# Java Debug Wire Protocol

Protocol details

## Overview

The Java Debug Wire Protocol (JDWP) is the protocol used for communication between a debugger and the Java virtual machine (VM) which it debugs (hereafter called the target VM). JDWP is optional; it might not be available in some implementations of the JDK. The existence of JDWP can allow the same debugger to work

- 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. A JDWP implementation can be designed to accept different transport mechanisms through a simple API. A particular transport does not necessarily support each of the debugger/target VM combinations listed above.

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

Currently, the JDWP does not specify any mechanism for transport rendezvous or any directory services. This may be changed in the future, but it may be addressed in a separate document.

JDWP is one layer within the Java Platform Debugger Architecture (JPDA). This architecture also contains the higher-level Java Debug Interface (JDI). The JDWP is designed to facilitate efficient use by the JDI; many of its abilities are tailored to that end. The JDI is more appropriate than JDWP for many debugger tools, particularly those written in the Java programming language. For more information on the Java Platform Debugger Architecture, see the Java Platform Debugger Architecture documentation for this release.

## JDWP Start Up

After the transport connection is established and before any packets are sent, a handshake occurs between the two sides of the connection:

The handshake process has the following steps:

- The debugger side sends 14 bytes to the VM side, consisting of the 14 ASCII characters of the string "JDWP-Handshake".
- The VM side replies with the same 14 bytes: JDWP-Handshake

## JDWP Packets

The JDWP is packet based and is not stateful. There are two basic packet types: command packets and reply packets.

Command packets may be sent by either the debugger or the target VM. They are used by the debugger to request information from the target VM, or to control program execution. Command packets are sent by the target VM to notify the debugger of some event in the target VM such as a breakpoint or exception.

A reply packet is sent only in response to a command packet and always provides information success or failure of the command. Reply packets may also carry data requested in the command (for example, the value of a field or variable). Currently, events sent from the target VM do not require a response packet from the debugger.

The JDWP is asynchronous; multiple command packets may be sent before the first reply packet is received.

Command and reply packet headers are equal in size; this is to make transports easier to implement and abstract. The layout of each packet looks like this:

Command Packet

- Header
  - length (4 bytes)
  - id (4 bytes)
  - flags (1 byte)
  - command set (1 byte)
  - command (1 byte)
- data (Variable)

Reply Packet

- Header
  - length (4 bytes)
  - id (4 bytes)
  - flags (1 byte)
  - error code (2 bytes)
- data (Variable)

All fields and data sent via JDWP should be in big-endian format. (See the Java Virtual Machine Specification for the definition of big-endian.) The first three fields are identical in both packet types.

## Command and Reply Packet Fields

### Shared Header Fields

length
: The length field is the size, in bytes, of the entire packet, including the length field. The header size is 11 bytes, so a packet with no data would set this field to 11.

id
: The id field is used to uniquely identify each packet command/reply pair. A reply packet has the same id as the command packet to which it replies. This allows asynchronous commands and replies to be matched. The id field must be unique among all outstanding commands sent from one source. (Outstanding commands originating from the debugger may use the

same id as outstanding commands originating from the target VM.) Other than that, there are no requirements on the allocation of id's.

A simple monotonic counter should be adequate for most implementations. It will allow 2^32 unique outstanding packets and is the simplest implementation alternative.

flags

Flags are used to alter how any command is queued and processed and to tag command packets that originate from the target VM. There is currently one flag bits defined; future versions of the protocol may define additional flags.

`0x80`

Reply packet

The reply bit, when set, indicates that this packet is a reply.

**Command Packet Header Fields**

command set

This field is useful as a means for grouping commands in a meaningful way. The Sun defined command sets are used to group commands by the interfaces they support in the JDI. For example, all commands that support the JDI VirtualMachine interface are grouped in a VirtualMachine command set.

The command set space is roughly divided as follows:

`0 - 63`

Sets of commands sent to the target VM

`64 - 127`

Sets of commands sent to the debugger

`128 - 256`

Vendor-defined commands and extensions.

command

This field identifies a particular command in a command set. This field, together with the command set field, is used to indicate how the command packet should be processed. More succinctly, they tell the receiver what to do. Specific commands are presented later in this document.

**Reply Packet Header Fields**

error code

This field is used to indicate if the command packet that is being replied too was successfully processed. A value of zero indicates success, a non-zero value indicates an error. The error code returned may be specific to each command set/command, but it is often mapped to a JVM TI error code.

**Data**

The data field is unique to each command set/command. It is also different between command and reply packet pairs. For example, a command packet that requests a field value will contain references to the object and field id's for the desired value in its data field. The reply packet's data field will contain the value of the field.

# Detailed Command Information

In general, the data field of a command or reply packet is an abstraction of a group of multiple fields that define the command or reply data. Each subfield of a data field is encoded in big endian (Java) format. The detailed composition of data fields for each command and its reply are described in this section.

There is a small set of common data types that are common to many of the different JDWP commands and replies. They are described below.

| Name | Size | Description |
|---|---|---|
| `byte` | 1 byte | A byte value. |
| `boolean` | 1 byte | A boolean value, encoded as 0 for false and non-zero for true. |
| `int` | 4 bytes | An four-byte integer value. The integer is signed unless explicitly stated to be unsigned. |
| `long` | 8 bytes | An eight-byte integer value. The value is signed unless explicitly stated to be unsigned. |
| `objectID` | Target VM-specific, up to 8 bytes (see below) | Uniquely identifies an object in the target VM. A particular object will be identified by exactly one objectID in JDWP commands and replies throughout its lifetime (or until the objectID is explicitly disposed). An ObjectID is not reused to identify a different object unless it has been explicitly disposed, regardless of whether the referenced object has been garbage collected. An objectID of 0 represents a null object.<br><br>Note that the existence of an object ID does not prevent the garbage collection of the object. Any attempt to access a a garbage collected object with its object ID will result in the INVALID_OBJECT error code. Garbage collection can be disabled with the DisableCollection command, but it is not usually necessary to do so. |
| `tagged-objectID` | size of objectID plus one byte | The first byte is a signature byte which is used to identify the object's type. See JDWP.Tag for the possible values of this byte (note that only object tags, not primitive tags, may be used). It is followed immediately by the objectID itself. |
| `threadID` | same as objectID | Uniquely identifies an object in the target VM that is known to be a thread |
| `threadGroupID` | same as objectID | Uniquely identifies an object in the target VM that is known to be a thread group |
| `stringID` | same as objectID | Uniquely identifies an object in the target VM that is known to be a string object. Note: this is very different from string, which is a value. |
| `classLoaderID` | same as objectID | Uniquely identifies an object in the target VM that is known to be a class loader object |

| Name | Size | Description |
|---|---|---|
| classObjectID | same as objectID | Uniquely identifies an object in the target VM that is known to be a class object. |
| arrayID | same as objectID | Uniquely identifies an object in the target VM that is known to be an array. |
| referenceTypeID | same as objectID | Uniquely identifies a reference type in the target VM. It should not be assumed that for a particular class, the classObjectID and the referenceTypeID are the same. A particular reference type will be identified by exactly one ID in JDWP commands and replies throughout its lifetime A referenceTypeID is not reused to identify a different reference type, regardless of whether the referenced class has been unloaded. |
| classID | same as referenceTypeID | Uniquely identifies a reference type in the target VM that is known to be a class type. |
| interfaceID | same as referenceTypeID | Uniquely identifies a reference type in the target VM that is known to be an interface type. |
| arrayTypeID | same as referenceTypeID | Uniquely identifies a reference type in the target VM that is known to be an array type. |
| methodID | Target VM-specific, up to 8 bytes (see below) | Uniquely identifies a method in some class in the target VM. The methodID must uniquely identify the method within its class/interface or any of its subclasses/subinterfaces/implementors. A methodID is not necessarily unique on its own; it is always paired with a referenceTypeID to uniquely identify one method. The referenceTypeID can identify either the declaring type of the method or a subtype. |
| fieldID | Target VM-specific, up to 8 bytes (see below) | Uniquely identifies a field in some class in the target VM. The fieldID must uniquely identify the field within its class/interface or any of its subclasses/subinterfaces/implementors. A fieldID is not necessarily unique on its own; it is always paired with a referenceTypeID to uniquely identify one field. The referenceTypeID can identify either the declaring type of the field or a subtype. |
| frameID | Target VM-specific, up to 8 bytes (see below) | Uniquely identifies a frame in the target VM. The frameID must uniquely identify the frame within the entire VM (not only within a given thread). The frameID need only be valid during the time its thread is suspended. |
| location | Target VM specific | An executable location. The location is identified by one byte type tag followed by a a classID followed by a methodID followed by an unsigned eight-byte index, which identifies the location within the method. Index values are restricted as follows:<br>• The index of the start location for the method is less than all other locations in the method. |

| Name | Size | Description |
|---|---|---|
| | | • The index of the end location for the method is greater than all other locations in the method.<br>• If a line number table exists for a method, locations that belong to a particular line must fall between the line's location index and the location index of the next line in the table.<br><br>Index values within a method are monotonically increasing from the first executable point in the method to the last. For many implementations, each byte-code instruction in the method has its own index, but this is not required.<br><br>The type tag is necessary to identify whether location's classID identifies a class or an interface. Almost all locations are within classes, but it is possible to have executable code in the static initializer of an interface. |
| `string` | Variable | A UTF-8 encoded string, not zero terminated, preceded by a four-byte integer length. |
| `value` | Variable | A value retrieved from the target VM. The first byte is a signature byte which is used to identify the type. See JDWP.Tag for the possible values of this byte. It is followed immediately by the value itself. This value can be an objectID (see Get ID Sizes) or a primitive value (1 to 8 bytes). More details about each value type can be found in the next table. |
| `untagged-value` | Variable | A `value` as described above without the signature byte. This form is used when the signature information can be determined from context. |
| `arrayregion` | Variable | A compact representation of values used with some array operations. The first byte is a signature byte which is used to identify the type. See JDWP.Tag for the possible values of this byte. Next is a four-byte integer indicating the number of values in the sequence. This is followed by the values themselves: Primitive values are encoded as a sequence of `untagged-values`; Object values are encoded as a sequence of `values`. |

Object ids, reference type ids, field ids, method ids, and frame ids may be sized differently in different target VM implementations. Typically, their sizes correspond to size of the native identifiers used for these items in JNI and JVMDI calls. The maximum size of any of these types is 8 bytes. The "idSizes" command in the VirtualMachine command set is used by the debugger to determine the size of each of these types.

