Debug Interface (JDI), which lets the JVM's internals be examined as the application executes.

This separation of the JPDA into three parts allows the debugger/tracer to work at a distance – outside the JVM, or even across a network. There are a number of different kinds of connectors, the simplest being a 'launching' connector (which I'll be using) that also starts the JVM.

Once the JDI has established a connection between the debugger and the JVM, it must specify the events that it wants to receive. These may relate to the starting and stopping of threads, the loading/unloading of classes, object state changes, method entry/exit, code execution, or JVM state changes (i.e. when it starts, dies, or is disconnected).

Typically the debugger uses the arrival of an event as an opportunity to examine the application's execution state inside the JVM, which is illustrated in Figure 4.

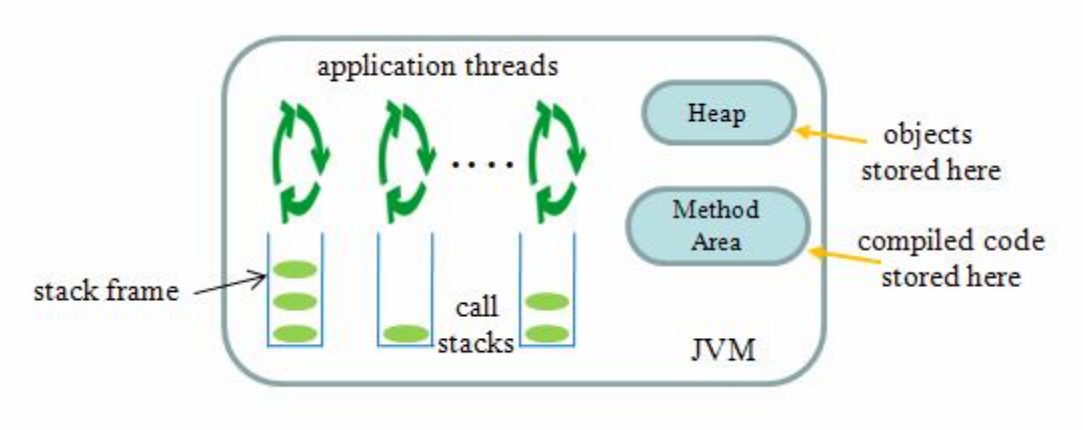

Figure 4. A Simplified View of the JVM.

Figure 4 simplifies matters considerably, but shows most of the JVM's runtime data structures. For more details, a useful textbook is *The Java Virtual Machine Specification*, 2nd Edition, by Tim Lindholm and Frank Yellin, which is available online at http://java.sun.com/docs/books/jvms/.

An application is made up of threads (some created by the application itself, and others by the JVM, perhaps for the GUI or the OS). Each thread has its own call stack, which stores frames for the methods being used by the thread.

A new frame is pushed onto the stack each time a method is invoked, and popped when the method completes. Each frame contains an array of local variables, an operand stack, and a reference to the JVM's constant pool (which isn't shown in Figure 4).

The heap, which is shared by all the threads, stores the objects used in the application. Java's automatic garbage collection algorithms regularly sweep the heap deleting unused objects.

The global method area stores compiled code and data, such as the runtime constant pool, field and method data, and the code for methods and constructors.

The JDI supports a wide range of classes for examining the application state inside the JVM, and for setting breakpoints, monitoring the change of specific fields, and suspending/resuming threads. I'll consider most of these topics in this chapter, the main exception being breakpoints which I don't need in my tracer.

The main JPDA website is at http://java.sun.com/javase/technologies/core/toolsapis/jpda/, which includes links to forums (http://forum.java.sun.com/category.jspa?categoryID=40) and the Java SE documentation on the JPDA (http://java.sun.com/javase/6/docs/technotes/guides/jpda/). Most JPDA developers only utilize the JDI API, which is listed at http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/.

2. Using the Simple Tracer

As an example of what my tracer can do, consider the following TestStack class:

```
public class TestStack
{
  public static void main(String args[])
  {
    Stack stk = new Stack();
    stk.push(42);
    stk.push(17);
    stk.pop();
    System.out.println("Top value is " + stk.topOf());
  }
} // end of TestStack.java
```

A Stack object is initialized, and values pushed and popped.

