# Java Debugger Internals

Java Platform Debugger Architecture(JPDA)

Rajesh Kumar Thandapani

rajesh.kumar.thandapani@oracle.com

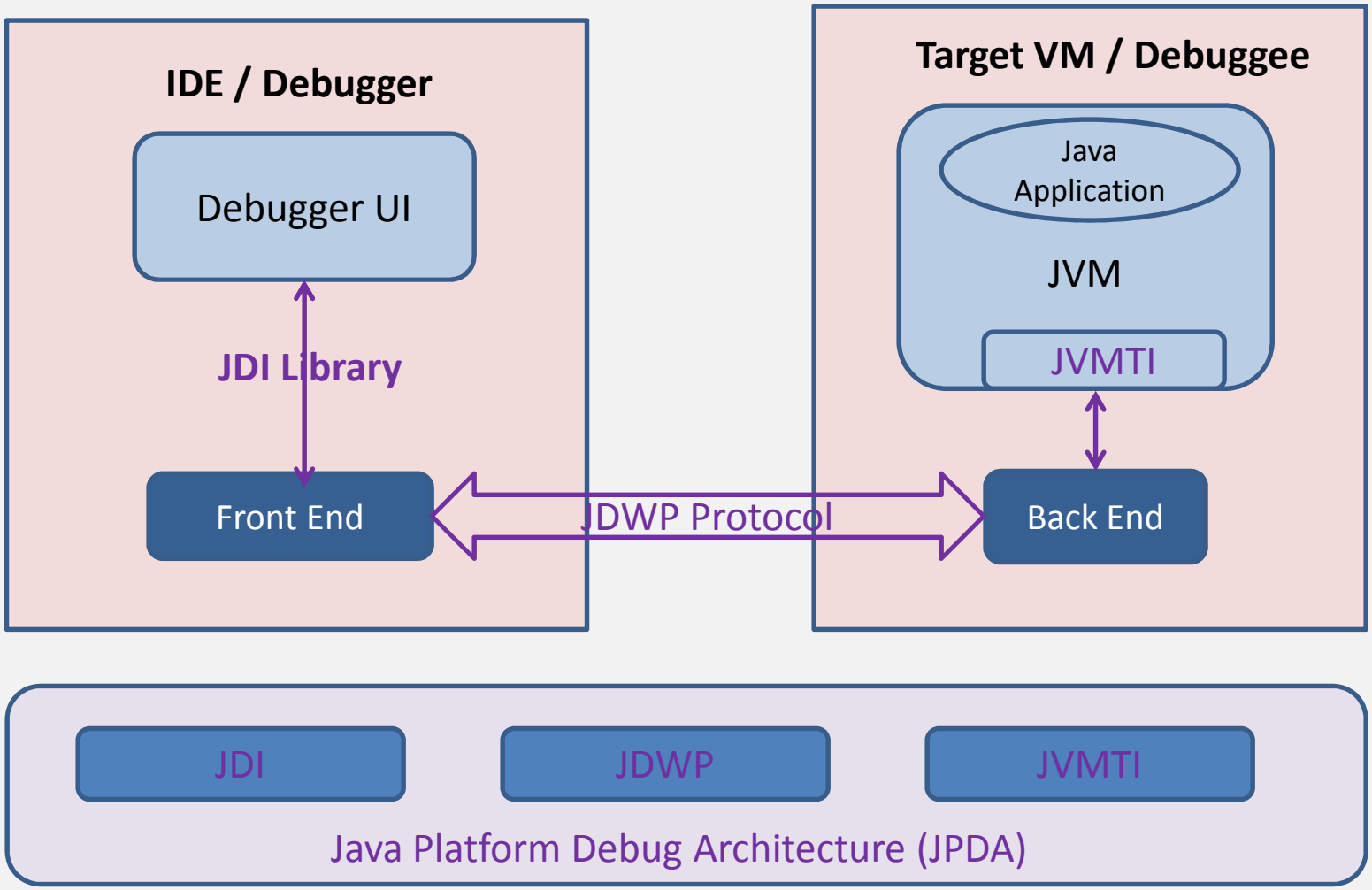
Bangalore Java User Group Meeting (October 2015)

# Agenda

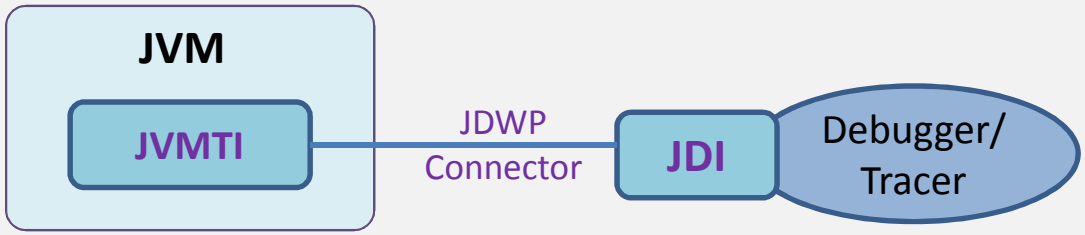
- **Java Platform Debug Architecture (JPDA) Overview**
- Java Debug Interface (JDI)
- Java Debug Wire Protocol (JDWP)
- JVM Tools Interface (JVMTI)
- Debugger Utilities
- QA

# JPDA Overview

The Java Platform Debugger Architecture (JPDA) specification defines set of interfaces for debugging Java application in different Levels of abstractions.



# JPDA Overview: What is JPDA?

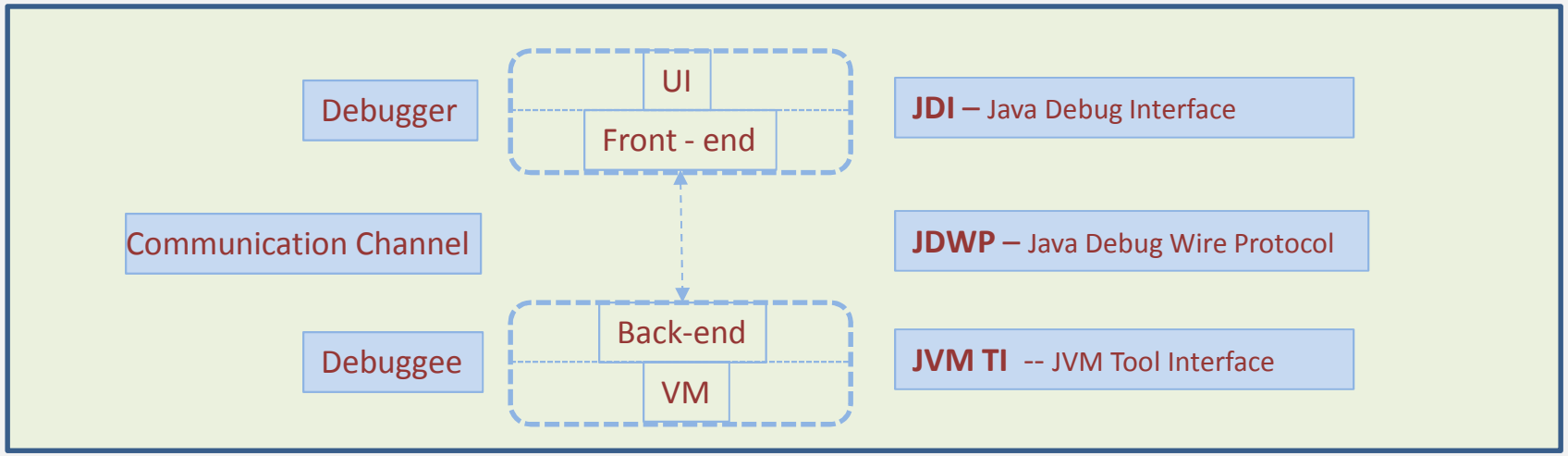



- JDPA is the infrastructure needed to build end- user debugger applications for the Java Platform.