If a debuggee receives a Command Packet with a non-implemented or non-recognized command set or command then it returns a Reply Packet with the error code field set to NOT_IMPLEMENTED (see Error Constants).

Protocol details

- VirtualMachine Command Set (1)
    - Version (1)
    - ClassesBySignature (2)
    - AllClasses (3)
    - AllThreads (4)
    - TopLevelThreadGroups (5)
    - Dispose (6)
    - IDSizes (7)
    - Suspend (8)
    - Resume (9)
    - Exit (10)
    - CreateString (11)
    - Capabilities (12)
    - ClassPaths (13)
    - DisposeObjects (14)
    - HoldEvents (15)
    - ReleaseEvents (16)
    - CapabilitiesNew (17)
    - RedefineClasses (18)
    - SetDefaultStratum (19)
    - AllClassesWithGeneric (20)
    - InstanceCounts (21)

- ReferenceType Command Set (2)
    - Signature (1)
    - ClassLoader (2)
    - Modifiers (3)
    - Fields (4)
    - Methods (5)
    - GetValues (6)
    - SourceFile (7)
    - NestedTypes (8)
    - Status (9)
    - Interfaces (10)
    - ClassObject (11)
    - SourceDebugExtension (12)
    - SignatureWithGeneric (13)
    - FieldsWithGeneric (14)
    - MethodsWithGeneric (15)
    - Instances (16)
    - ClassFileVersion (17)
    - ConstantPool (18)

- ClassType Command Set (3)
    - Superclass (1)
    - SetValues (2)
    - InvokeMethod (3)

    - NewInstance (4)

- ArrayType Command Set (4)
    - NewInstance (1)

- InterfaceType Command Set (5)
    - InvokeMethod (1)

- Method Command Set (6)
    - LineTable (1)
    - VariableTable (2)
    - Bytecodes (3)
    - IsObsolete (4)
    - VariableTableWithGeneric (5)

- Field Command Set (8)
- ObjectReference Command Set (9)
    - ReferenceType (1)
    - GetValues (2)
    - SetValues (3)
    - MonitorInfo (5)
    - InvokeMethod (6)
    - DisableCollection (7)
    - EnableCollection (8)
    - IsCollected (9)
    - ReferringObjects (10)

- StringReference Command Set (10)
    - Value (1)

- ThreadReference Command Set (11)
    - Name (1)
    - Suspend (2)
    - Resume (3)
    - Status (4)
    - ThreadGroup (5)
    - Frames (6)
    - FrameCount (7)
    - OwnedMonitors (8)
    - CurrentContendedMonitor (9)
    - Stop (10)
    - Interrupt (11)
    - SuspendCount (12)
    - OwnedMonitorsStackDepthInfo (13)
    - ForceEarlyReturn (14)

- ThreadGroupReference Command Set (12)
    - Name (1)
    - Parent (2)
    - Children (3)
- ArrayReference Command Set (13)
    - Length (1)
    - GetValues (2)
    - SetValues (3)
- ClassLoaderReference Command Set (14)
    - VisibleClasses (1)
- EventRequest Command Set (15)
    - Set (1)
    - Clear (2)
    - ClearAllBreakpoints (3)
- StackFrame Command Set (16)
    - GetValues (1)
    - SetValues (2)
    - ThisObject (3)
    - PopFrames (4)
- ClassObjectReference Command Set (17)
    - ReflectedType (1)
- Event Command Set (64)
    - Composite (100)
- Error Constants
- EventKind Constants
- ThreadStatus Constants
- SuspendStatus Constants
- ClassStatus Constants
- TypeTag Constants
- Tag Constants
- StepDepth Constants
- StepSize Constants

- SuspendPolicy Constants
- InvokeOptions Constants

## VirtualMachine Command Set (1)

### Version Command (1)

Returns the JDWP version implemented by the target VM. The version string format is implementation dependent.

Out Data

(None)

Reply Data

| | | |
|---|---|---|
| string | *description* | Text information on the VM version |
| int | *jdwpMajor* | Major JDWP Version number |
| int | *jdwpMinor* | Minor JDWP Version number |
| string | *vmVersion* | Target VM JRE version, as in the java.version property |
| string | *vmName* | Target VM name, as in the java.vm.name property |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### ClassesBySignature Command (2)

Returns reference types for all the classes loaded by the target VM which match the given signature. Multple reference types will be returned if two or more class loaders have loaded a class of the same name. The search is confined to loaded classes only; no attempt is made to load a class of the given signature.

Out Data

| | | |
|---|---|---|
| string | *signature* | JNI signature of the class to find (for example, "Ljava/lang/String;"). |

Reply Data

| | | | |
|---|---|---|---|
| int | | *classes* | Number of reference types that follow. |
| Repeated *classes* times: | | | |
| | byte | *refTypeTag* | Kind of following reference type. |
| | referenceTypeID | *typeID* | Matching loaded reference type |

| | | | |
|---|---|---|---|
| | int | *status* | The current class status. |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### AllClasses Command (3)

Returns reference types for all classes currently loaded by the target VM.

Out Data

(None)

Reply Data

| | | | |
|---|---|---|---|
| int | | *classes* | Number of reference types that follow. |
| Repeated *classes* times: | | | |
| | byte | *refTypeTag* | Kind of following reference type. |
| | referenceTypeID | *typeID* | Loaded reference type |
| | string | *signature* | The JNI signature of the loaded reference type |
| | int | *status* | The current class status. |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### AllThreads Command (4)

Returns all threads currently running in the target VM . The returned list contains threads created through java.lang.Thread, all native threads attached to the target VM through JNI, and system threads created by the target VM. Threads that have not yet been started and threads that have completed their execution are not included in the returned list.

Out Data

(None)

Reply Data

| | | | |
|---|---|---|---|
| int | | *threads* | Number of threads that follow. |
| Repeated *threads* times: | | | |
| | threadID | *thread* | A running thread |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### TopLevelThreadGroups Command (5)

Returns all thread groups that do not have a parent. This command may be used as the first step in building a tree (or trees) of the existing thread groups.

Out Data

(None)

Reply Data

| | | | |
|---|---|---|---|
| int | | *groups* | Number of thread groups that follow. |
| Repeated *groups* times: | | | |
| | threadGroupID | *group* | A top level thread group |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### Dispose Command (6)

Invalidates this virtual machine mirror. The communication channel to the target VM is closed, and the target VM prepares to accept another subsequent connection from this debugger or another debugger, including the following tasks:

- All event requests are cancelled.
- All threads suspended by the thread-level resume command or the VM-level resume command are resumed as many times as necessary for them to run.
- Garbage collection is re-enabled in all cases where it was disabled

Any current method invocations executing in the target VM are continued after the disconnection. Upon completion of any such method invocation, the invoking thread continues from the location where it was originally stopped.

Resources originating in this VirtualMachine (ObjectReferences, ReferenceTypes, etc.) will become invalid.

Out Data

(None)

Reply Data

(None)

Error Data

(None)

### IDSizes Command (7)

Returns the sizes of variably-sized data types in the target VM.The returned values indicate the number of bytes used by the identifiers in command and reply packets.

Out Data

(None)

Reply Data

| | | |
|---|---|---|
| int | *fieldIDSize* | fieldID size in bytes |
| int | *methodIDSize* | methodID size in bytes |
| int | *objectIDSize* | objectID size in bytes |
| int | *referenceTypeIDSize* | referenceTypeID size in bytes |
| int | *frameIDSize* | frameID size in bytes |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### Suspend Command (8)

Suspends the execution of the application running in the target VM. All Java threads currently running will be suspended.

Unlike java.lang.Thread.suspend, suspends of both the virtual machine and individual threads are counted. Before a thread will run again, it must be resumed through the VM-level resume command or the thread-level resume command the same number of times it has been suspended.

Out Data

(None)

Reply Data

(None)

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### Resume Command (9)

Resumes execution of the application after the suspend command or an event has stopped it. Suspensions of the Virtual Machine and individual threads are counted. If a particular thread is suspended n times, it must resumed n times before it will continue.

Out Data

(None)

Reply Data

(None)

Error Data

(None)

### Exit Command (10)

Terminates the target VM with the given exit code. On some platforms, the exit code might be truncated, for example, to the low order 8 bits. All ids previously returned from the target VM become invalid. Threads running in the VM are abruptly terminated. A thread death exception is not thrown and finally blocks are not run.

Out Data

| | | |
|---|---|---|
| int | *exitCode* | the exit code |

Reply Data

(None)

Error Data

(None)

### CreateString Command (11)

Creates a new string object in the target VM and returns its id.

Out Data

| | | |
|---|---|---|
| string | *utf* | UTF-8 characters to use in the created string. |

Reply Data

| | | |
|---|---|---|
| stringID | *stringObject* | Created string (instance of java.lang.String) |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### Capabilities Command (12)

Retrieve this VM's capabilities. The capabilities are returned as booleans, each indicating the presence or absence of a capability. The commands associated with each capability will return the NOT_IMPLEMENTED error if the cabability is not available.

Out Data

(None)

Reply Data

| | | |
|---|---|---|
| boolean | *canWatchFieldModification* | Can the VM watch field modification, and therefore can it send the Modification Watchpoint Event? |

| | | |
|---|---|---|
| boolean | *canWatchFieldAccess* | Can the VM watch field access, and therefore can it send the Access Watchpoint Event? |
| boolean | *canGetBytecodes* | Can the VM get the bytecodes of a given method? |
| boolean | *canGetSyntheticAttribute* | Can the VM determine whether a field or method is synthetic? (that is, can the VM determine if the method or the field was invented by the compiler?) |
| boolean | *canGetOwnedMonitorInfo* | Can the VM get the owned monitors information for a thread? |
| boolean | *canGetCurrentContendedMonitor* | Can the VM get the current contended monitor of a thread? |
| boolean | *canGetMonitorInfo* | Can the VM get the monitor information for a given object? |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### ClassPaths Command (13)

Retrieve the classpath and bootclasspath of the target VM. If the classpath is not defined, returns an empty list. If the bootclasspath is not defined returns an empty list.