The Stack class uses an array to implement the stack's internal data structure. A fragment of the code gives the general idea:

```
public class Stack
{
  private int store[];
  private int max_len, top;

  public Stack()
  { store = new int[15];      // default size
    max_len = store.length-1;
    top = -1;
  }

  public boolean push(int number)
  { if (top == max_len)
      return false;
    top++;
    store[top] = number;
    return true;
  }

  // method code for pop(), topOf(), isEmpty() ...

}  // end of Stack class
```

The initial output from the tracer when it's monitoring TestStack and Stack is shown below:

```
> java -cp "C:\Program Files\Java\jdk1.6.0_10\lib\tools.jar;."
                                              SimpleTrace TestStack
-- VM Started --
main thread started
TestStack.java added to listings
loaded class: TestStack from TestStack.java - fields=0, methods=2
  method names:
    | <init>()
    | main()
```



```
entered TestStack.main()
TestStack.java: 10.         Stack stk = new Stack();
  locals:
    | args = instance of java.lang.String[0] (id=61)
Stack.java added to listings
loaded class: Stack from Stack.java - fields=3, methods=5
  method names:
    | <init>()
    | push()
    | pop()
    | topOf()
    | isEmpty()

entered Stack constructor
Stack.java: 13.   {
  object: instance of Stack(id=64)
  fields:
    | store = null
    | max_len = 0
    | top = 0
Stack.java: 14.     store = new int[15];        // default size
    > store = instance of int[15] (id=66)
Stack.java: 15.     max_len = store.length-1;
    > max_len = 14
Stack.java: 16.     top = -1;
    > top = -1
Stack.java: 17.   }
exiting Stack constructor

TestStack.java: 10.         Stack stk = new Stack();
TestStack.java: 11.         stk.push(42);    :
    :  // more output
```

The output includes the method names in TestStack and Stack, the fields in the created Stack object, and the display of executing lines. If a line changes an object's field, then the new value is printed out.

The call to SimpleTrace includes the JRE's tools.jar in the classpath since that's where the JDI classes are located. It's also necessary to add tools.jar to the classpath for javac.exe when compiling the tracer.

3. Overview of the Tracer

The UML class diagrams for the tracer appear in Figure 5 (only public methods are shown).

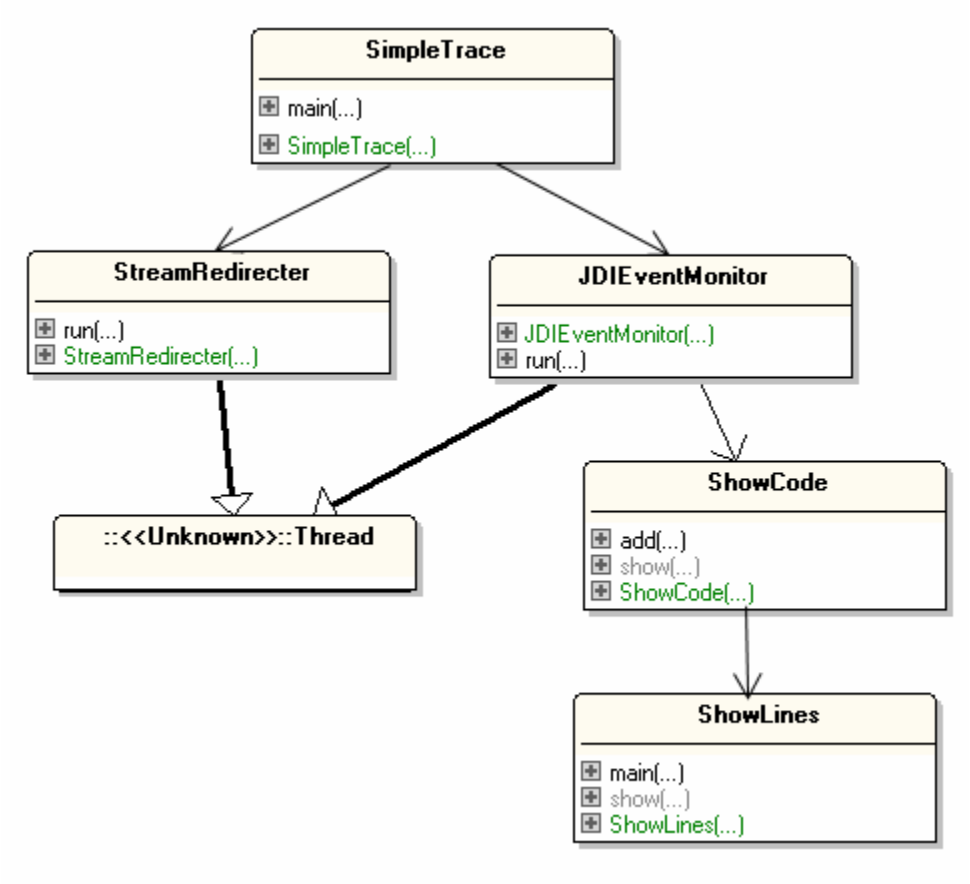

Figure 5. Class Diagrams for the Tracer.