- JPDA is not an application nor a debugging tool but a set of well-designed and implemented interfaces and protocols for debugging.

- communication is connection oriented :
  - one side acts as a server, listening for a connection.
  - another side acts as a client and connects to the server.

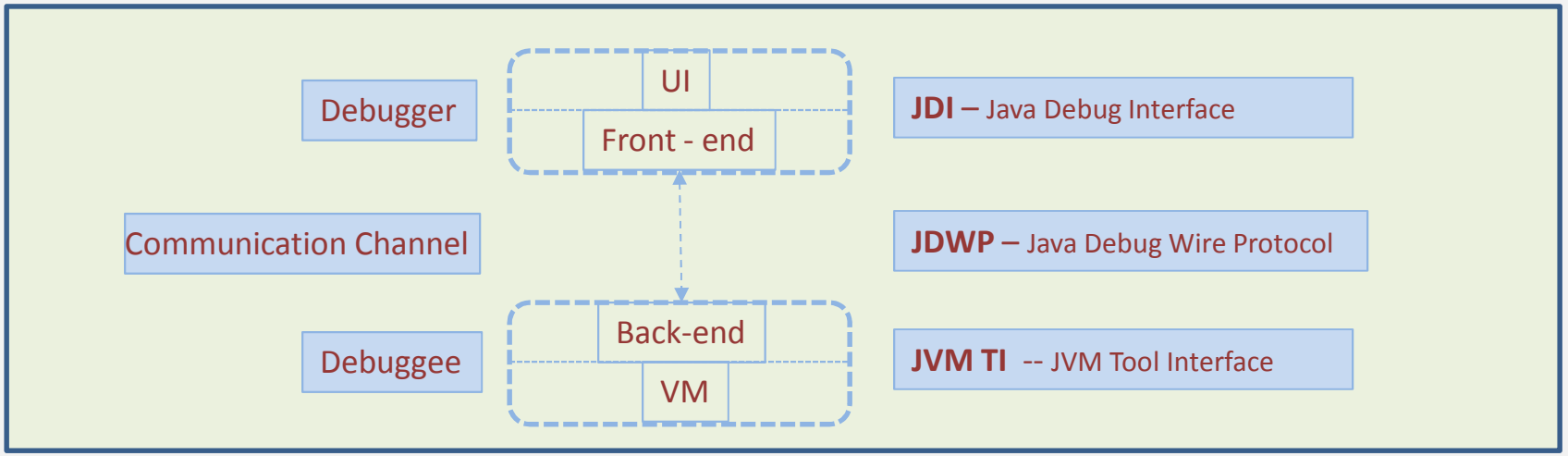
# JPDA Overview: Three layers of JPDA




Three Layers of **JPDA**:

- Debugger :
  - Java Debug Interface (**JDI**) defines the way debugger UI makes debugger calls.
- Communication Channel:
  - Java Debug Wire Protocol (**JDWP**) defines the debugging packets specification.
- Debugger:
  - JVM Tools Interface (**JVMTI**) specifies how the backend interacts with Target VM

# JPDA Overview: Three layers of JPDA



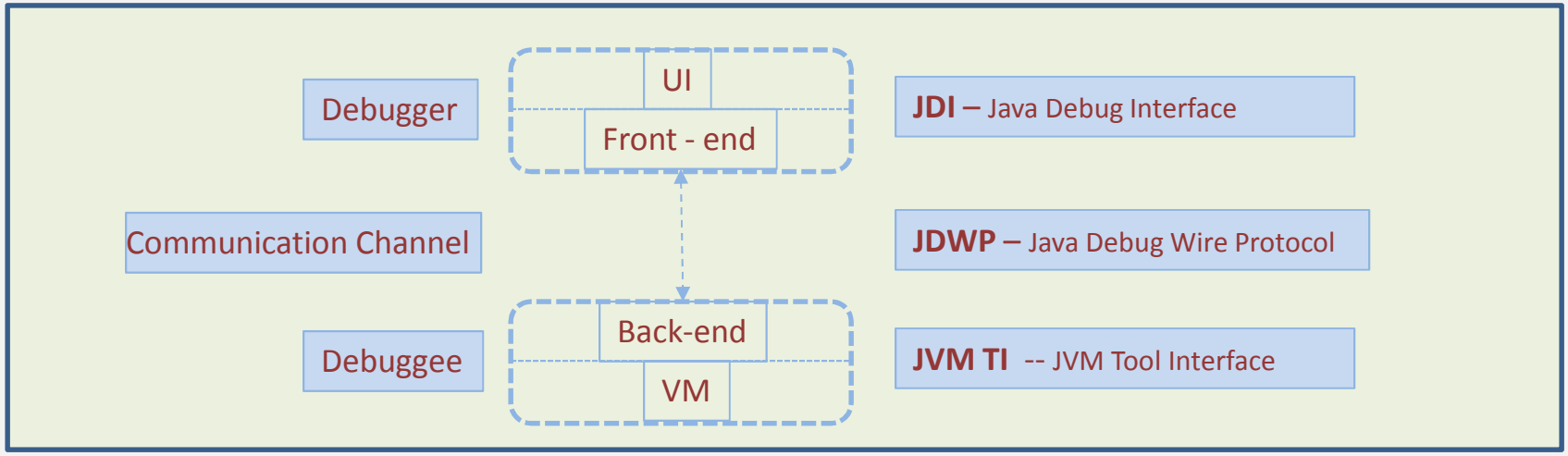

## Why Three Layers needed ?

- Theoretically, the Java Debug Wire Protocol (JDWP) would allow cross-platform remote debugging. But writing directly to **JDWP** is a very low-level protocol and will take lot of effort to form the packets for each request by clients. So The **Java Debug Interface (JDI)** is provided to make interfacing to JDWP easier.

- **JDI** not only formats data to be sent across the wire and parse incoming data, but it also provides queuing, caching, connection initiation and many other services. And all this functionality is available from an easy to use Java programming language interface.

- In an analogous way, **JVM TI** insulates Java virtual machine implementors from the intricacies of the debuggee side of JDWP.