Out Data

(None)

Reply Data

| | | | |
|---|---|---|---|
| string | | *baseDir* | Base directory used to resolve relative paths in either of the following lists. |
| int | | *classpaths* | Number of paths in classpath. |
| Repeated *classpaths* times: | | | |
| | string | *path* | One component of classpath |
| int | | *bootclasspaths* | Number of paths in bootclasspath. |
| Repeated *bootclasspaths* times: | | | |
| | string | *path* | One component of bootclasspath |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### DisposeObjects Command (14)

Releases a list of object IDs. For each object in the list, the following applies. The count of references held by the back-end (the reference count) will be decremented by refCnt. If thereafter the reference count is less than or equal to zero, the ID is freed. Any back-end resources associated with the freed ID may be freed, and if garbage collection was disabled for the object, it will be re-enabled. The sender of this command promises that no further commands will be sent referencing a freed ID.

Use of this command is not required. If it is not sent, resources associated with each ID will be freed by the back-end at some time after the corresponding object is garbage collected. It is most useful to use this command to reduce the load on the back-end if a very large number of objects has been retrieved from the back-end (a large array, for example) but may not be garbage collected any time soon.

IDs may be re-used by the back-end after they have been freed with this command.This description assumes reference counting, a back-end may use any implementation which operates equivalently.

Out Data

| | | | |
|---|---|---|---|
| int | | *requests* | Number of object dispose requests that follow |
| Repeated *requests* times: | | | |
| | objectID | *object* | The object ID |
| | int | *refCnt* | The number of times this object ID has been part of a packet received from the back-end. An accurate count prevents the object ID from being freed on the back-end if it is part of an incoming packet, not yet handled by the front-end. |

Reply Data

(None)

Error Data

(None)

### HoldEvents Command (15)

Tells the target VM to stop sending events. Events are not discarded; they are held until a subsequent ReleaseEvents command is sent. This command is useful to control the number of events sent to the debugger VM in situations where very large numbers of events are generated. While events are held by the debugger back-end, application execution may be frozen by the debugger back-end to prevent buffer overflows on the back end.Responses to commands are never held and are not affected by thiscommand. If events are already being held, this command is ignored.

Out Data

(None)

Reply Data

(None)

Error Data

(None)

### ReleaseEvents Command (16)

Tells the target VM to continue sending events. This command is used to restore normal activity after a HoldEvents command. If there is no current HoldEvents command in effect, this command is ignored.

Out Data

(None)

Reply Data

(None)

Error Data

(None)

### CapabilitiesNew Command (17)

Retrieve all of this VM's capabilities. The capabilities are returned as booleans, each indicating the presence or absence of a capability. The commands associated with each capability will return the NOT_IMPLEMENTED error if the cabability is not available.Since JDWP version 1.4.

Out Data

(None)

Reply Data

| | | |
|---|---|---|
| boolean | *canWatchFieldModification* | Can the VM watch field modification, and therefore can it send the Modification Watchpoint Event? |
| boolean | *canWatchFieldAccess* | Can the VM watch field access, and therefore can it send the Access Watchpoint Event? |
| boolean | *canGetBytecodes* | Can the VM get the bytecodes of a given method? |
| boolean | *canGetSyntheticAttribute* | Can the VM determine whether a field or method is synthetic? (that is, can the VM determine if the method or the field was invented by the compiler?) |
| boolean | *canGetOwnedMonitorInfo* | Can the VM get the owned monitors information for a thread? |
| boolean | *canGetCurrentContendedMonitor* | Can the VM get the current contended monitor of a thread? |

| | | |
|---|---|---|
| boolean | *canGetMonitorInfo* | Can the VM get the monitor information for a given object? |
| boolean | *canRedefineClasses* | Can the VM redefine classes? |
| boolean | *canAddMethod* | Can the VM add methods when redefining classes? |
| boolean | *canUnrestrictedlyRedefineClasses* | Can the VM redefine classesin arbitrary ways? |
| boolean | *canPopFrames* | Can the VM pop stack frames? |
| boolean | *canUseInstanceFilters* | Can the VM filter events by specific object? |
| boolean | *canGetSourceDebugExtension* | Can the VM get the source debug extension? |
| boolean | *canRequestVMDeathEvent* | Can the VM request VM death events? |
| boolean | *canSetDefaultStratum* | Can the VM set a default stratum? |
| boolean | *canGetInstanceInfo* | Can the VM return instances, counts of instances of classes and referring objects? |
| boolean | *canRequestMonitorEvents* | Can the VM request monitor events? |
| boolean | *canGetMonitorFrameInfo* | Can the VM get monitors with frame depth info? |
| boolean | *canUseSourceNameFilters* | Can the VM filter class prepare events by source name? |
| boolean | *canGetConstantPool* | Can the VM return the constant pool information? |
| boolean | *canForceEarlyReturn* | Can the VM force early return from a method? |
| boolean | *reserved22* | Reserved for future capability |
| boolean | *reserved23* | Reserved for future capability |
| boolean | *reserved24* | Reserved for future capability |
| boolean | *reserved25* | Reserved for future capability |
| boolean | *reserved26* | Reserved for future capability |
| boolean | *reserved27* | Reserved for future capability |
| boolean | *reserved28* | Reserved for future capability |

| | | |
|---|---|---|
| boolean | *reserved29* | Reserved for future capability |
| boolean | *reserved30* | Reserved for future capability |
| boolean | *reserved31* | Reserved for future capability |
| boolean | *reserved32* | Reserved for future capability |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### RedefineClasses Command (18)

Installs new class definitions. If there are active stack frames in methods of the redefined classes in the target VM then those active frames continue to run the bytecodes of the original method. These methods are considered obsolete - see IsObsolete. The methods in the redefined classes will be used for new invokes in the target VM. The original method ID refers to the redefined method. All breakpoints in the redefined classes are cleared.If resetting of stack frames is desired, the PopFrames command can be used to pop frames with obsolete methods.

Requires canRedefineClasses capability - see CapabilitiesNew. In addition to the canRedefineClasses capability, the target VM must have the canAddMethod capability to add methods when redefining classes, or the canUnrestrictedlyRedefineClasses to redefine classes in arbitrary ways.

Out Data

| | | | |
|---|---|---|---|
| int | | *classes* | Number of reference types that follow. |
| Repeated *classes* times: | | | |
| | referenceTypeID | *refType* | The reference type. |
| | int | *classfile* | Number of bytes defining class (below) |
| | Repeated *classfile* times: | | |
| | byte | *classbyte* | byte in JVM class file format. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_CLASS | One of the refTypes is not the ID of a reference type. |
| INVALID_OBJECT | One of the refTypes is not a known ID. |

| | |
|---|---|
| UNSUPPORTED_VERSION | A class file has a version number not supported by this VM. |
| INVALID_CLASS_FORMAT | The virtual machine attempted to read a class file and determined that the file is malformed or otherwise cannot be interpreted as a class file. |
| CIRCULAR_CLASS_DEFINITION | A circularity has been detected while initializing a class. |
| FAILS_VERIFICATION | The verifier detected that a class file, though well formed, contained some sort of internal inconsistency or security problem. |
| NAMES_DONT_MATCH | The class name defined in the new class file is different from the name in the old class object. |
| NOT_IMPLEMENTED | No aspect of this functionality is implemented (CapabilitiesNew.canRedefineClasses is false) |
| ADD_METHOD_NOT_IMPLEMENTED | Adding methods has not been implemented. |
| SCHEMA_CHANGE_NOT_IMPLEMENTED | Schema change has not been implemented. |
| HIERARCHY_CHANGE_NOT_IMPLEMENTED | A direct superclass is different for the new class version, or the set of directly implemented interfaces is different and canUnrestrictedlyRedefineClasses is false. |
| DELETE_METHOD_NOT_IMPLEMENTED | The new class version does not declare a method declared in the old class version and canUnrestrictedlyRedefineClasses is false. |
| CLASS_MODIFIERS_CHANGE_NOT_IMPLEMENTED | The new class version has different modifiers and and canUnrestrictedlyRedefineClasses is false. |
| METHOD_MODIFIERS_CHANGE_NOT_IMPLEMENTED | A method in the new class version has different modifiers than its counterpart |

| | |
|---|---|
| | in the old class version and and canUnrestrictedlyRedefineClasses is false. |
| VM_DEAD | The virtual machine is not running. |

### SetDefaultStratum Command (19)

Set the default stratum. Requires canSetDefaultStratum capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| string | *stratumID* | default stratum, or empty string to use reference type default. |

Reply Data

(None)

Error Data

| | |
|---|---|
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

### AllClassesWithGeneric Command (20)

Returns reference types for all classes currently loaded by the target VM. Both the JNI signature and the generic signature are returned for each class. Generic signatures are described in the signature attribute section in *The Java™ Virtual Machine Specification*. Since JDWP version 1.5.

Out Data

(None)

Reply Data

| | | |
|---|---|---|
| int | *classes* | Number of reference types that follow. |
| Repeated *classes* times: | | |
| byte | *refTypeTag* | Kind of following reference type. |
| referenceTypeID | *typeID* | Loaded reference type |
| string | *signature* | The JNI signature of the loaded reference type. |
| string | *genericSignature* | The generic signature of the loaded reference type or an empty string if there is none. |
| int | *status* | The current class status. |

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### InstanceCounts Command (21)

Returns the number of instances of each reference type in the input list. Only instances that are reachable for the purposes of garbage collection are counted. If a reference type is invalid, eg. it has been unloaded, zero is returned for its instance count.