SimpleTrace sets up the command-line launching connection which starts the JVM and creates a local link with the JVM on the same machine. It passes the application's name and input arguments over to the JVM, and employs the StreamRedirecter class to redirect the JVM's output and error streams to stdout and stderr.

The tracing work is carried out by JDIEventMonitor, which starts by specifying the events it is interested in (e.g. class loading/unloading, thread creation/deletion, method entry/exit), and then monitors the incoming JDI events.

This tracer is essentially a simplified version of the Trace example included in the demo/jpda/examples.jar file in the JDK. SimpleTrace is a variant of Trace.java, StreamRedirecter is a slightly changed StreamRedirectThread.java, and JDIEventMonitor is a modified EventThread.java.

One new feature of JDIEventMonitor is its displaying of more details about object fields and local variables. The tracer also includes two new classes, ShowCode and ShowLines, whose job is to list the source code for an executing line.

The other examples in demo/jpda/examples.jar are the jdb debugger and the prototype javadt GUI debugger.

4. Initializing the Tracer

SimpleTrace creates a launching connection to the JVM, requests various events, and starts monitoring the JVM for those events:

```
public SimpleTrace(String[] args)
{
  VirtualMachine vm = launchConnect(args);
  monitorJVM(vm);
}
```

launchConnect() sets up the launching connection, which involves finding a suitable connector, and passing the tracer's input arguments over to the JVM.

```
private VirtualMachine launchConnect(String[] args)
{
  VirtualMachine vm = null;
  LaunchingConnector conn = getCommandLineConnector();
  Map<String,Connector.Argument> connArgs = setMainArgs(conn, args);

  try {
    vm = conn.launch(connArgs);   // launch the JVM and connect to it
    }
  catch (IOException e) {
    throw new Error("Unable to launch JVM: " + e);
  }
  catch (IllegalConnectorArgumentsException e) {
    throw new Error("Internal error: " + e);
  }
  catch (VMStartException e) {
    throw new Error("JVM failed to start: " + e);
  }
  return vm;
}  // end of launchConnect()
```

getCommandLineConnector() searches for a command-line launching connector in the JVM's supported connectors by looking for the name "com.sun.jdi.CommandLineLaunch".

```
private LaunchingConnector getCommandLineConnector()
{
  List<Connector> conns =
      Bootstrap.virtualMachineManager().allConnectors();
  for (Connector conn: conns) {
    if (conn.name().equals("com.sun.jdi.CommandLineLaunch"))
      return (LaunchingConnector) conn;
  }
  throw new Error("No launching connector found");
}
```

setMainArgs() assigns the tracer's input arguments to the connector's "main" field, which tells the JVM which class to invoke, and with what arguments.

```
private Map<String,Connector.Argument> setMainArgs(
                        LaunchingConnector conn, String[] args)
{
```



```
  // get connector field for program's main() method
  Map<String,Connector.Argument> connArgs = conn.defaultArguments();
  Connector.Argument mArgs =
           (Connector.Argument) connArgs.get("main");
  if (mArgs == null)
    throw new Error("Bad launching connector");

  // concatenate all tracer's input args into a single string
  StringBuffer sb = new StringBuffer();
  for (int i=0; i < args.length; i++)
    sb.append(args[i] + " ");

  mArgs.setValue(sb.toString());  // assign args to main field
  return connArgs;
}  // end of setMainArgs()
```

For example, if SimpleTrace is called like so:

```
java -cp "C:\Program Files\Java\jdk1.6.0_10\lib\tools.jar;."
                                        SimpleTrace Foo 10
```

then the JVM receives a "main" field containing "Foo 10", which tells it to invoke the Foo class with the input argument 10.

Monitoring the JVM

The monitoring of the JVM is handed onto an instance of JDIEventMonitor. Also, SimpleTrace utilizes StreamRedirecter instances to re-route the output and error streams coming from the JVM to stdout and stderr.

```
private void monitorJVM(VirtualMachine vm)
{
  // start JDI event handler which displays trace info
  JDIEventMonitor watcher = new JDIEventMonitor(vm);
  watcher.start();

  /* redirect VM's output and error streams
     to the system output and error streams */
  Process process = vm.process();
  Thread errRedirect = new StreamRedirecter("error reader",
                           process.getErrorStream(), System.err);
  Thread outRedirect = new StreamRedirecter("output reader",
                           process.getInputStream(), System.out);
  errRedirect.start();
  outRedirect.start();

  vm.resume();           // start the application

  try {
    watcher.join();      // Wait until JDI watcher terminates
    errRedirect.join();  // make sure all outputs have been forwarded
    outRedirect.join();
  }
  catch (InterruptedException e) { }
}  // end of monitorJVM()
```


Once the JDIEventMonitor watcher has been started, and the streams redirected, then the VirtualMachine.resume() call gets the JVM to start the application.