# JPDA Overview: Three layers of JPDA




## Which Interface should be used when?

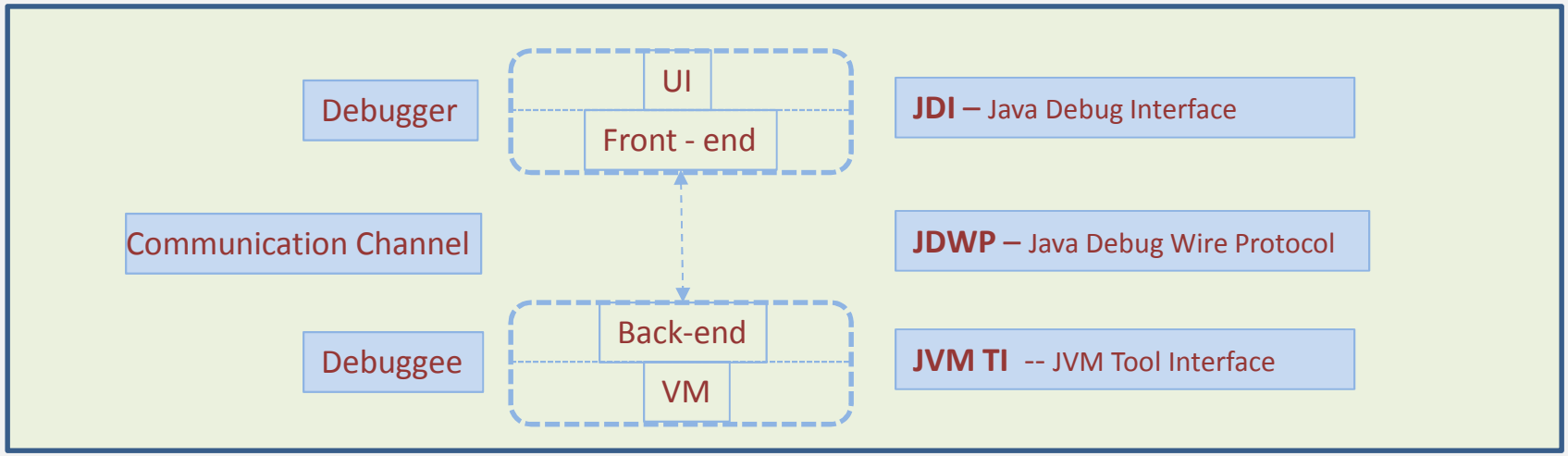
- If you are writing a debugger (or debugger like tool), the easy answer is, use **JDI**.

- You might want to use **JDWP** if your front-end tool is not written in the Java programming language which uses JVM. (like Scala etc)

- You might want to use **JVM TI** if your tool has very specialized functionality, not available in JDWP/JDI, that can only be performed in the debuggee process like Profiling, Heap analysis tools etc.

- If you are writing a Java virtual machine the easy answer is, implement JVM TI.

# JPDA Overview: Three layers of JPDA




**Communication Channel** is the link between the front and back ends of the debugger. It can be thought of as consisting of two mechanisms:

- Connectors
- Transport

**Connector** - A connector is an object that is the means by which a connection is established between the front and back-ends. Connectors are specified and implemented in **JDI.**

**Transport**. A transport is the underlying mechanism used to move bits between the front-end and the back-end. The format and semantics of the serialized bit-stream flowing over the channel is specified by the **JDWP**.

# JPDA Overview: Connectors & Transport

A ***connector*** is a JDI abstraction that is used in establishing a connection between a debugger application (written to the JDI) and a target VM.

**listening**: The front-end listens for an incoming connection from the back-end.

**attaching**: The front-end attaches to an already running back-end.

**launching**: The front-end actually launches the Java process that will run the debuggee code and the back-end.

**Transport:** Specifies different transport mechanisms Socket & Shared Memory.

**Socket :**

- Used for remote/network debugging (dt_socket)
- socket transport addresses have the format "<name>:<port>" where <name>
  - host name and <port> is the socket port number at which it attaches or listens.

***Shared Memory***

- Uses a shared memory region to exchange JDWP packets between the debugger application and the target VM.
- With the shared memory transport, the debugger application and target VM must reside on the same machine.
- The shared memory transport is identified through a unique string, dt_shmem.

# JPDA Overview: Debug Launcher examples

**Java debugger launching/connecting command line arguments**

For JDK5 & above

**-agentlib:jdwp=<sub-options>**

For Prior version of JDK5

**-Xdebug and -Xrunjdwp:<sub-options>**

**<sub-options>:**

- ***transport*** - is a method of communication between a debugger and the virtual machine that is being debugged.
- ***Address*** - when establishing a connection, transport addresses is used to identify the end-point of the connection.
- ***server*** - if the server property is 'y', the application will listen for a debugger to attach; otherwise, it will attach to the debugger at the specified address.
- ***suspend*** - if suspend is 'n', the application will start immediately and will not wait for a debugger to attach to it. If 'y', the application will be suspended until you attach to it.

# JPDA Overview: Debug Launcher examples

**Java debugger launching/connecting command line examples**

- **Socket Listen:**

  -agentlib:jdwp=transport=**dt_socket**,address=localhost:7007,**server=y,**suspend=y

- **Socket Attach:**

  -agentlib:jdwp=transport=**dt_socket**,address=localhost:7007,**server=n,**suspend=y

- **Shared Memory Listen:**

  *-agentlib:jdwp=transport=**dt_shmem**,server=y,suspend=n*

  *prints the chosen "shared memory address" to stdout.*

- **Shared Memory Attach:**

  *-agentlib:jdwp=transport=**dt_shmem**, address=**<mysharedmem>***

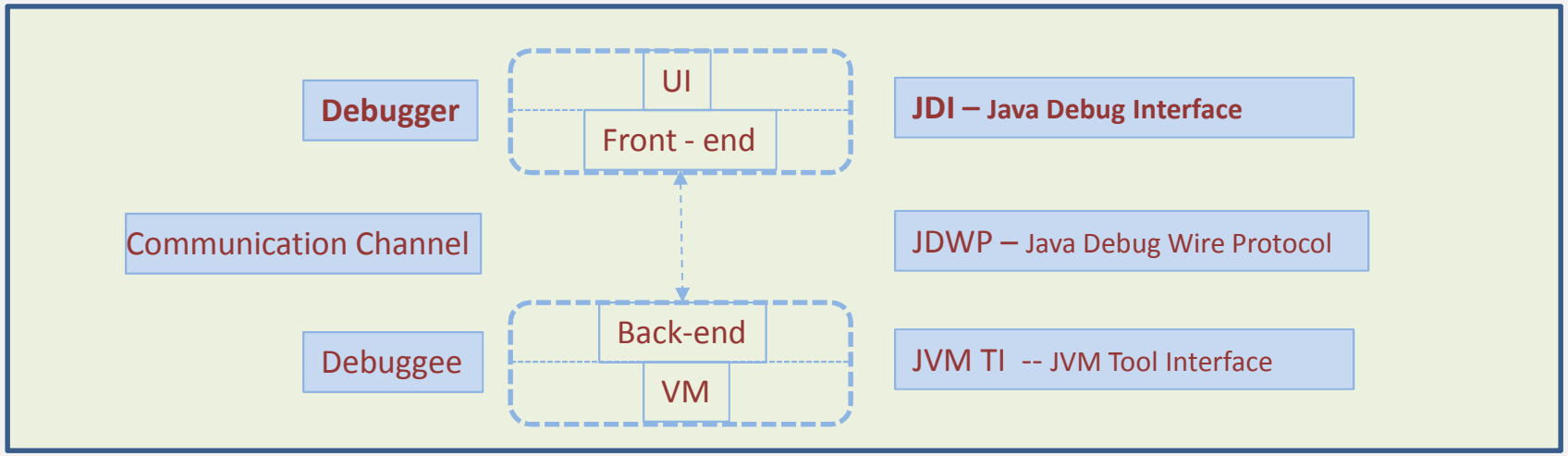
- **Shared Memory Launch:**

  -agentlib:jdwp=transport=**dt_shmem**,server=y,onuncaught=y,**launch**=d:\bin\debugstub.exe

# Agenda

- Java Platform Debug Architecture (JPDA) Overview
- **Java Debug Interface (JDI)**
- Java Debug Wire Protocol (JDWP)
- JVM Tools Interface (JVMTI)
- Debugger Utilities
- QA

# Java Debug Interface (JDI) Overview




**JDI – Java Debug Interface**

- A 100% Java interface implemented by the front-end.
- Defines information and requests at a user code level.
- While debugger implementers could directly use the Java Debug Wire Protocol (JDWP) or Java Virtual Machine Debug Interface (JVM TI), this interface greatly facilitates the integration of debugging capabilities into development environments.
- Since it is a Java interface implemented by the front-end, the command-line parameters depends on the Application.