Since JDWP version 1.6. Requires canGetInstanceInfo capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| int | *refTypesCount* | Number of reference types that follow. Must be non-negative. |
| Repeated *refTypesCount* times: | | |
| referenceTypeID | *refType* | A reference type ID. |

Reply Data

| | | |
|---|---|---|
| int | *counts* | The number of counts that follow. |
| Repeated *counts* times: | | |
| long | *instanceCount* | The number of instances for the corresponding reference type in 'Out Data'. |

Error Data

| | |
|---|---|
| ILLEGAL_ARGUMENT | refTypesCount is less than zero. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

## ReferenceType Command Set (2)

### Signature Command (1)

Returns the JNI signature of a reference type. JNI signature formats are described in the Java Native Inteface Specification

For primitive classes the returned signature is the signature of the corresponding primitive type; for example, "I" is returned as the signature of the class represented by java.lang.Integer.TYPE.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| string | *signature* | The JNI signature for the reference type. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### ClassLoader Command (2)

Returns the instance of java.lang.ClassLoader which loaded a given reference type. If the reference type was loaded by the system class loader, the returned object ID is null.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| classLoaderID | *classLoader* | The class loader for the reference type. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Modifiers Command (3)

Returns the modifiers (also known as access flags) for a reference type. The returned bit mask contains information on the declaration of the reference type. If the reference type is an array or a primitive class (for example, java.lang.Integer.TYPE), the value of the returned bit mask is undefined.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| int | *modBits* | Modifier bits as defined in Chapter 4 of *The Java™ Virtual Machine Specification* |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### Fields Command (4)

Returns information for each field in a reference type. Inherited fields are not included. The field list will include any synthetic fields created by the compiler. Fields are returned in the order they occur in the class file.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *declared* | Number of declared fields. |
| Repeated *declared* times: | | | |
| | fieldID | *fieldID* | Field ID. |
| | string | *name* | Name of field. |
| | string | *signature* | JNI Signature of field. |
| | int | *modBits* | The modifier bit flags (also known as access flags) which provide additional information on the field declaration. Individual flag values are defined in Chapter 4 of *The Java™ Virtual Machine Specification*. In addition, The `0xf0000000` bit identifies the field as synthetic, if the synthetic attribute capability is available. |

Error Data

| | |
|---|---|
| CLASS_NOT_PREPARED | Class has been loaded but not yet prepared. |
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Methods Command (5)

Returns information for each method in a reference type. Inherited methods are not included. The list of methods will include constructors (identified with the name "<init>"), the initialization method (identified with the name "<clinit>") if present, and any synthetic methods created by the compiler. Methods are returned in the order they occur in the class file.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *declared* | Number of declared methods. |
| Repeated *declared* times: | | | |
| | methodID | *methodID* | Method ID. |
| | string | *name* | Name of method. |
| | string | *signature* | JNI signature of method. |
| | int | *modBits* | The modifier bit flags (also known as access flags) which provide additional information on the method declaration. Individual flag values are defined in Chapter 4 of *The Java™ Virtual Machine Specification*. In addition, The `0xf0000000` bit identifies the method as synthetic, if the synthetic attribute capability is available. |

Error Data

| | |
|---|---|
| CLASS_NOT_PREPARED | Class has been loaded but not yet prepared. |
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### GetValues Command (6)

Returns the value of one or more static fields of the reference type. Each field must be member of the reference type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced; for example, the values of private fields can be obtained.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |
| int | *fields* | The number of values to get |
| Repeated *fields* times: | | |
| fieldID | *fieldID* | A field to get |

Reply Data

| | | |
|---|---|---|
| int | *values* | The number of values returned, always equal to fields, the number of values to get. |

| Repeated *values* times: | | | |
|---|---|---|---|
| | value | *value* | The field value |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| INVALID_FIELDID | Invalid field. |
| VM_DEAD | The virtual machine is not running. |

### SourceFile Command (7)

Returns the name of source file in which a reference type was declared.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| string | *sourceFile* | The source file name. No path information for the file is included |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| ABSENT_INFORMATION | The source file attribute is absent. |
| VM_DEAD | The virtual machine is not running. |

### NestedTypes Command (8)

Returns the classes and interfaces directly nested within this type.Types further nested within those types are not included.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| int | *classes* | The number of nested classes and interfaces |
| Repeated *classes* times: | | |

| | | | |
|---|---|---|---|
| | byte | *refTypeTag* | Kind of following reference type. |
| | referenceTypeID | *typeID* | The nested class or interface ID. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Status Command (9)

Returns the current status of the reference type. The status indicates the extent to which the reference type has been initialized, as described in section 2.1.6 of *The Java™ Virtual Machine Specification*. If the class is linked the PREPARED and VERIFIED bits in the returned status bits will be set. If the class is initialized the INITIALIZED bit in the returned status bits will be set. If an error occured during initialization then the ERROR bit in the returned status bits will be set. The returned status bits are undefined for array types and for primitive classes (such as java.lang.Integer.TYPE).

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| int | *status* | Status bits:See JDWP.ClassStatus |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Interfaces Command (10)

Returns the interfaces declared as implemented by this class. Interfaces indirectly implemented (extended by the implemented interface or implemented by a superclass) are not included.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| int | *interfaces* | The number of implemented interfaces |
| Repeated *interfaces* times: | | |

| | | | |
|---|---|---|---|
| | interfaceID | *interfaceType* | implemented interface. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### ClassObject Command (11)

Returns the class object corresponding to this type.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| classObjectID | *classObject* | class object. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### SourceDebugExtension Command (12)

Returns the value of the SourceDebugExtension attribute. Since JDWP version 1.4. Requires canGetSourceDebugExtension capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| string | *extension* | extension attribute |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| ABSENT_INFORMATION | If the extension is not specified. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### SignatureWithGeneric Command (13)

Returns the JNI signature of a reference type along with the generic signature if there is one. Generic signatures are described in the signature attribute section in *The Java™ Virtual Machine Specification*. Since JDWP version 1.5.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | |
|---|---|---|
| string | *signature* | The JNI signature for the reference type. |
| string | *genericSignature* | The generic signature for the reference type or an empty string if there is none. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### FieldsWithGeneric Command (14)

Returns information, including the generic signature if any, for each field in a reference type. Inherited fields are not included. The field list will include any synthetic fields created by the compiler. Fields are returned in the order they occur in the class file. Generic signatures are described in the signature attribute section in *The Java™ Virtual Machine Specification*. Since JDWP version 1.5.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *declared* | Number of declared fields. |
| Repeated *declared* times: | | | |
| | fieldID | *fieldID* | Field ID. |
| | string | *name* | The name of the field. |
| | string | *signature* | The JNI signature of the field. |

| | | | |
|---|---|---|---|
| | string | *genericSignature* | The generic signature of the field, or an empty string if there is none. |
| | int | *modBits* | The modifier bit flags (also known as access flags) which provide additional information on the field declaration. Individual flag values are defined in Chapter 4 of *The Java™ Virtual Machine Specification*. In addition, The `0xf0000000` bit identifies the field as synthetic, if the synthetic attribute capability is available. |

Error Data

| | |
|---|---|
| CLASS_NOT_PREPARED | Class has been loaded but not yet prepared. |
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### MethodsWithGeneric Command (15)

Returns information, including the generic signature if any, for each method in a reference type. Inherited methodss are not included. The list of methods will include constructors (identified with the name "<init>"), the initialization method (identified with the name "<clinit>") if present, and any synthetic methods created by the compiler. Methods are returned in the order they occur in the class file. Generic signatures are described in the signature attribute section in *The Java™ Virtual Machine Specification*. Since JDWP version 1.5.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *declared* | Number of declared methods. |
| Repeated *declared* times: | | | |
| | methodID | *methodID* | Method ID. |
| | string | *name* | The name of the method. |
| | string | *signature* | The JNI signature of the method. |
| | string | *genericSignature* | The generic signature of the method, or an empty string if there is none. |
| | int | *modBits* | The modifier bit flags (also known as access flags) which provide additional information on the method declaration. |

| | | | |
|---|---|---|---|
| | | | Individual flag values are defined in Chapter 4 of *The Java™ Virtual Machine Specification*. In addition, The `0xf0000000` bit identifies the method as synthetic, if the synthetic attribute capability is available. |

Error Data

| | |
|---|---|
| CLASS_NOT_PREPARED | Class has been loaded but not yet prepared. |
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Instances Command (16)

Returns instances of this reference type. Only instances that are reachable for the purposes of garbage collection are returned.

Since JDWP version 1.6. Requires canGetInstanceInfo capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The reference type ID. |
| int | *maxInstances* | Maximum number of instances to return. Must be non-negative. If zero, all instances are returned. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *instances* | The number of instances that follow. |
| Repeated *instances* times: | | | |
| | tagged-objectID | *instance* | An instance of this reference type. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| ILLEGAL_ARGUMENT | maxInstances is less than zero. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

### ClassFileVersion Command (17)

Returns the class file major and minor version numbers, as defined in the class file format of the Java Virtual Machine specification.

Since JDWP version 1.6.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The class. |

Reply Data

| | | |
|---|---|---|
| int | *majorVersion* | Major version number |
| int | *minorVersion* | Minor version number |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| ABSENT_INFORMATION | The class file version information is absent for primitive and array types. |
| VM_DEAD | The virtual machine is not running. |

### ConstantPool Command (18)

Return the raw bytes of the constant pool in the format of the constant_pool item of the Class File Format in *The Java™ Virtual Machine Specification*.

Since JDWP version 1.6. Requires canGetConstantPool capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The class. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *count* | Total number of constant pool entries plus one. This corresponds to the constant_pool_count item of the Class File Format in *The Java™ Virtual Machine Specification*. |
| int | | *bytes* | |
| Repeated *bytes* times: | | | |
| | byte | *cpbytes* | Raw bytes of constant pool |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| NOT_IMPLEMENTED | If the target virtual machine does not support the retrieval of constant pool information. |
| ABSENT_INFORMATION | The Constant Pool information is absent for primitive and array types. |
| VM_DEAD | The virtual machine is not running. |

## ClassType Command Set (3)

### Superclass Command (1)

Returns the immediate superclass of a class.

Out Data

| | | |
|---|---|---|
| classID | *clazz* | The class type ID. |

Reply Data

| | | |
|---|---|---|
| classID | *superclass* | The superclass (null if the class ID for java.lang.Object is specified). |

Error Data

| | |
|---|---|
| INVALID_CLASS | clazz is not the ID of a class. |
| INVALID_OBJECT | clazz is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### SetValues Command (2)

Sets the value of one or more static fields. Each field must be member of the class type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced; for example, the values of private fields can be set. Final fields cannot be set.For primitive values, the value's type must match the field's type exactly. For object values, there must exist a widening reference conversion from the value's type to thefield's type and the field's type must be loaded.

Out Data

| | | |
|---|---|---|
| classID | *clazz* | The class type ID. |
| int | *values* | The number of fields to set. |

| Repeated *values* times: | | | |
|---|---|---|---|
| | fieldID | *fieldID* | Field to set. |
| | untagged-value | *value* | Value to put in the field. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_CLASS | clazz is not the ID of a class. |
| CLASS_NOT_PREPARED | Class has been loaded but not yet prepared. |
| INVALID_OBJECT | clazz is not a known ID or a value of an object field is not a known ID. |
| INVALID_FIELDID | Invalid field. |
| VM_DEAD | The virtual machine is not running. |

### InvokeMethod Command (3)

Invokes a static method. The method must be member of the class type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced; for example, private methods can be invoked.

The method invocation will occur in the specified thread. Method invocation can occur only if the specified thread has been suspended by an event. Method invocation is not supported when the target VM has been suspended by the front-end.