SimpleTrace finishes by coordinating the termination of the watcher and the two stream redirection threads.

5. Stream Redirection

StreamRedirecter is a thread which keeps copying the data arriving on a specified input stream onto a specified output stream, and terminates when the input stream closes. The code is almost identical to the StreamRedirectThread class included in the demo/jpda/examples.jar file in the JDK.

```
public class StreamRedirecter extends Thread
{
  private static final int BUFFER_SIZE = 2048;
  private final Reader in;
  private final Writer out;


  public StreamRedirecter(String name, InputStream in,
                                          OutputStream out)
  { super(name);
    this.in = new InputStreamReader(in);     // stream to copy from
    this.out = new OutputStreamWriter(out);  // stream to copy to
    setPriority(Thread.MAX_PRIORITY - 1);
  }  // end of StreamRedirecter()


  public void run()
  // copy BUFFER_SIZE chars at a time
  { try {
      char[] cbuf = new char[BUFFER_SIZE];
      int count;
      while ((count = in.read(cbuf, 0, BUFFER_SIZE)) >= 0)
        out.write(cbuf, 0, count);
      out.flush();
    }
    catch (IOException e)
    { System.err.println("StreamRedirecter: " + e);  }
  }  // end of run()

}  // end of StreamRedirecter class
```


6. JDI Event Monitoring

JDIEventMonitor centers around the processing of JDI events added to an event queue by the JVM. However, before that can begin, the JDIEventMonitor constructor calls setEventRequests() to tell the JVM what types of event to send across, how they should be filtered, and how the JVM should suspend while an event is being processed.

```
// globals
// exclude events generated for these classes
private final String[] excludes =
            { "java.*", "javax.*", "sun.*", "com.sun.*"};

private final VirtualMachine vm;   // the JVM


private void setEventRequests()
{
  EventRequestManager mgr = vm.eventRequestManager();

  MethodEntryRequest menr = mgr.createMethodEntryRequest();
  for (int i = 0; i < excludes.length; ++i)  // report method entries
    menr.addClassExclusionFilter(excludes[i]);
  menr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
  menr.enable();

  MethodExitRequest mexr = mgr.createMethodExitRequest();
  for (int i = 0; i < excludes.length; ++i)  // report method exits
    mexr.addClassExclusionFilter(excludes[i]);
  mexr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
  mexr.enable();

  ClassPrepareRequest cpr = mgr.createClassPrepareRequest();
  for (int i = 0; i < excludes.length; ++i)  // report class loads
    cpr.addClassExclusionFilter(excludes[i]);
  cpr.enable();

  ClassUnloadRequest cur = mgr.createClassUnloadRequest();
  for (int i = 0; i < excludes.length; ++i)  // report class unloads
    cur.addClassExclusionFilter(excludes[i]);
  cur.enable();

  ThreadStartRequest tsr = mgr.createThreadStartRequest();
  tsr.enable();                              // report thread starts

  ThreadDeathRequest tdr = mgr.createThreadDeathRequest();
  tdr.enable();                              // report thread deaths
}  // end of setEventRequests()
```

setEventRequests() asks for six types of events (method entry, method exit, class preparation (similar to class loading), class unloading, thread creation, and thread death), which are defined as subclasses of com.sun.jdi.request.EventRequest, and registered with the JVM by the JDI EventRequestManager.

Each event request can also include filters to reduce the amount of event traffic sent from the JVM. My code uses class filters to discard events unrelated to the application classes. There are also filters based on threads, objects, and event counts.

The other task carried out by setEventRequest() is to set the threads suspension policy when an event occurs. The default behavior is to suspend all the threads (EventRequest.SUSPEND_ALL), but this is unnecessarily harsh for method execution when only EventRequest.SUSPEND_EVENT_THREAD is needed to suspend the thread employing the method.

setEventRequests() doesn't specify all the events monitored by JDIEventMonitor. As we'll see, single stepping events (for monitoring the execution of lines of code) and modified field events (for watching for object field changes) must be requested while the application is running.