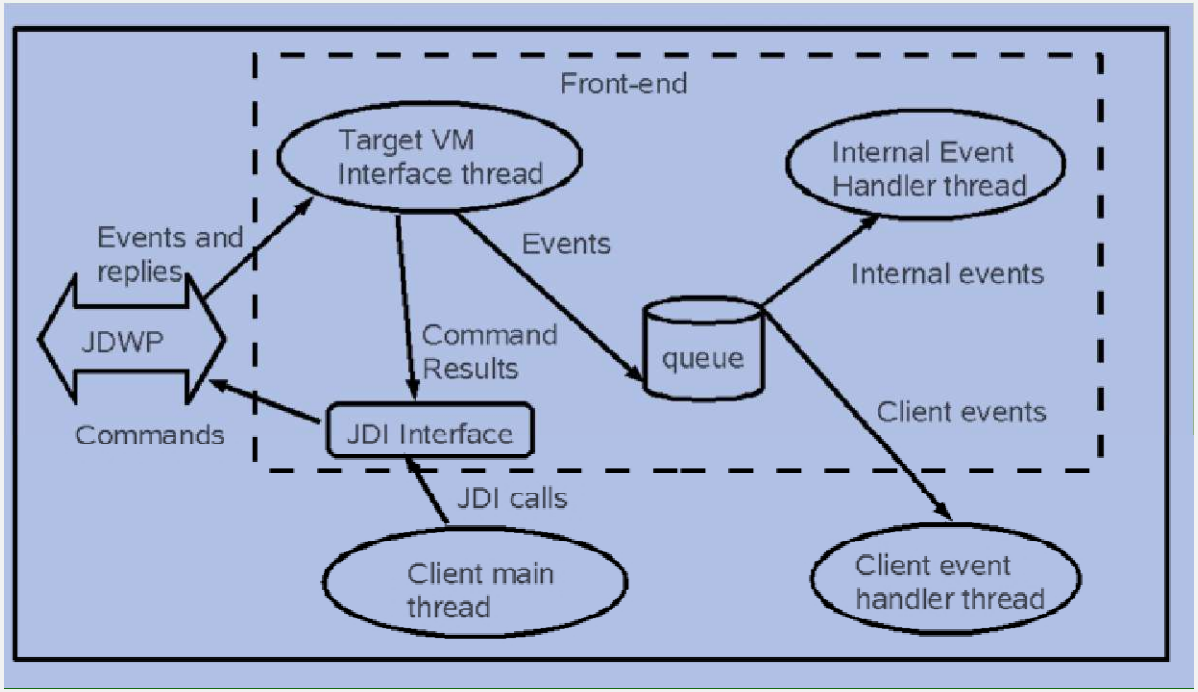
# Java Debug Interface (JDI)

**JDI Library Features:**

- Provides a **high level Java API** providing information useful for debuggers
- Provides Access to a running **virtual machine's state**, Class, Array, Interface, and primitive types, and instances of those types.
- Provides ability explicit control over a virtual machine's execution. ability to **suspend and resume** threads etc.
- Provides ability and to set **breakpoints, watch points**, notification of exceptions, class loading, thread creation...
- Provides ability to inspect a suspended **thread's state, local variables, stack back trace**.
- Provides implementation of various **connectors** for transport suck as listening, attaching and other connectors.

# Java Debug Interface (JDI)

**JDI library framework:**




- provides framework for event handling mechanism using threads and event queues
- provides synchronization methods and throttling for debugger events
- Translates API calls to corresponding JDWP packets and vice versa.

# Java Debug Interface (JDI) : Using JDI API

**JDI library framework usage:**

In order to use the JPDA API, you need to add the JDK **tools.jar** to the class path. It can be found in the JDK lib directory.

**com.sun.jdi** package holds most of the classes required for writing our own JDI.

High level steps of writing our own JDI includes:

- Acquiring the **VirtualMachine** object instance.
- Getting the Connector object from **VirtualMachine** object.
- We can use VirtualMachine's **EventRequestManager** to instruct it to notify us of events. The VirtualMachine **EventQueue** is then used to process the generated events.

Based on our requirements, we can use the relevant functions and get the necessary output.

# Java Debug Interface (JDI): Events Example

**JDI library framework: Method Entry request example:**

```
EventRequestManager em=vm.eventRequestManager();
MethodEntryRequest meR=em.createMethodEntryRequest();
meR.addClassFilter("mypckg.*");
meR.enable();
EventQueue eventQ=vm.eventQueue();
while (running) {
        EventSet eventSet=null;
         eventSet=eventQ.remove();
        EventIterator eventIterator=eventSet.eventIterator();
        while (eventIterator.hasNext()) {
                Event event=eventIterator.nextEvent();
                if (event instanceof MethodEntryEvent) {
                        // process this event
                }
                vm.resume();
        }
 }
```

# Java Debug Interface (JDI): Connectors

**Connector Types:**

A ***connector*** is a JDI abstraction that is used in establishing a connection between a debugger application (written to the JDI) and a target VM.