The specified method is invoked with the arguments in the specified argument list. The method invocation is synchronous; the reply packet is not sent until the invoked method returns in the target VM. The return value (possibly the void value) is included in the reply packet. If the invoked method throws an exception, the exception object ID is set in the reply packet; otherwise, the exception object ID is null.

For primitive arguments, the argument value's type must match the argument's type exactly. For object arguments, there must exist a widening reference conversion from the argument value's type to the argument's type and the argument's type must be loaded.

By default, all threads in the target VM are resumed while the method is being invoked if they were previously suspended by an event or by command. This is done to prevent the deadlocks that will occur if any of the threads own monitors that will be needed by the invoked method. It is possible that breakpoints or other events might occur during the invocation. Note, however, that this implicit resume acts exactly like the ThreadReference resume command, so if the thread's suspend count is greater than 1, it will remain in a suspended state during the invocation. By default, when the invocation completes, all threads in the target VM are suspended, regardless their state before the invocation.

The resumption of other threads during the invoke can be prevented by specifying the INVOKE_SINGLE_THREADED bit flag in the `options` field; however, there is no protection against or recovery from the deadlocks described above, so this option should be used with great caution. Only the specified thread will be resumed (as described for all threads above). Upon completion of a single threaded invoke, the invoking thread will be suspended once again. Note that any threads started during the single threaded invocation will not be suspended when the invocation completes.

If the target VM is disconnected during the invoke (for example, through the VirtualMachine dispose command) the method invocation continues.

Out Data

| | | |
|---|---|---|
| classID | *clazz* | The class type ID. |
| threadID | *thread* | The thread in which to invoke. |
| methodID | *methodID* | The method to invoke. |
| int | *arguments* | |
| Repeated *arguments* times: | | |
| value | *arg* | The argument value. |
| int | *options* | Invocation options |

Reply Data

| | | |
|---|---|---|
| value | *returnValue* | The returned value. |
| tagged-objectID | *exception* | The thrown exception. |

Error Data

| | |
|---|---|
| INVALID_CLASS | clazz is not the ID of a class. |
| INVALID_OBJECT | clazz is not a known ID. |
| INVALID_METHODID | methodID is not the ID of a static method in this class type or one of its superclasses. |
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| THREAD_NOT_SUSPENDED | If the specified thread has not been suspended by an event. |
| VM_DEAD | The virtual machine is not running. |

### NewInstance Command (4)

Creates a new object of this type, invoking the specified constructor. The constructor method ID must be a member of the class type.

Instance creation will occur in the specified thread. Instance creation can occur only if the specified thread has been suspended by an event. Method invocation is not supported when the target VM has been suspended by the front-end.

The specified constructor is invoked with the arguments in the specified argument list. The constructor invocation is synchronous; the reply packet is not sent until the invoked method returns in the target VM. The return value (possibly the void value) is included in the reply packet. If the constructor throws an exception, the exception object ID is set in the reply packet; otherwise, the exception object ID is null.

For primitive arguments, the argument value's type must match the argument's type exactly. For object arguments, there must exist a widening reference conversion from the argument value's type to the argument's type and the argument's type must be loaded.

By default, all threads in the target VM are resumed while the method is being invoked if they were previously suspended by an event or by command. This is done to prevent the deadlocks that will occur if any of the threads own monitors that will be needed by the invoked method. It is possible that breakpoints or other events might occur during the invocation. Note, however, that this implicit resume acts exactly like the ThreadReference resume command, so if the thread's suspend count is greater than 1, it will remain in a suspended state during the invocation. By default, when the invocation completes, all threads in the target VM are suspended, regardless their state before the invocation.

The resumption of other threads during the invoke can be prevented by specifying the INVOKE_SINGLE_THREADED bit flag in the `options` field; however, there is no protection against or recovery from the deadlocks described above, so this option should be used with great caution. Only the specified thread will be resumed (as described for all threads above). Upon completion of a single threaded invoke, the invoking thread will be suspended once again. Note that any threads started during the single threaded invocation will not be suspended when the invocation completes.

If the target VM is disconnected during the invoke (for example, through the VirtualMachine dispose command) the method invocation continues.

Out Data

| | | |
|---|---|---|
| classID | *clazz* | The class type ID. |
| threadID | *thread* | The thread in which to invoke the constructor. |
| methodID | *methodID* | The constructor to invoke. |
| int | *arguments* | |
| Repeated *arguments* times: | | |

| | | | |
|---|---|---|---|
| | value | *arg* | The argument value. |
| int | | *options* | Constructor invocation options |

Reply Data

| | | |
|---|---|---|
| tagged-objectID | *newObject* | The newly created object, or null if the constructor threw an exception. |
| tagged-objectID | *exception* | The thrown exception, if any; otherwise, null. |

Error Data

| | |
|---|---|
| INVALID_CLASS | clazz is not the ID of a class. |
| INVALID_OBJECT | clazz is not a known ID or a value of an object parameter is not a known ID.. |
| INVALID_METHODID | methodID is not the ID of a method. |
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| THREAD_NOT_SUSPENDED | If the specified thread has not been suspended by an event. |
| VM_DEAD | The virtual machine is not running. |

## ArrayType Command Set (4)

### NewInstance Command (1)

Creates a new array object of this type with a given length.

Out Data

| | | |
|---|---|---|
| arrayTypeID | *arrType* | The array type of the new instance. |
| int | *length* | The length of the array. |

Reply Data

| | | |
|---|---|---|
| tagged-objectID | *newArray* | The newly created array object. |

Error Data

| | |
|---|---|
| INVALID_ARRAY | The array is invalid. |
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

## InterfaceType Command Set (5)

### InvokeMethod Command (1)

Invokes a static method. The method must not be a static initializer. The method must be a member of the interface type.

Since JDWP version 1.8

The method invocation will occur in the specified thread. Method invocation can occur only if the specified thread has been suspended by an event. Method invocation is not supported when the target VM has been suspended by the front-end.

The specified method is invoked with the arguments in the specified argument list. The method invocation is synchronous; the reply packet is not sent until the invoked method returns in the target VM. The return value (possibly the void value) is included in the reply packet. If the invoked method throws an exception, the exception object ID is set in the reply packet; otherwise, the exception object ID is null.

For primitive arguments, the argument value's type must match the argument's type exactly. For object arguments, there must exist a widening reference conversion from the argument value's type to the argument's type and the argument's type must be loaded.

By default, all threads in the target VM are resumed while the method is being invoked if they were previously suspended by an event or by a command. This is done to prevent the deadlocks that will occur if any of the threads own monitors that will be needed by the invoked method. It is possible that breakpoints or other events might occur during the invocation. Note, however, that this implicit resume acts exactly like the ThreadReference resume command, so if the thread's suspend count is greater than 1, it will remain in a suspended state during the invocation. By default, when the invocation completes, all threads in the target VM are suspended, regardless their state before the invocation.

The resumption of other threads during the invoke can be prevented by specifying the INVOKE_SINGLE_THREADED bit flag in the `options` field; however, there is no protection against or recovery from the deadlocks described above, so this option should be used with great caution. Only the specified thread will be resumed (as described for all threads above). Upon completion of a single threaded invoke, the invoking thread will be suspended once again. Note that any threads started during the single threaded invocation will not be suspended when the invocation completes.

If the target VM is disconnected during the invoke (for example, through the VirtualMachine dispose command) the method invocation continues.

Out Data

| | | |
|---|---|---|
| interfaceID | *clazz* | The interface type ID. |

| | | |
|---|---|---|
| threadID | *thread* | The thread in which to invoke. |
| methodID | *methodID* | The method to invoke. |
| int | *arguments* | |
| Repeated *arguments* times: | | |
| value | *arg* | The argument value. |
| int | *options* | Invocation options |

Reply Data

| | | |
|---|---|---|
| value | *returnValue* | The returned value. |
| tagged-objectID | *exception* | The thrown exception. |

Error Data

| | |
|---|---|
| INVALID_CLASS | clazz is not the ID of an interface. |
| INVALID_OBJECT | clazz is not a known ID. |
| INVALID_METHODID | methodID is not the ID of a static method in this interface type or is the ID of a static initializer. |
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| THREAD_NOT_SUSPENDED | If the specified thread has not been suspended by an event. |
| VM_DEAD | The virtual machine is not running. |

## Method Command Set (6)

### LineTable Command (1)

Returns line number information for the method, if present. The line table maps source line numbers to the initial code index of the line. The line table is ordered by code index (from lowest to highest). The line number information is constant unless a new class definition is installed using RedefineClasses.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The class. |
| methodID | *methodID* | The method. |

Reply Data

| | | |
|---|---|---|
| long | *start* | Lowest valid code index for the method, >=0, or -1 if the method is native |
| long | *end* | Highest valid code index for the method, >=0, or -1 if the method is native |
| int | *lines* | The number of entries in the line table for this method. |
| Repeated *lines* times: | | |
| long | *lineCodeIndex* | Initial code index of the line, start <= lineCodeIndex < end |
| int | *lineNumber* | Line number. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| INVALID_METHODID | methodID is not the ID of a method. |
| VM_DEAD | The virtual machine is not running. |

### VariableTable Command (2)

Returns variable information for the method. The variable table includes arguments and locals declared within the method. For instance methods, the "this" reference is included in the table. Also, synthetic variables may be present.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The class. |
| methodID | *methodID* | The method. |

Reply Data

| | | |
|---|---|---|
| int | *argCnt* | The number of words in the frame used by arguments. Eight-byte arguments use two words; all others use one. |
| int | *slots* | The number of variables. |
| Repeated *slots* times: | | |
| long | *codeIndex* | First code index at which the variable is visible (unsigned). Used in conjunction with `length`. The variable can be get or set only when the current `codeIndex` <= current frame code index < `codeIndex + length` |

| | | | |
|---|---|---|---|
| | string | *name* | The variable's name. |
| | string | *signature* | The variable type's JNI signature. |
| | int | *length* | Unsigned value used in conjunction with `codeIndex`. The variable can be get or set only when the current `codeIndex` <= current frame code index < `code index + length` |
| | int | *slot* | The local variable's index in its frame |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| INVALID_METHODID | methodID is not the ID of a method. |
| ABSENT_INFORMATION | there is no variable information for the method. |
| VM_DEAD | The virtual machine is not running. |

### Bytecodes Command (3)

Retrieve the method's bytecodes as defined in *The Java™ Virtual Machine Specification*. Requires canGetBytecodes capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The class. |
| methodID | *methodID* | The method. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *bytes* | |
| Repeated *bytes* times: | | | |
| | byte | *bytecode* | A Java bytecode. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| INVALID_METHODID | methodID is not the ID of a method. |
| NOT_IMPLEMENTED | If the target virtual machine does not support the retrieval of bytecodes. |

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### IsObsolete Command (4)

Determine if this method is obsolete. A method is obsolete if it has been replaced by a non-equivalent method using the RedefineClasses command. The original and redefined methods are considered equivalent if their bytecodes are the same except for indices into the constant pool and the referenced constants are equal.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The class. |
| methodID | *methodID* | The method. |

Reply Data

| | | |
|---|---|---|
| boolean | *isObsolete* | true if this method has been replacedby a non-equivalent method usingthe RedefineClasses command. |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| INVALID_METHODID | methodID is not the ID of a method. |
| NOT_IMPLEMENTED | If the target virtual machine does not support this query. |
| VM_DEAD | The virtual machine is not running. |

### VariableTableWithGeneric Command (5)

Returns variable information for the method, including generic signatures for the variables. The variable table includes arguments and locals declared within the method. For instance methods, the "this" reference is included in the table. Also, synthetic variables may be present. Generic signatures are described in the signature attribute section in *The Java™ Virtual Machine Specification*. Since JDWP version 1.5.