6.1. Processing Incoming Events

JDIEventMonitor's run() method pulls events off the JDI event queue.

```
// globals
private final VirtualMachine vm;   // the JVM
private boolean connected = true;  // connected to VM?


public void run()
{
  EventQueue queue = vm.eventQueue();
  while (connected) {
    try {
      EventSet eventSet = queue.remove();
      for(Event event : eventSet)
        handleEvent(event);
      eventSet.resume();
    }
    catch (InterruptedException e) { }  // Ignore
    catch (VMDisconnectedException discExc) {
      handleDisconnectedException();
      break;
    }
  }
}  // end of run()
```

Each event is passed to handleEvent() for processing, after which any suspended threads are resumed by a call to EventSet.resume().

If the JVM inadvertently disappears, a disconnection exception will be raised, and run() uses handleDisconnectedException() to empty the event queue before terminating.

```
// globals
private final VirtualMachine vm;   // the JVM
private boolean connected = true;  // connected to VM?


private synchronized void handleDisconnectedException()
{
  EventQueue queue = vm.eventQueue();
  while (connected) {
    try {
```



```
      EventSet eventSet = queue.remove();
      for(Event event : eventSet) {
        if (event instanceof VMDeathEvent)
          vmDeathEvent((VMDeathEvent) event);
        else if (event instanceof VMDisconnectEvent)
          vmDisconnectEvent((VMDisconnectEvent) event);
      }
      eventSet.resume();
    }
    catch (InterruptedException e) { }  // ignore
  }
}  // end of handleDisconnectedException()
```

handleDisconnectedException flushes the event queue, dealing only with exit events (VMDeath, VMDisconnect). In a more complicated application, these might trigger file closures, log termination, or socket disconnections. My code only prints messages to stdout, and sets some globals.

```
// globals
private boolean connected = true;  // connected to VM?
private boolean vmDied;            // has VM death occurred?


private void vmDeathEvent(VMDeathEvent event)
// Notification of VM termination
{ vmDied = true;
  System.out.println("-- The application has exited --");
}

private void vmDisconnectEvent(VMDisconnectEvent event)
/* Notification of VM disconnection, either through normal
   termination or because of an exception/error. */
{ connected = false;
  if (!vmDied)
    System.out.println("- The application has been disconnected -");
}
```

When the connected boolean is set to false, the event processing loop will exit.

6.2. Handling a JDI Event

handleEvent() passes an event to the relevant processing method, depending on its type.

```
private void handleEvent(Event event)
{
  // method events
  if (event instanceof MethodEntryEvent)
    methodEntryEvent((MethodEntryEvent) event);
  else if (event instanceof MethodExitEvent)
    methodExitEvent((MethodExitEvent) event);

  // class events
  else if (event instanceof ClassPrepareEvent)
    classPrepareEvent((ClassPrepareEvent) event);
  else if (event instanceof ClassUnloadEvent)
```

```
    classUnloadEvent((ClassUnloadEvent) event);

  // thread events
  else if (event instanceof ThreadStartEvent)
    threadStartEvent((ThreadStartEvent) event);
  else if (event instanceof ThreadDeathEvent)
    threadDeathEvent((ThreadDeathEvent) event);

  // step event -- a line of code is about to be executed
  else if (event instanceof StepEvent)
    stepEvent((StepEvent) event);

  // modified field event  -- a field is about to be changed
  else if (event instanceof ModificationWatchpointEvent)
    fieldWatchEvent((ModificationWatchpointEvent) event);

  // VM events
  else if (event instanceof VMStartEvent)
    vmStartEvent((VMStartEvent) event);
  else if (event instanceof VMDeathEvent)
    vmDeathEvent((VMDeathEvent) event);
  else if (event instanceof VMDisconnectEvent)
    vmDisconnectEvent((VMDisconnectEvent) event);

  else
    throw new Error("Unexpected event type");
}  // end of handleEvent()
```

handleEvent() deals with 11 types of event – six were specified in setEventRequests(), the three JVM events must always be included (VM start-up, death, and disconnection), and two events are requested at runtime (StepEvent and ModificationWatchpointEvent), as explained below.

6.3. Method Event Handling

There are two events related to methods: MethodEntryEvent and MethodExitEvent.

A MethodEntryEvent is sent out by the JVM when a method has just been entered but before any code is executed. A MethodExitEvent occurs after all the code in a method has been executed, but before it returns.