**listening**: The front-end listens for an incoming connection from the back-end.

**attaching**: The front-end attaches to an already running back-end.

**launching**: The front-end actually launches the Java process that will run the debuggee code and the back-end.

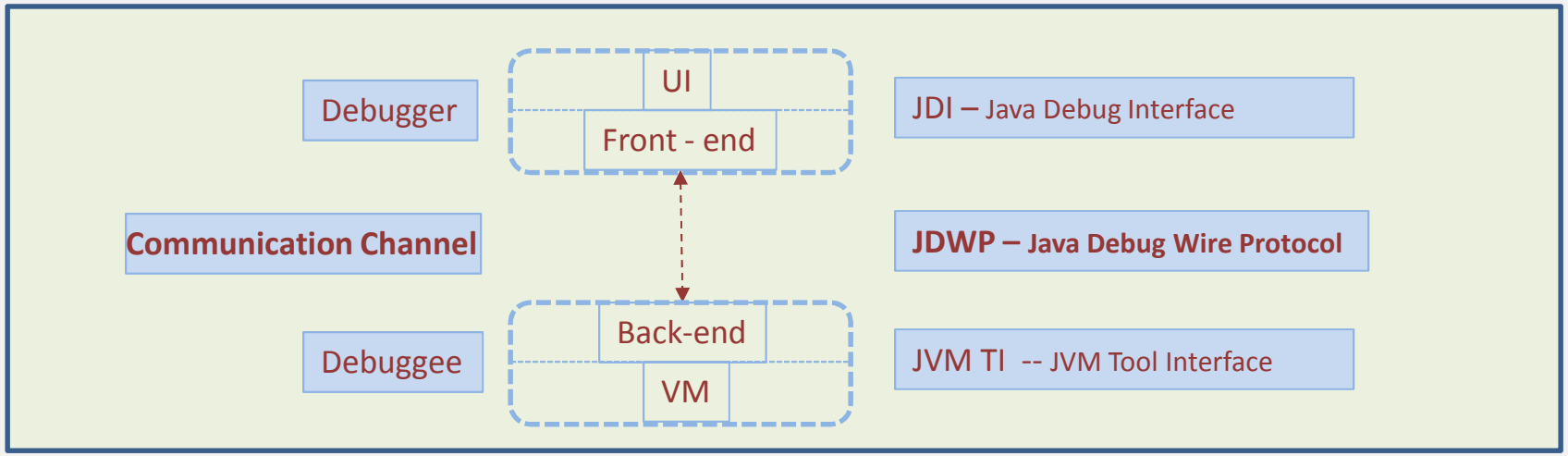
**JDI Connector Examples:**

```
vmm = com.sun.jdi.Bootstrap.virtualMachineManager();}
prm = atconn.defaultArguments();
prm.get("port").setValue(7896)
prm.get("hostname“).setValue("127.0.0.1”)
vmm.attach(prm);
```

# Agenda

- Java Platform Debug Architecture (JPDA) Overview
- Java Debug Interface (JDI)
- **Java Debug Wire Protocol (JDWP)**
- JVM Tools Interface (JVMTI)
- Debugger Utilities
- QA

# Java Debug Wire Protocol (JDWP)




**Java Debug Wire Protocol:** Specification of Debug Transport Packets

- **Defines the format of information and requests transferred between the debuggee process and the debugger front-end.**
- **It does not define the transport mechanism (socket, serial line, shared memory, ...).**
- **The specification of the protocol allows the debuggee and debugger front-end to run under separate VM implementations and/or on separate platforms.**

# Java Debug Wire Protocol (JDWP) : Format

- The Java Debug Wire Protocol (JDWP) is the protocol used for communication between a debugger and the target VM in a different process on the same computer, or on a remote computer.

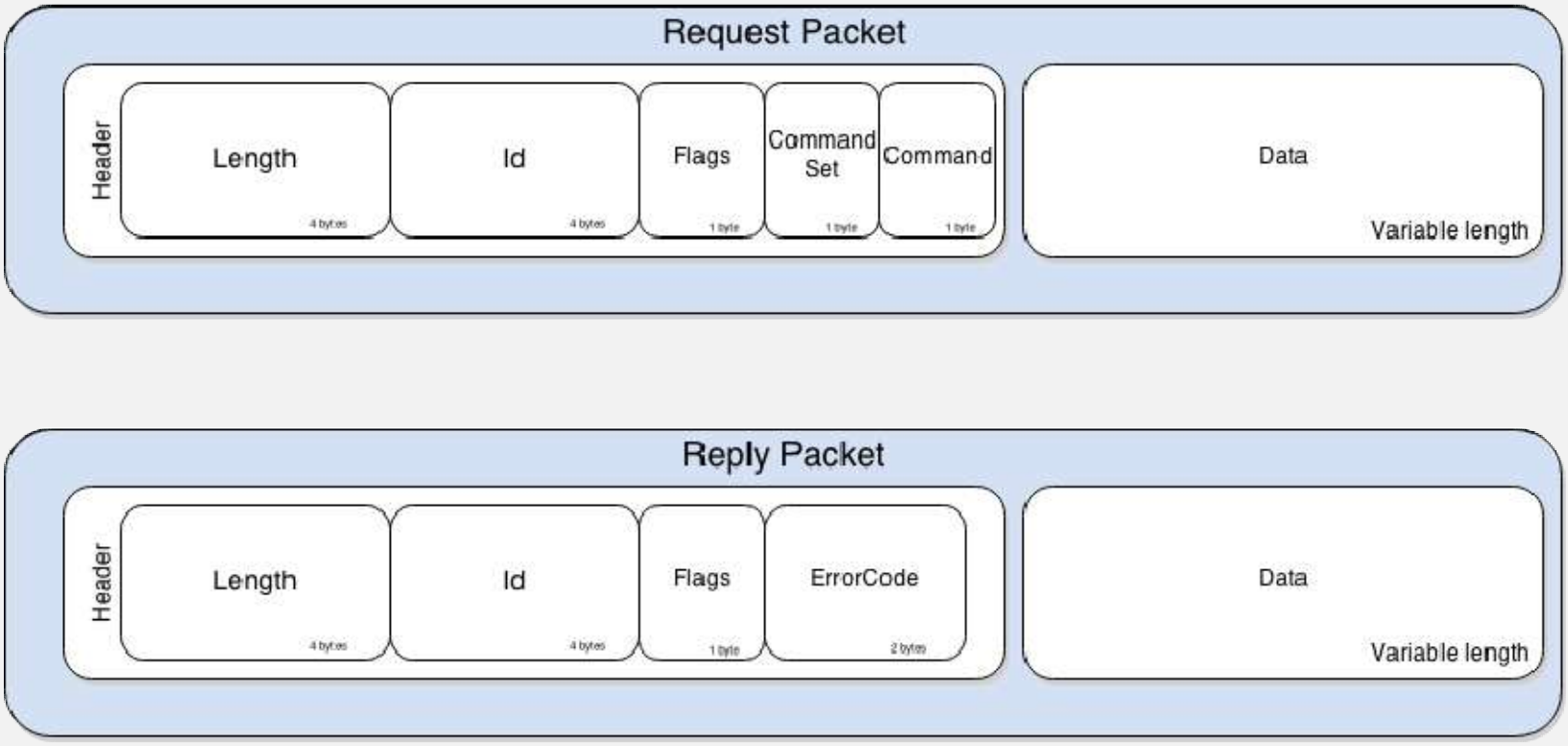
- The JDWP differs from many protocol specifications in that it only details format and layout, not transport.

- The JDWP is designed to be simple enough for easy implementation, yet it is flexible enough for future growth.