Out Data

| | | |
|---|---|---|
| referenceTypeID | *refType* | The class. |
| methodID | *methodID* | The method. |

Reply Data

| | | |
|---|---|---|
| int | *argCnt* | The number of words in the frame used by arguments. Eight-byte arguments use two words; all others use one. |

| | | | |
|---|---|---|---|
| int | | *slots* | The number of variables. |
| Repeated *slots* times: | | | |
| | long | *codeIndex* | First code index at which the variable is visible (unsigned). Used in conjunction with `length`. The variable can be get or set only when the current `codeIndex` <= current frame code index < `codeIndex + length` |
| | string | *name* | The variable's name. |
| | string | *signature* | The variable type's JNI signature. |
| | string | *genericSignature* | The variable type's generic signature or an empty string if there is none. |
| | int | *length* | Unsigned value used in conjunction with `codeIndex`. The variable can be get or set only when the current `codeIndex` <= current frame code index < `code index + length` |
| | int | *slot* | The local variable's index in its frame |

Error Data

| | |
|---|---|
| INVALID_CLASS | refType is not the ID of a reference type. |
| INVALID_OBJECT | refType is not a known ID. |
| INVALID_METHODID | methodID is not the ID of a method. |
| ABSENT_INFORMATION | there is no variable information for the method. |
| VM_DEAD | The virtual machine is not running. |

## Field Command Set (8)

## ObjectReference Command Set (9)

### ReferenceType Command (1)

Returns the runtime type of the object. The runtime type will be a class or an array.

Out Data

| | | |
|---|---|---|
| objectID | *object* | The object ID |

Reply Data

| | | |
|---|---|---|
| byte | *refTypeTag* | Kind of following reference type. |

| | | |
|---|---|---|
| referenceTypeID | *typeID* | The runtime reference type. |

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| VM_DEAD | The virtual machine is not running. |

### GetValues Command (2)

Returns the value of one or more instance fields. Each field must be member of the object's type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced; for example, the values of private fields can be obtained.

Out Data

| | | | |
|---|---|---|---|
| objectID | | *object* | The object ID |
| int | | *fields* | The number of values to get |
| Repeated *fields* times: | | | |
| | fieldID | *fieldID* | Field to get. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *values* | The number of values returned, always equal to 'fields', the number of values to get. Field values are ordered in the reply in the same order as corresponding fieldIDs in the command. |
| Repeated *values* times: | | | |
| | value | *value* | The field value |

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_FIELDID | Invalid field. |
| VM_DEAD | The virtual machine is not running. |

### SetValues Command (3)

Sets the value of one or more instance fields. Each field must be member of the object's type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced; for example, the values of private fields can be set. For primitive values, the value's type must match the field's type exactly. For object values, there must be a widening reference conversion from the value's type to thefield's type and the field's type must be loaded.

Out Data

| | | | |
|---|---|---|---|
| objectID | | *object* | The object ID |
| int | | *values* | The number of fields to set. |
| Repeated *values* times: | | | |
| | fieldID | *fieldID* | Field to set. |
| | untagged-value | *value* | Value to put in the field. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_FIELDID | Invalid field. |
| VM_DEAD | The virtual machine is not running. |

### MonitorInfo Command (5)

Returns monitor information for an object. All threads int the VM must be suspended.Requires canGetMonitorInfo capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| objectID | *object* | The object ID |

Reply Data

| | | | |
|---|---|---|---|
| threadID | | *owner* | The monitor owner, or null if it is not currently owned. |
| int | | *entryCount* | The number of times the monitor has been entered. |
| int | | *waiters* | The number of threads that are waiting for the monitor 0 if there is no current owner |
| Repeated *waiters* times: | | | |
| | threadID | *thread* | A thread waiting for this monitor. |

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### InvokeMethod Command (6)

Invokes a instance method. The method must be member of the object's type or one of its superclasses, superinterfaces, or implemented interfaces. Access control is not enforced; for example, private methods can be invoked.

The method invocation will occur in the specified thread. Method invocation can occur only if the specified thread has been suspended by an event. Method invocation is not supported when the target VM has been suspended by the front-end.

The specified method is invoked with the arguments in the specified argument list. The method invocation is synchronous; the reply packet is not sent until the invoked method returns in the target VM. The return value (possibly the void value) is included in the reply packet. If the invoked method throws an exception, the exception object ID is set in the reply packet; otherwise, the exception object ID is null.

For primitive arguments, the argument value's type must match the argument's type exactly. For object arguments, there must be a widening reference conversion from the argument value's type to the argument's type and the argument's type must be loaded.

By default, all threads in the target VM are resumed while the method is being invoked if they were previously suspended by an event or by a command. This is done to prevent the deadlocks that will occur if any of the threads own monitors that will be needed by the invoked method. It is possible that breakpoints or other events might occur during the invocation. Note, however, that this implicit resume acts exactly like the ThreadReference resume command, so if the thread's suspend count is greater than 1, it will remain in a suspended state during the invocation. By default, when the invocation completes, all threads in the target VM are suspended, regardless their state before the invocation.

The resumption of other threads during the invoke can be prevented by specifying the INVOKE_SINGLE_THREADED bit flag in the `options` field; however, there is no protection against or recovery from the deadlocks described above, so this option should be used with great caution. Only the specified thread will be resumed (as described for all threads above). Upon completion of a single threaded invoke, the invoking thread will be suspended once again. Note that any threads started during the single threaded invocation will not be suspended when the invocation completes.

If the target VM is disconnected during the invoke (for example, through the VirtualMachine dispose command) the method invocation continues.

#### Out Data

| | | |
|---|---|---|
| objectID | *object* | The object ID |
| threadID | *thread* | The thread in which to invoke. |
| classID | *clazz* | The class type. |

| | | |
|---|---|---|
| methodID | *methodID* | The method to invoke. |
| int | *arguments* | The number of arguments. |
| Repeated *arguments* times: | | |
| value | *arg* | The argument value. |
| int | *options* | Invocation options |

Reply Data

| | | |
|---|---|---|
| value | *returnValue* | The returned value, or null if an exception is thrown. |
| tagged-objectID | *exception* | The thrown exception, if any. |

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_CLASS | clazz is not the ID of a reference type. |
| INVALID_METHODID | methodID is not the ID of an instance method in this object's type or one of its superclasses, superinterfaces, or implemented interfaces. |
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| THREAD_NOT_SUSPENDED | If the specified thread has not been suspended by an event. |
| VM_DEAD | The virtual machine is not running. |

### DisableCollection Command (7)

Prevents garbage collection for the given object. By default all objects in back-end replies may be collected at any time the target VM is running. A call to this command guarantees that the object will not be collected. The EnableCollection command can be used to allow collection once again.

Note that while the target VM is suspended, no garbage collection will occur because all threads are suspended. The typical examination of variables, fields, and arrays during the suspension is safe without explicitly disabling garbage collection.

This method should be used sparingly, as it alters the pattern of garbage collection in the target VM and, consequently, may result in application behavior under the debugger that differs from its non-debugged behavior.

Out Data

| | | |
|---|---|---|
| objectID | *object* | The object ID |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| VM_DEAD | The virtual machine is not running. |

### EnableCollection Command (8)

Permits garbage collection for this object. By default all objects returned by JDWP may become unreachable in the target VM, and hence may be garbage collected. A call to this command is necessary only if garbage collection was previously disabled with the DisableCollection command.

Out Data

| | | |
|---|---|---|
| objectID | *object* | The object ID |

Reply Data

(None)

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### IsCollected Command (9)

Determines whether an object has been garbage collected in the target VM.

Out Data

| | | |
|---|---|---|
| objectID | *object* | The object ID |

Reply Data

| | | |
|---|---|---|
| boolean | *isCollected* | true if the object has been collected; false otherwise |

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| VM_DEAD | The virtual machine is not running. |

### ReferringObjects Command (10)

Returns objects that directly reference this object. Only objects that are reachable for the purposes of garbage collection are returned. Note that an object can also be referenced in other ways, such as from a local variable in a stack frame, or from a JNI global reference. Such non-object referrers are not returned by this command.

Since JDWP version 1.6. Requires canGetInstanceInfo capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| objectID | *object* | The object ID |
| int | *maxReferrers* | Maximum number of referring objects to return. Must be non-negative. If zero, all referring objects are returned. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *referringObjects* | The number of objects that follow. |
| Repeated *referringObjects* times: | | | |
| | tagged-objectID | *instance* | An object that references this object. |

Error Data

| | |
|---|---|
| INVALID_OBJECT | object is not a known ID. |
| ILLEGAL_ARGUMENT | maxReferrers is less than zero. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

## StringReference Command Set (10)

### Value Command (1)

Returns the characters contained in the string.

Out Data

| | | |
|---|---|---|
| objectID | *stringObject* | The String object ID. |

Reply Data

| | | |
|---|---|---|
| string | *stringValue* | UTF-8 representation of the string value. |

Error Data

| | |
|---|---|
| INVALID_STRING | The string is invalid. |
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| VM_DEAD | The virtual machine is not running. |

## ThreadReference Command Set (11)

### Name Command (1)

Returns the thread name.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | |
|---|---|---|
| string | *threadName* | The thread name. |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Suspend Command (2)

Suspends the thread.