The two handler functions print method-related information by obtaining a com.sun.jdi.Method object from the event. It references the JVM's method area (see Figure 4).

```
private void methodEntryEvent(MethodEntryEvent event)
// entered a method but no code executed yet
{
  Method meth = event.method();
  String className = meth.declaringType().name();

  System.out.println();
  if (meth.isConstructor())
    System.out.println("entered " + className + " constructor");
  else
    System.out.println("entered " + className+"."+meth.name()+"()");
}  // end of methodEntryEvent()
```



```
private void methodExitEvent(MethodExitEvent event)
// all code in method has been executed, and about to return
{
  Method meth = event.method();
  String className = meth.declaringType().name();

  if (meth.isConstructor())
    System.out.println("exiting " + className + " constructor");
  else
    System.out.println("exiting " + className+"."+meth.name()+"()");
  System.out.println();
}  // end of methodExitEvent()
```

The Method object contains the location of the method in the Java program file, its arguments, its return type, local variables, and keywords usage (e.g. whether it is abstract, native, static, synchronized).

A useful thing which isn't available via Method is the method's Java source, since the method area only holds compiled code. In general, the JVM doesn't retain source code, so it's necessary to store that information separately, which is the purpose of the ShowCode and ShowLines classes in this example.

6.4. Class Event Handling

There are two events related to classes: ClassPrepareEvent and ClassUnloadEvent.

A ClassPrepareEvent is sent out when a class is loaded into the JVM and stored in the methods area. A ClassUnloadEvent occurs when a class is unloaded, which typically happens when the application terminates or the JVM exits.

There's not usually a lot to be done at class unloading time, as reflected by the shortness of the method:

```
// global
private boolean vmDied;       // has VM death occurred?


private void classUnloadEvent(ClassUnloadEvent event)
{ if (!vmDied)
    System.out.println("unloaded class: " + event.className());
}
```

When a class is loaded, the event contains a ReferenceType object, which can represent a class, interface, or array type. This rather strange organization is a reflection of how the JVM works, which maintains class data structures for all three.

Via ReferenceType, details can be accessed about a class' fields, methods, its source file, its location inside that file, and super and subclasses (if the ReferenceType actually refers to a real class, in which case it can be cast to com.sun.jdi.ClassType).

```
private void classPrepareEvent(ClassPrepareEvent event)
// a new class has been loaded
{
  ReferenceType ref = event.referenceType();
```

```
  List<Field> fields = ref.fields();
  List<Method> methods = ref.methods();

  String fnm;
  try {
    fnm = ref.sourceName();  // get filename of the class
    showCode.add(fnm);
  }
  catch (AbsentInformationException e)
  {  fnm = "??"; }

  System.out.println("loaded class: " + ref.name()+" from " +fnm +
     " - fields=" + fields.size() + ", methods=" + methods.size() );

  System.out.println("  method names: ");
  for(Method m : methods)
    System.out.println("    | " + m.name() +   "()" );

  setFieldsWatch(fields);
}  // end of classPrepareEvent()
```

ReferenceType.sourceName() will throw an AbsentInformationException if the reference is pointing to an array or a primitive class.

The showCode.add() call passes the class' filename to an instance of ShowCode, which stores the text of that file for later use.

Once a class has been loaded, its fields can have watchpoints attached to them. JDI offers two forms of watchpoints: those which trigger events whenever a field is accessed, and those that only release an event when a field is modified. The latter type is generally more useful, and also means that less information is generated during a trace.

```
private void setFieldsWatch(List<Field> fields)
{
  EventRequestManager mgr = vm.eventRequestManager();

  for (Field field : fields) {
    ModificationWatchpointRequest req =
              mgr.createModificationWatchpointRequest(field);
    for (int i = 0; i < excludes.length; i++)
      req.addClassExclusionFilter(excludes[i]);
    req.setSuspendPolicy(EventRequest.SUSPEND_NONE);
    req.enable();
  }
}  // end of setFieldsWatch()
```

The code in setFieldsWatch() is very similar to that in setEventRequests() – a modification watchpoint is requested for every field, but fields in non-application classes are ignored. Also there's no need to suspend threads while a field is being examined.

6.5. Modified Field Event Handling

After setFieldsWatch() has requested the modification watchpoints, ModificationWatchpointEvents will start appearing on the event queue, and handleEvent() processes each one by calling fieldWatchEvent():

```
private void fieldWatchEvent(ModificationWatchpointEvent event)
{
   Field f = event.field();
   Value value = event.valueToBe(); // value that _will_ be assigned
   System.out.println("    > " + f.name() + " = " + value);
}  // end of fieldWatchEvent()
```

The event contains information on the field being modified, and the value that is about to be assigned to the field.