- Every Command has a packet format
  - Header
    - length (4 bytes)
    - id (4 bytes)
    - flags (1 byte)
    - command set (1 byte)
    - command (1 byte)
  - data (Variable)

# Java Debug Wire Protocol (JDWP) : Format

**JDWP Packet Formats:**



# Java Debug Wire Protocol (JDWP)

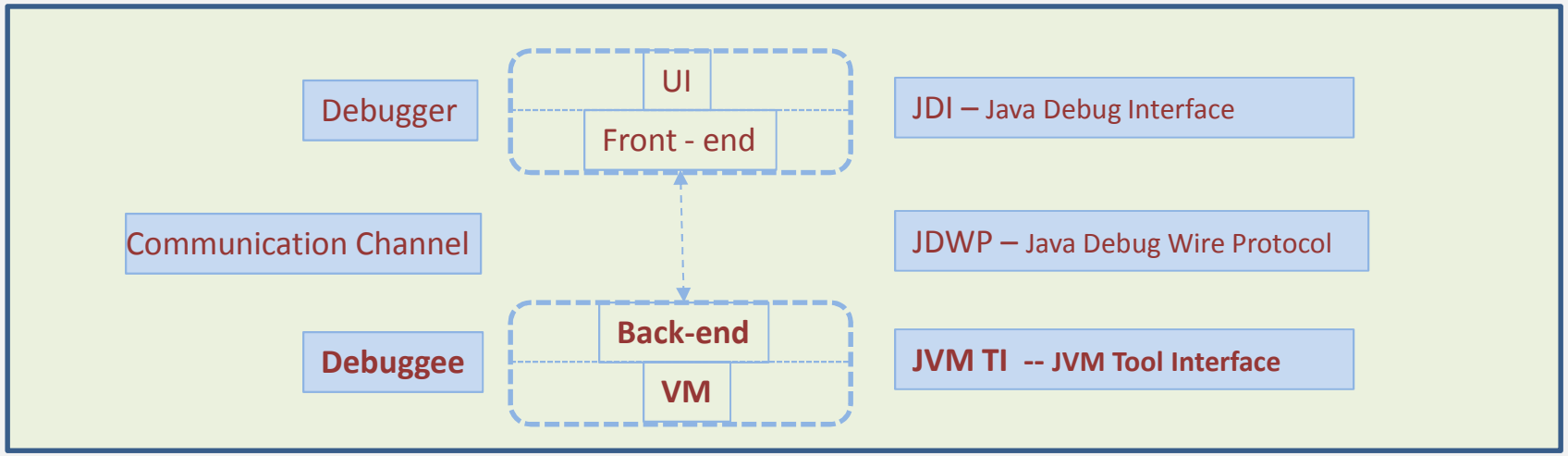
**JDWP Command sets/ Commands:**

| Command Set | Commands |
|---|---|
| Virtual Machine | Version, ClassesBySignature, Suspend, Resume etc |
| Reference Type | Signature, ClassLoader, Fields, Methods etc |
| Class Type | Super Class, Set Values, Invoke Method, NewInstance |
| Array Type | New Instance |
| Interface Type | |
| Method | Line Table, Variable Table, Byte Codes, IsObsolete etc |
| Field | |
| Object Reference | Reference Type, Get Values, Set Values, Monitor Info etc |
| String Reference | Value |
| Thread Reference | Name, Suspend, Resume, Status, Thread Group, Frames etc |
| Thread Group Reference | Name, Parent, Childern |
| Etc | |

# Agenda

- Java Platform Debug Architecture (JPDA) Overview
- Java Debug Interface (JDI)
- Java Debug Wire Protocol (JDWP)
- **JVM Tools Interface (JVMTI)**
- Debugger Utilities
- QA

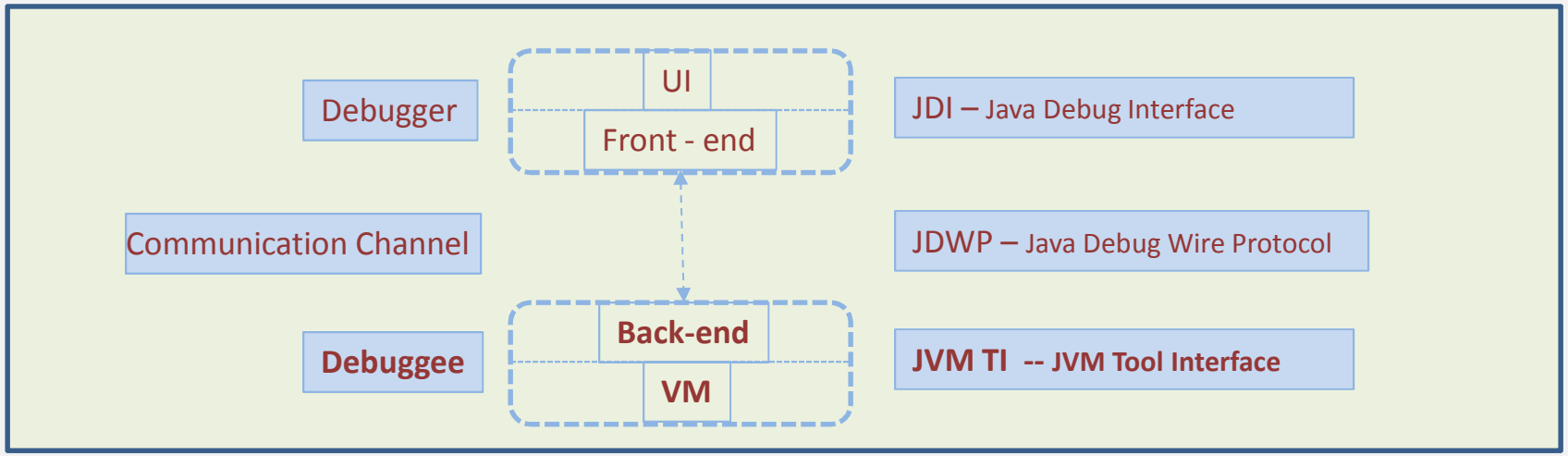
# JVM Tools Interface (JVMTI)




The Debuggee is the process that is being debugged.
It consists of :

- Application being debugged
- Target VM running the application
- Back-end of the Debuggee providing transport services

# JVM Tools Interface (JVMTI)



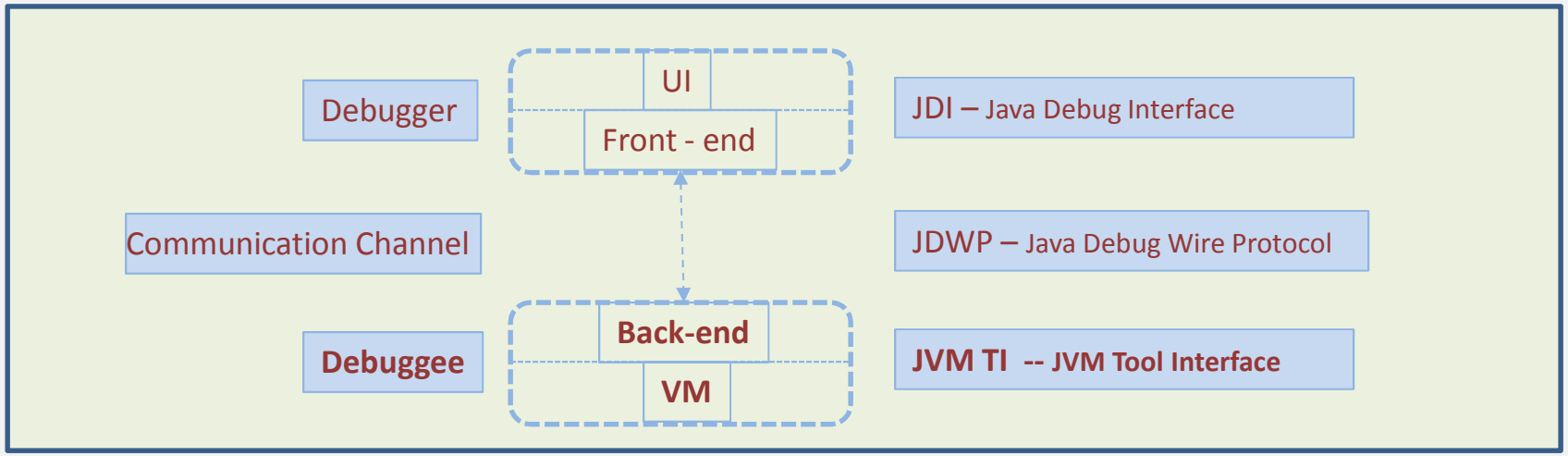

**Java Virtual Machine (VM)**

- Refers to the VM running the Application being debugged.
- VM implements the Java Virtual Machine Debug Interface (JVM TI).