Unlike java.lang.Thread.suspend(), suspends of both the virtual machine and individual threads are counted. Before a thread will run again, it must be resumed the same number of times it has been suspended.

Suspending single threads with command has the same dangers java.lang.Thread.suspend(). If the suspended thread holds a monitor needed by another running thread, deadlock is possible in the target VM (at least until the suspended thread is resumed again).

The suspended thread is guaranteed to remain suspended until resumed through one of the JDI resume methods mentioned above; the application in the target VM cannot resume the suspended thread through {@link java.lang.Thread#resume}.

Note that this doesn't change the status of the thread (see the ThreadStatus command.) For example, if it was Running, it will still appear running to other threads.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

### Resume Command (3)

Resumes the execution of a given thread. If this thread was not previously suspended by the front-end, calling this command has no effect. Otherwise, the count of pending suspends on this thread is decremented. If it is decremented to 0, the thread will continue to execute.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Status Command (4)

Returns the current status of a thread. The thread status reply indicates the thread status the last time it was running. the suspend status provides information on the thread's suspension, if any.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | |
|---|---|---|
| int | *threadStatus* | One of the thread status codes See JDWP.ThreadStatus |
| int | *suspendStatus* | One of the suspend status codes See JDWP.SuspendStatus |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### ThreadGroup Command (5)

Returns the thread group that contains a given thread.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | |
|---|---|---|
| threadGroupID | *group* | The thread group of this thread. |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Frames Command (6)

Returns the current call stack of a suspended thread. The sequence of frames starts with the currently executing frame, followed by its caller, and so on. The thread must be suspended, and the returned frameID is valid only while the thread is suspended.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |
| int | *startFrame* | The index of the first frame to retrieve. |
| int | *length* | The count of frames to retrieve (-1 means all remaining). |

Reply Data

| | | | |
|---|---|---|---|
| int | | *frames* | The number of frames retreived |
| Repeated *frames* times: | | | |
| | frameID | *frameID* | The ID of this frame. |
| | location | *location* | The current location of this frame |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### FrameCount Command (7)

Returns the count of frames on this thread's stack. The thread must be suspended, and the returned count is valid only while the thread is suspended. Returns JDWP.Error.errorThreadNotSuspended if not suspended.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | |
|---|---|---|
| int | *frameCount* | The count of frames on this thread's stack. |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### OwnedMonitors Command (8)

Returns the objects whose monitors have been entered by this thread. The thread must be suspended, and the returned information is relevant only while the thread is suspended. Requires canGetOwnedMonitorInfo capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | |
|---|---|---|
| int | *owned* | The number of owned monitors |
| Repeated *owned* times: | | |
| tagged-objectID | *monitor* | An owned monitor |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

### CurrentContendedMonitor Command (9)

Returns the object, if any, for which this thread is waiting. The thread may be waiting to enter a monitor, or it may be waiting, via the java.lang.Object.wait method, for another thread to invoke the notify method. The thread must be suspended, and the returned information is relevant only while the thread is suspended. Requires canGetCurrentContendedMonitor capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | |
|---|---|---|
| tagged-objectID | *monitor* | The contended monitor, or null if there is no current contended monitor. |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

### Stop Command (10)

Stops the thread with an asynchronous exception, as if done by java.lang.Thread.stop

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |
| objectID | *throwable* | Asynchronous exception. This object must be an instance of java.lang.Throwable or a subclass |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | If thread is not a known ID or the asynchronous exception has been garbage collected. |
| VM_DEAD | The virtual machine is not running. |

### Interrupt Command (11)

Interrupt the thread, as if done by java.lang.Thread.interrupt

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### SuspendCount Command (12)

Get the suspend count for this thread. The suspend count is the number of times the thread has been suspended through the thread-level or VM-level suspend commands without a corresponding resume

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | |
|---|---|---|
| int | *suspendCount* | The number of outstanding suspends of this thread. |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### OwnedMonitorsStackDepthInfo Command (13)

Returns monitor objects owned by the thread, along with stack depth at which the monitor was acquired. Returns stack depth of -1 if the implementation cannot determine the stack depth (e.g., for monitors acquired by JNI MonitorEnter).The thread must be suspended, and the returned information is relevant only while the thread is suspended. Requires canGetMonitorFrameInfo capability - see CapabilitiesNew.

Since JDWP version 1.6.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *owned* | The number of owned monitors |
| Repeated *owned* times: | | | |
| | tagged-objectID | *monitor* | An owned monitor |

| | | | |
|---|---|---|---|
| | int | *stack_depth* | Stack depth location where monitor was acquired |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

### ForceEarlyReturn Command (14)

Force a method to return before it reaches a return statement.

The method which will return early is referred to as the called method. The called method is the current method (as defined by the Frames section in *The Java™ Virtual Machine Specification*) for the specified thread at the time this command is received.

The specified thread must be suspended. The return occurs when execution of Java programming language code is resumed on this thread. Between sending this command and resumption of thread execution, the state of the stack is undefined.

No further instructions are executed in the called method. Specifically, finally blocks are not executed. Note: this can cause inconsistent states in the application.

A lock acquired by calling the called method (if it is a synchronized method) and locks acquired by entering synchronized blocks within the called method are released. Note: this does not apply to JNI locks or java.util.concurrent.locks locks.

Events, such as MethodExit, are generated as they would be in a normal return.

The called method must be a non-native Java programming language method. Forcing return on a thread with only one frame on the stack causes the thread to exit when resumed.

For void methods, the value must be a void value. For methods that return primitive values, the value's type must match the return type exactly. For object values, there must be a widening reference conversion from the value's type to the return type type and the return type must be loaded.

Since JDWP version 1.6. Requires canForceEarlyReturn capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |
| value | *value* | The value to return. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | Thread or value is not a known ID. |
| THREAD_NOT_SUSPENDED | If the specified thread has not been suspended by an event. |
| THREAD_NOT_ALIVE | Thread has not been started or is now dead. |
| OPAQUE_FRAME | Attempted to return early from a frame corresponding to a native method. Or the implementation is unable to provide this functionality on this frame. |
| NO_MORE_FRAMES | There are no more Java or JNI frames on the call stack. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| TYPE_MISMATCH | Value is not an appropriate type for the return value of the method. |
| VM_DEAD | The virtual machine is not running. |

## ThreadGroupReference Command Set (12)

### Name Command (1)

Returns the thread group name.

Out Data

| | | |
|---|---|---|
| threadGroupID | *group* | The thread group object ID. |

Reply Data

| | | |
|---|---|---|
| string | *groupName* | The thread group's name. |

Error Data

| | |
|---|---|
| INVALID_THREAD_GROUP | Thread group invalid. |
| INVALID_OBJECT | group is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Parent Command (2)

Returns the thread group, if any, which contains a given thread group.

Out Data

| | | |
|---|---|---|
| threadGroupID | *group* | The thread group object ID. |

Reply Data

| | | |
|---|---|---|
| threadGroupID | *parentGroup* | The parent thread group object, or null if the given thread group is a top-level thread group |

Error Data

| | |
|---|---|
| INVALID_THREAD_GROUP | Thread group invalid. |
| INVALID_OBJECT | group is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

### Children Command (3)

Returns the live threads and active thread groups directly contained in this thread group. Threads and thread groups in child thread groups are not included. A thread is alive if it has been started and has not yet been stopped. See java.lang.ThreadGroup for information about active ThreadGroups.

Out Data

| | | |
|---|---|---|
| threadGroupID | *group* | The thread group object ID. |

Reply Data

| | | | |
|---|---|---|---|
| int | | *childThreads* | The number of live child threads. |
| Repeated *childThreads* times: | | | |
| | threadID | *childThread* | A direct child thread ID. |
| int | | *childGroups* | The number of active child thread groups. |
| Repeated *childGroups* times: | | | |
| | threadGroupID | *childGroup* | A direct child thread group ID. |

Error Data

| | |
|---|---|
| INVALID_THREAD_GROUP | Thread group invalid. |
| INVALID_OBJECT | group is not a known ID. |
| VM_DEAD | The virtual machine is not running. |

## ArrayReference Command Set (13)

### Length Command (1)

Returns the number of components in a given array.

Out Data

| | | |
|---|---|---|
| arrayID | *arrayObject* | The array object ID. |

Reply Data

| | | |
|---|---|---|
| int | *arrayLength* | The length of the array. |

Error Data

| | |
|---|---|
| INVALID_OBJECT | arrayObject is not a known ID. |
| INVALID_ARRAY | The array is invalid. |
| VM_DEAD | The virtual machine is not running. |

### GetValues Command (2)

Returns a range of array components. The specified range must be within the bounds of the array.

Out Data

| | | |
|---|---|---|
| arrayID | *arrayObject* | The array object ID. |
| int | *firstIndex* | The first index to retrieve. |
| int | *length* | The number of components to retrieve. |

Reply Data

| | | |
|---|---|---|
| arrayregion | *values* | The retrieved values. If the values are objects, they are tagged-values; otherwise, they are untagged-values |

Error Data

| | |
|---|---|
| INVALID_LENGTH | If index is beyond the end of this array. |
| INVALID_OBJECT | arrayObject is not a known ID. |
| INVALID_ARRAY | The array is invalid. |
| VM_DEAD | The virtual machine is not running. |

### SetValues Command (3)

Sets a range of array components. The specified range must be within the bounds of the array. For primitive values, each value's type must match the array component type exactly. For object values, there must be a widening reference conversion from the value's type to thearray component type and the array component type must be loaded.

Out Data

| | | |
|---|---|---|
| arrayID | *arrayObject* | The array object ID. |
| int | *firstIndex* | The first index to set. |
| int | *values* | The number of values to set. |
| Repeated *values* times: | | |
| untagged-value | *value* | A value to set. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_LENGTH | If index is beyond the end of this array. |
| INVALID_OBJECT | arrayObject is not a known ID. |
| INVALID_ARRAY | The array is invalid. |
| VM_DEAD | The virtual machine is not running. |

## ClassLoaderReference Command Set (14)

### VisibleClasses Command (1)

Returns a list of all classes which this class loader has been requested to load. This class loader is considered to be an *initiating* class loader for each class in the returned list. The list contains each reference type defined by this loader and any types for which loading was delegated by this class loader to another class loader.

The visible class list has useful properties with respect to the type namespace. A particular type name will occur at most once in the list. Each field or variable declared with that type name in a class defined by this class loader must be resolved to that single type.