The com.sun.jdi.Value interface has numerous sub-interfaces for different types and classes. For complex classes (e.g. arrays) it may be necessary to cast the value object to the relevant interface so its methods can be utilized to print its details.

6.6. Thread Event Handling

There are several events related to threads; the two simplest are ThreadStartRequest and ThreadDeathRequest, which are issued when a thread starts and subsequently dies. There are also four events related to when a thread employs a synchronized object (also called a monitor). JDIEventMonitor only deals with thread creation and destruction.

Thread death causes the thread's name to be printed, which is accessed by obtaining a ThreadReference object from the event.

```
private void threadDeathEvent(ThreadDeathEvent event)
// the thread is about to terminate
{
  ThreadReference thr = event.thread();
  if (thr.name().equals("DestroyJavaVM") ||
      thr.name().startsWith("AWT-") )
    return;

  if (thr.threadGroup().name().equals("system"))//ignore sys threads
    return;

  System.out.println(thr.name() + " thread about to die");
}  // end of threadDeathEvent()
```

threadDeathEvent() tries not to report the termination of system threads, by checking the name and thread group name.

Thread creation prints similar details, and also calls setStepping():

```
private void threadStartEvent(ThreadStartEvent event)
// a new thread has started running -- switch on single stepping
{
  ThreadReference thr = event.thread();
```

```
  if (thr.name().equals("Signal Dispatcher") ||
      thr.name().equals("DestroyJavaVM") ||
      thr.name().startsWith("AWT-") )     // AWT threads
    return;

  if (thr.threadGroup().name().equals("system"))// ignore sys threads
    return;

  System.out.println(thr.name() + " thread started");

  setStepping(thr);
} // end of threadStartEvent()
```

setStepping() asks the JVM to issue step events, which are sent out just before each line of code is executed.

```
private void setStepping(ThreadReference thr)
// start single stepping through the new thread
{
  EventRequestManager mgr = vm.eventRequestManager();

  StepRequest sr = mgr.createStepRequest(thr,
            StepRequest.STEP_LINE, StepRequest.STEP_INTO);
  sr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);

  for (int i = 0; i < excludes.length; ++i)
    sr.addClassExclusionFilter(excludes[i]);
  sr.enable();
}  // end of setStepping()
```

There are a few different kinds of step requests, but the most common is a combination of STEP_INTO and STEP_LINE which means that every line in every method will be examined.

6.7. Single Stepping Event Handling

After setStepping() has requested single stepping, StepEvents will start to be added to the event queue, and handleEvent() will process them by calling stepEvent().

An event contains the location of the line that's about to be executed, and a reference to the thread. stepEvent() prints that line (by calling ShowCode.show()), and if it's the first line in a method then it also prints the local variables and the object's fields (by calling printInitialState()).

```
private void stepEvent(StepEvent event)
{
  Location loc = event.location();

  try {   // print the line
    String fnm = loc.sourceName();  // get code's filename
    System.out.println(fnm + ": " +
                    showCode.show(fnm, loc.lineNumber()) );
  }
  catch (AbsentInformationException e) {}

  if (loc.codeIndex() == 0)   // at the start of a method
```



```
    printInitialState( event.thread() );
}  // end of stepEvent()
```

As shown in Figure 4, each thread has a call stack holding frames for the methods it is using. This information is accessible via the ThreadReference object as a list of com.sun.jdi.StackFrame objects. Each StackFrame contains the current state of the method, including its local variables and a reference to the object calling it (i.e. the 'this' object).

The ThreadReference passed to printInitialState() lets it access the call stack of the currently executing application thread. It prints the local variables in the top-most stack frame (i.e. the method which is currently active), and the fields of the 'this' object:

```
private void printInitialState(ThreadReference thr)
{
  // get top-most stack frame
  StackFrame currFrame = null;
  try {
    currFrame = thr.frame(0);
  }
  catch (Exception e) {
    return;
  }

  printLocals(currFrame);

  // print fields for the 'this' object
  ObjectReference objRef = currFrame.thisObject();
  if (objRef != null) {
    System.out.println("  object: " + objRef.toString());
    printFields(objRef);
  }
}  // end of printInitialState()
```

printInitialState() only examines the top-most frame, but any frame in the call stack could be accessed.