**Back-end**

- Back-end helps in communicating requests from the debugger front-end to the debuggee.
- It also helps in communicating the responses to the requests, including events.
- The back-end communicates with the debuggee VM using the JVM Tool Interface (JVM TI).
- JDK ships with default back end/Agent in JDWP.dll (or) JDWP.so.

# JVM Tools Interface (JVMTI)




**A native interface implemented by the VM. It defines interfaces VM must provide for debugging. It includes:**

- Requests for Information (ex, current heap usage)
- Actions (ex, set break point)
- Notifications (ex, when break point has been hit)

**JVM TI is a two-way interface.**

- A client of JVMTI can be notified of interesting occurrences through events.
- JVM TI can query and control the application through many functions

JVM TI was introduced in J2SE 5.0 and replaced JVMDI and JVM Profiling Interface (JVMPI). JVMDI was removed in Java SE 6 and JVMPI will be removed in Java SE 7.

# JVM Tools Interface (JVMTI) : JVMTI Agents

**JVMTI Back end: Agents**

- Agents can be written in any native language that supports C language calling conventions and C or C++ definitions.

- The function, event, data type, and constant definitions needed for using JVM TI are defined in the include file jvmti.h.

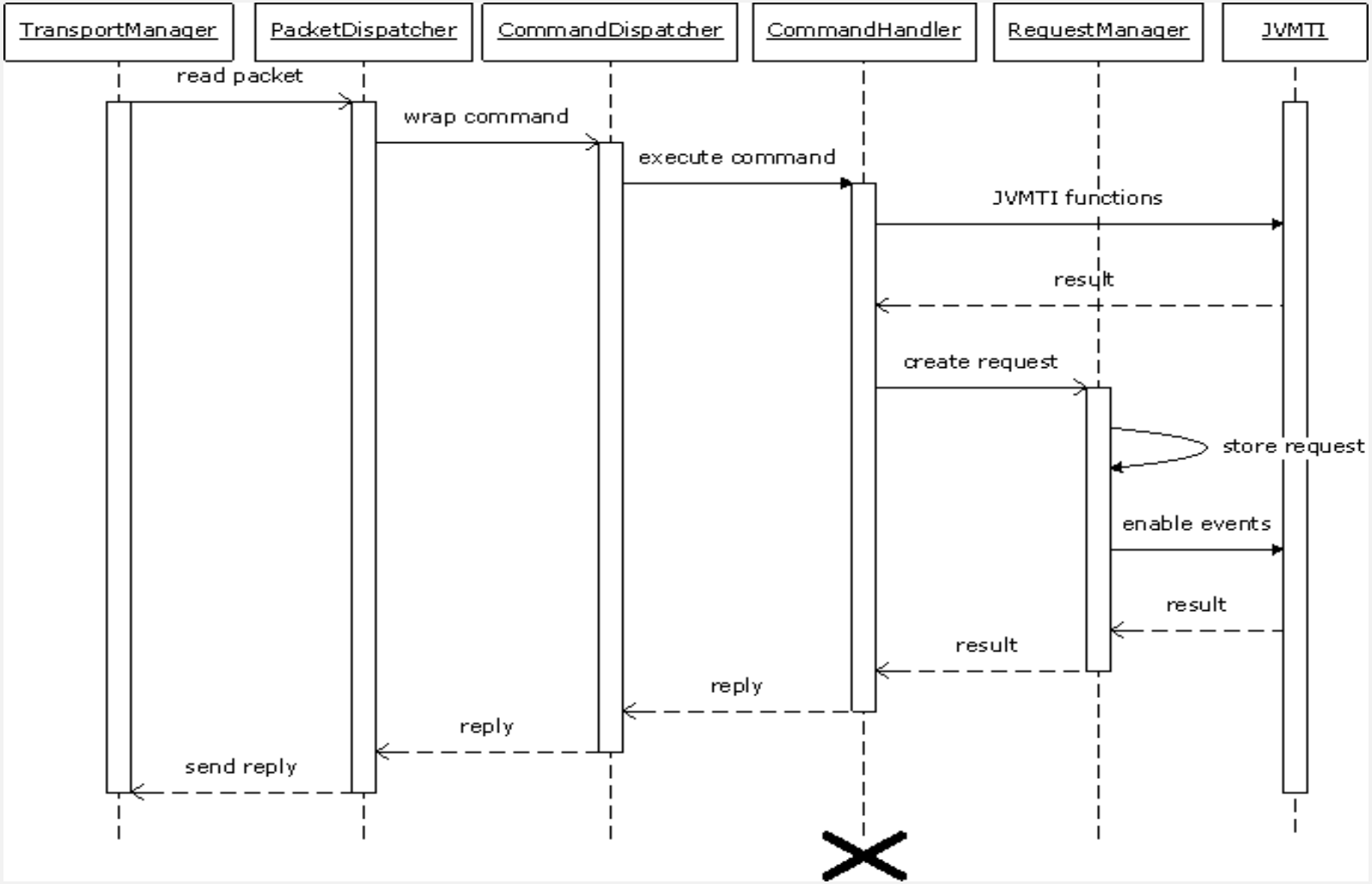
- Agents run in the same process with and communicate directly with the VM executing the application being examined. This communication is through a native interface (JVM TI). The native in-process interface allows maximal control with minimal intrusion on the part of a tool.

- The JVMTI specification supports the use of multiple simultaneous JVMTI agents. Each agent has its own JVMTI environment i.e., JVM TI state is separate for each agent - changes to one environment do not affect the others.

- JDK Ships with default Debug agent in JDWP.dll for debugging.

# JVM Tools Interface (JVMTI) : JVMTI Agents

## JVMTI --- Agent Control Flow



# JVM Tools Interface (JVMTI) : JVMTI Agents

## JVMTI Agent Life Cycle

**Agent startup:**

The VM starts each agent by invoking a start-up function. If the agent is started in the OnLoad phase the function Agent_OnLoad will be invoked. If the agent is started in the live phase the function Agent_OnAttach will be invoked. Exactly one call to a start-up function is made per agent.

UI

**Agent shutdown:**

This function will be called by the VM when the library is about to be unloaded.
***JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm)***

**Loading Agent:**

-***agentlib***:*<agent-lib-name>=<options>* agent-lib-name is the name of the library to load. Filename and the location is dependent on the OS platform. For ex, **-agentlib:myagent**, VM will load myagent.dll file from PATH under windows. myagent.so from LD_LIBRARY_PATH for solaris. Options can also be passed to the agent.

-***agentpath***:*<path-to-agent>=<options>* path-to-agent will have the absolute path to load the library. For ex, -agentpath:d:\myagent\MyAgent.dll will load the **MyAgent .dll** library.