Since printInitialState() is only called at the start of a method, printLocals()'s output of local variables will be quite brief.

```
private void printLocals(StackFrame currFrame)
{
  List<LocalVariable> locals = null;
  try {
    locals = currFrame.visibleVariables();
  }
  catch (Exception e) {
    return;
  }

  if (locals.size() == 0)   // no local vars in the list
    return;

  System.out.println("  locals: ");
  for(LocalVariable l : locals)
    System.out.println("    | " + l.name() +
```

```
                              " = " + currFrame.getValue(l) );
}  // end of printLocals()
```

StackFrame.visibleVariables() returns the local variables which are 'visible' from the current line (i.e. those variables that are currently in scope). Since printLocals() is only called at the start of the method, the only visible local variables are the method's input arguments.

printFields() gets the object's field names by looking in the object's class. However, the field *values* are obtained from ObjectReference.

```
private void printFields(ObjectReference objRef)
{
  ReferenceType ref = objRef.referenceType();  // get class of object
  List<Field> fields = null;
  try {
    fields = ref.fields();   // get fields from the class
  }
  catch (ClassNotPreparedException e) {
    return;
  }

  System.out.println("  fields: ");
  for(Field f : fields)   // print field name and value
    System.out.println("    | " + f.name() + " = " +
                                    objRef.getValue(f) );
}  // end of printFields()
```

ReferenceType.fields() only returns the fields declared in the class. There's also an allFields() method which includes the fields declared in its superclasses and interfaces.

7. Displaying Code

As mentioned earlier, JDI doesn't supply source code details since the JVM only stores compiled code in its method area. However, it's not difficult to add a basic code listing capability to the tracer, as illustrated by the ShowCode and ShowLines classes.

A ShowCode object is created in JDIEventMonitor's constructor:

```
// global
private ShowCode showCode;

// in the JDIEventMonitor constructor
showCode = new ShowCode();
```

When a class is first loaded, its filename is passed to ShowCode in classPrepareEvent():

```
// in classPrepareEvent():
```

```
String fnm;
try {
  fnm = ref.sourceName();   // get filename of the class
  showCode.add(fnm);
}
catch (AbsentInformationException e)
{  fnm = "??"; }
```

ShowCode.add() loads the text from the file, ready to be used later for listing source lines. This occurs in stepEvent(), when the code's filename and the current line number are passed to ShowCode.show(); it returns the matching line of code as a string.

```
// in stepEvent()
Location loc = event.location();
try {   // print the line
  String fnm = loc.sourceName();  // get code's filename
  System.out.println(fnm + ": " +
            showCode.show(fnm, loc.lineNumber()) );
}
catch (AbsentInformationException e) {}
```

7.1. Tree-mapping Code

The heart of ShowCode is a TreeMap which maps a filename to a ShowLines object. ShowCode delegates the loading of the file's code, and the searching for a line, to ShowLines.

```
public class ShowCode
{
  private TreeMap<String,ShowLines> listings;

  public ShowCode()
  {  listings = new TreeMap<String,ShowLines>();  }


  public void add(String fnm)
  // add fnm-ShowLines pair to map
  {
    if (listings.containsKey(fnm)) {
      System.out.println(fnm + "already listed");
      return;
    }
    listings.put(fnm, new ShowLines(fnm));
    System.out.println(fnm + " added to listings");
  }  // end of add()


  public String show(String fnm, int lineNum)
  // return the specified line from fnm
  {
    ShowLines lines = listings.get(fnm);
    if (lines == null)
      return (fnm + "not listed");
    return lines.show(lineNum);
  }  // end of show()
```



```
}  // end of ShowCode class
```

7.2. Loading a File of Code

The ShowLines constructor reads in a file line-by-line with a BufferedReader, and stores each line in an ArrayList.

```
// global in ShowLines class
private ArrayList<String> code;


public ShowLines(String fileName)
{
  code = new ArrayList<String>();

  String line = null;
  BufferedReader in = null;

  try {
    in = new BufferedReader(new FileReader(fileName));
    while ((line = in.readLine()) != null)
       code.add(line);
  }
  catch (IOException ex) {
    System.out.println("Problem reading " + fileName);
  }
  finally {
    try {
      if (in != null)
        in.close();
    }
    catch (IOException e) {}
  }
}  // end of showLines()
```

7.3. Finding a Line of Code

A call to ShowLines.show() supplies a line number, which is used to do a lookup in the ShowLines' ArrayList. The only tricky aspect is remembering that line numbers start at 1 while the first string in the ArrayList is at index position 0.

```
public String show(int lineNum)
{
  if (code == null)
     return "No code to show";

  if ((lineNum < 1) || (lineNum > code.size()))
    return "Line no. out of range";

  return ( "" + lineNum + ".\t" + code.get(lineNum-1)); // use num-1
}  // end of show()
```