# JVM Tools Interface (JVMTI)

**JVMTI Agent Sample Functions:**

**Sample Functions:**

***Get Thread State*** → Get the state of a thread.
jvmtiError GetThreadState(jvmtiEnv* env, jthread thread, jint* thread_state_ptr)

***Get All Threads*** → Get all live threads
jvmtiError GetAllThreads(jvmtiEnv* env, jint* threads_count_ptr, jthread** threads_ptr)

***Iterate Over Heap***
jvmtiError IterateOverHeap(jvmtiEnv* env, jvmtiHeapObjectFilter object_filter, j
vmtiHeapObjectCallback heap_object_callback, const void* user_data)

# Agenda

- Java Platform Debug Architecture (JPDA) Overview
- Java Debug Interface (JDI)
- Java Debug Wire Protocol (JDWP)
- JVM Tools Interface (JVMTI)
- **Debugger Utilities**
- QA

# Java Debugger Utilities

**Some interesting debugger utilities**

- JDB (Command line interactive Java Debugger)

JDI – Java Debug Interface

- JDB Scripting tool
- JDWP Analyzer
- Writing your own custom debugging program using JDI library

# Java Debugger Utilities : JDB

- Compile the program with the -g option (extra class info).
- Start jdb java debugger
- Set breakpoints, Run program

JDI – Java Debug Interface

- Experiment with debugger commands:
  - list -- Displays the source code of the line and several lines around it
  - locals -- List the values of local variables that are currently in use
  - print <item> -- Display the value of the variable, object, array
  - step -- Executes the next line and stops again
  - cont -- Continues running the program
  - !! -- repeats the previous debugger command.

# Java Debugger Utilities : JDB

- Simple interactive debugger. No IDE needed.
- Generic debugger: works with any JVM that supports JDWP: for example with Java ME, Dalvic etc
- Can be scripted using tools like JDIScript for automating routine debugging tasks.
- Very light weight tool. Can be shipped in embedded target and used to debug programs in-place

# Java Debugger Utilities : JDIScript

- A Simple tool that can be used to script Java Debugging sessions. Jdiscript is an elegant wrapper for a more civilized Java Debugger Interface.

- It allows you to write scripts that use the JDI to control and inspect almost anything happening inside a running JVM

JDI – Java Debug Interface

- Open Source freely available tool: https://github.com/jfager/jdiscript

```
VirtualMachine vm = new VMSocketAttacher(12345).attach();
JDIScript j = new JDIScript(vm);

j.monitorContendedEnterRequest(e -> {
   j.printTrace(e, "ContendedEnter for "+e.monitor());
}).enable();

j.run();
```

# Java Debugger Utilities : JDWP Analyzer

- JDWP (Java Debug Wire Protocol) analysis tool to help you debug your JVM or Debugger .

JDI – Java Debug Interface

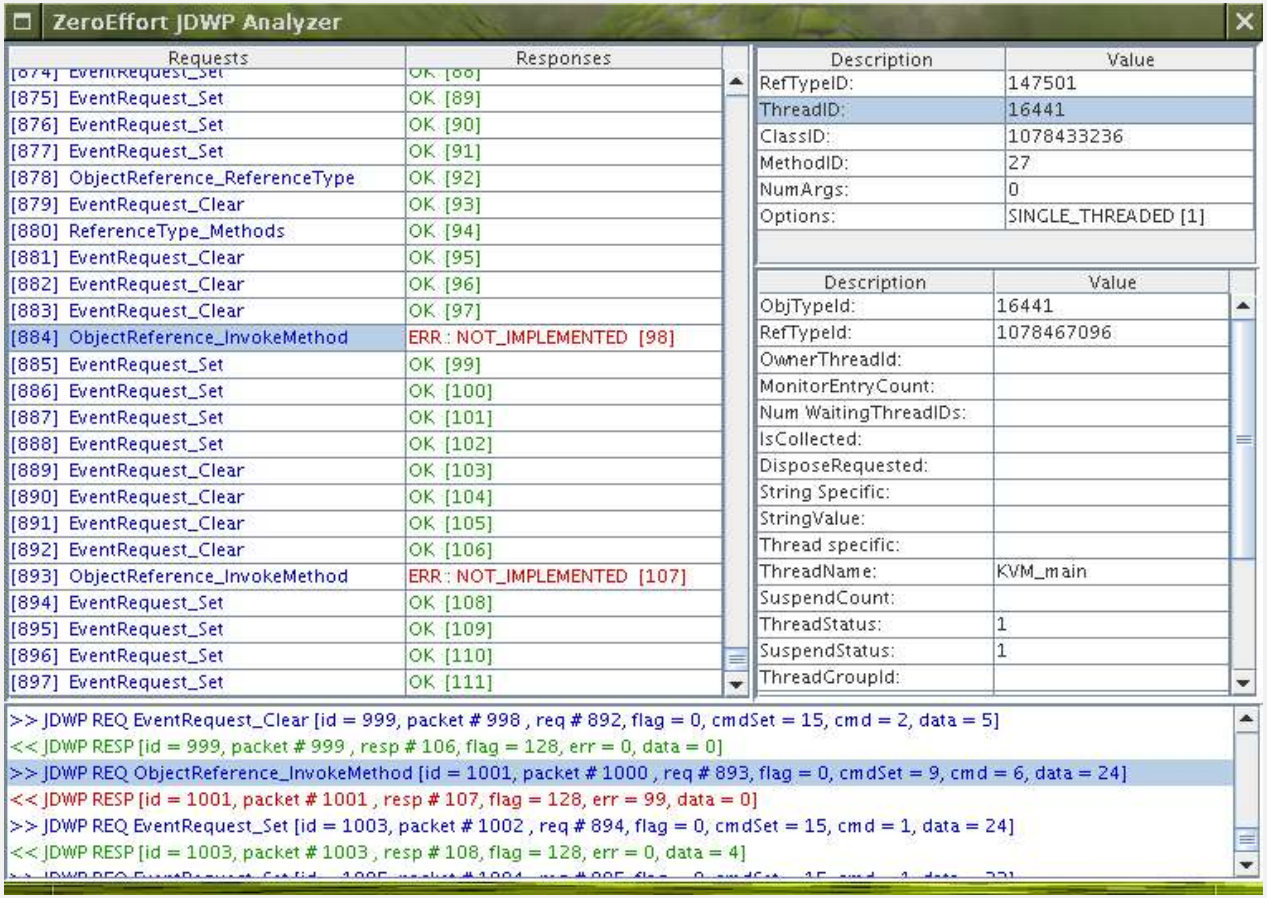
- It works on the JDWP packet format directly so can be used with any JVM or debugger.

- It acts as a transparent agent between the VM and debugger. No change is needed in either VM

- It provides logging of JDWP packets and also provides a GUI to visualize the exact packets exchanged between the VM & debugger.

# Java Debugger Utilities : JDWP Analyzer

Screen Shot:



# Agenda

- Java Platform Debug Architecture (JPDA) Overview
- Java Debug Interface (JDI)
- Java Debug Wire Protocol (JDWP)
- JVM Tools Interface (JVMTI)
- Debugger Utilities
- **QA**

# Java Debugger Internals

Any Questions?