No ordering of the returned list is guaranteed.

Out Data

| | | |
|---|---|---|
| classLoaderID | *classLoaderObject* | The class loader object ID. |

Reply Data

| | | |
|---|---|---|
| int | *classes* | The number of visible classes. |

| Repeated *classes* times: | | | |
|---|---|---|---|
| | byte | *refTypeTag* | Kind of following reference type. |
| | referenceTypeID | *typeID* | A class visible to this class loader. |

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_CLASS_LOADER | The class loader is invalid. |
| VM_DEAD | The virtual machine is not running. |

## EventRequest Command Set (15)

### Set Command (1)

Set an event request. When the event described by this request occurs, an event is sent from the target VM. If an event occurs that has not been requested then it is not sent from the target VM. The two exceptions to this are the VM Start Event and the VM Death Event which are automatically generated events - see Composite Command for further details.

Out Data

| | | |
|---|---|---|
| byte | *eventKind* | Event kind to request. See JDWP.EventKind for a complete list of events that can be requested; some events may require a capability in order to be requested. |
| byte | *suspendPolicy* | What threads are suspended when this event occurs? Note that the order of events and command replies accurately reflects the order in which threads are suspended and resumed. For example, if a VM-wide resume is processed before an event occurs which suspends the VM, the reply to the resume command will be written to the transport before the suspending event. |
| int | *modifiers* | Constraints used to control the number of generated events.Modifiers specify additional tests that an event must satisfy before it is placed in the event queue. Events are filtered by applying each modifier to an event in the order they are specified in this collection Only events that satisfy all modifiers are reported. A value of 0 means there are no modifiers in the request.<br><br>Filtering can improve debugger performance dramatically byreducing the amount of event |

| | | |
|---|---|---|
| | | traffic sent from the target VM to the debugger VM. |
| Repeated *modifiers* times: | | |
| byte | *modKind* | Modifier kind |
| Case Count - if *modKind* is 1: | | Limit the requested event to be reported at most once after a given number of occurrences. The event is not reported the first `count - 1` times this filter is reached. To request a one-off event, call this method with a count of 1.<br><br>Once the count reaches 0, any subsequent filters in this request are applied. If none of those filters cause the event to be suppressed, the event is reported. Otherwise, the event is not reported. In either case subsequent events are never reported for this request. This modifier can be used with any event kind. |
| int | *count* | Count before event. One for one-off. |
| Case Conditional - if *modKind* is 2: | | Conditional on expression |
| int | *exprID* | For the future |
| Case ThreadOnly - if *modKind* is 3: | | Restricts reported events to those in the given thread. This modifier can be used with any event kind except for class unload. |
| threadID | *thread* | Required thread |
| Case ClassOnly - if *modKind* is 4: | | For class prepare events, restricts the events generated by this request to be the preparation of the given reference type and any subtypes. For monitor wait and waited events, restricts the events generated by this request to those whose monitor object is of the given reference type or any of its subtypes. For other events, restricts the events generated by this request to those whose location is in the given reference type or any of its subtypes. An event will be generated for any location in a reference type that can be safely cast to the given reference type. This modifier can be used with any event kind except class unload, thread start, and thread end. |
| referenceTypeID | *clazz* | Required class |

| | | | |
|---|---|---|---|
| Case ClassMatch - if *modKind* is 5: | | | Restricts reported events to those for classes whose name matches the given restricted regular expression. For class prepare events, the prepared class name is matched. For class unload events, the unloaded class name is matched. For monitor wait and waited events, the name of the class of the monitor object is matched. For other events, the class name of the event's location is matched. This modifier can be used with any event kind except thread start and thread end. |
| | string | *classPattern* | Required class pattern. Matches are limited to exact matches of the given class pattern and matches of patterns that begin or end with '*'; for example, "*.Foo" or "java.*". |
| Case ClassExclude - if *modKind* is 6: | | | Restricts reported events to those for classes whose name does not match the given restricted regular expression. For class prepare events, the prepared class name is matched. For class unload events, the unloaded class name is matched. For monitor wait and waited events, the name of the class of the monitor object is matched. For other events, the class name of the event's location is matched. This modifier can be used with any event kind except thread start and thread end. |
| | string | *classPattern* | Disallowed class pattern. Matches are limited to exact matches of the given class pattern and matches of patterns that begin or end with '*'; for example, "*.Foo" or "java.*". |
| Case LocationOnly - if *modKind* is 7: | | | Restricts reported events to those that occur at the given location. This modifier can be used with breakpoint, field access, field modification, step, and exception event kinds. |
| | location | *loc* | Required location |
| Case ExceptionOnly - if *modKind* is 8: | | | Restricts reported exceptions by their class and whether they are caught or uncaught. This modifier can be used with exception event kinds only. |
| | referenceTypeID | *exceptionOrNull* | Exception to report. Null (0) means report exceptions of all types. A non-null type restricts the reported exception events to exceptions of the given type or any of its subtypes. |

| | | | |
|---|---|---|---|
| | boolean | *caught* | Report caught exceptions |
| | boolean | *uncaught* | Report uncaught exceptions. Note that it is not always possible to determine whether an exception is caught or uncaught at the time it is thrown. See the exception event catch location under composite events for more information. |
| Case FieldOnly - if *modKind* is 9: | | | Restricts reported events to those that occur for a given field. This modifier can be used with field access and field modification event kinds only. |
| | referenceTypeID | *declaring* | Type in which field is declared. |
| | fieldID | *fieldID* | Required field |
| Case Step - if *modKind* is 10: | | | Restricts reported step events to those which satisfy depth and size constraints. This modifier can be used with step event kinds only. |
| | threadID | *thread* | Thread in which to step |
| | int | *size* | size of each step. See JDWP.StepSize |
| | int | *depth* | relative call stack limit. See JDWP.StepDepth |
| Case InstanceOnly - if *modKind* is 11: | | | Restricts reported events to those whose active 'this' object is the given object. Match value is the null object for static methods. This modifier can be used with any event kind except class prepare, class unload, thread start, and thread end. Introduced in JDWP version 1.4. |
| | objectID | *instance* | Required 'this' object |
| Case SourceNameMatch - if *modKind* is 12: | | | Restricts reported class prepare events to those for reference types which have a source name which matches the given restricted regular expression. The source names are determined by the reference type's SourceDebugExtension. This modifier can only be used with class prepare events. Since JDWP version 1.6. Requires the canUseSourceNameFilters capability - see CapabilitiesNew. |
| | string | *sourceNamePattern* | Required source name pattern. Matches are limited to exact matches of the given pattern and matches of patterns that begin or end with '*'; for example, "*.Foo" or "java.*". |

Reply Data

| | | |
|---|---|---|
| int | *requestID* | ID of created request |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_CLASS | Invalid class. |
| INVALID_STRING | The string is invalid. |
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_COUNT | The count is invalid. |
| INVALID_FIELDID | Invalid field. |
| INVALID_METHODID | Invalid method. |
| INVALID_LOCATION | Invalid location. |
| INVALID_EVENT_TYPE | The specified event type id is not recognized. |
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

### Clear Command (2)

Clear an event request. See JDWP.EventKind for a complete list of events that can be cleared. Only the event request matching the specified event kind and requestID is cleared. If there isn't a matching event request the command is a no-op and does not result in an error. Automatically generated events do not have a corresponding event request and may not be cleared using this command.

Out Data

| | | |
|---|---|---|
| byte | *eventKind* | Event kind to clear |
| int | *requestID* | ID of request to clear |

Reply Data

(None)

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |
| INVALID_EVENT_TYPE | The specified event type id is not recognized. |

### ClearAllBreakpoints Command (3)

Removes all set breakpoints, a no-op if there are no breakpoints set.

Out Data

(None)

Reply Data

(None)

Error Data

| | |
|---|---|
| VM_DEAD | The virtual machine is not running. |

## StackFrame Command Set (16)

### GetValues Command (1)

Returns the value of one or more local variables in a given frame. Each variable must be visible at the frame's code index. Even if local variable information is not available, values can be retrieved if the front-end is able to determine the correct local variable index. (Typically, this index can be determined for method arguments from the method signature without access to the local variable table information.)

Out Data

| | | | |
|---|---|---|---|
| threadID | | *thread* | The frame's thread. |
| frameID | | *frame* | The frame ID. |
| int | | *slots* | The number of values to get. |
| Repeated *slots* times: | | | |
| | int | *slot* | The local variable's index in the frame. |
| | byte | *sigbyte* | A tag identifying the type of the variable |

Reply Data

| | | | |
|---|---|---|---|
| int | | *values* | The number of values retrieved, always equal to slots, the number of values to get. |
| Repeated *values* times: | | | |
| | value | *slotValue* | The value of the local variable. |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |

| | |
|---|---|
| INVALID_FRAMEID | Invalid jframeID. |
| INVALID_SLOT | Invalid slot. |
| VM_DEAD | The virtual machine is not running. |

### SetValues Command (2)

Sets the value of one or more local variables. Each variable must be visible at the current frame code index. For primitive values, the value's type must match the variable's type exactly. For object values, there must be a widening reference conversion from the value's type to thevariable's type and the variable's type must be loaded.

Even if local variable information is not available, values can be set, if the front-end is able to determine the correct local variable index. (Typically, thisindex can be determined for method arguments from the method signature without access to the local variable table information.)

Out Data

| | | | |
|---|---|---|---|
| threadID | | *thread* | The frame's thread. |
| frameID | | *frame* | The frame ID. |
| int | | *slotValues* | The number of values to set. |
| Repeated *slotValues* times: | | | |
| | int | *slot* | The slot ID. |
| | value | *slotValue* | The value to set. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_FRAMEID | Invalid jframeID. |
| VM_DEAD | The virtual machine is not running. |

### ThisObject Command (3)

Returns the value of the 'this' reference for this frame. If the frame's method is static or native, the reply will contain the null object reference.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The frame's thread. |
| frameID | *frame* | The frame ID. |

Reply Data

| | | |
|---|---|---|
| tagged-objectID | *objectThis* | The 'this' object for this frame. |

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| INVALID_FRAMEID | Invalid jframeID. |
| VM_DEAD | The virtual machine is not running. |

### PopFrames Command (4)

Pop the top-most stack frames of the thread stack, up to, and including 'frame'. The thread must be suspended to perform this command. The top-most stack frames are discarded and the stack frame previous to 'frame' becomes the current frame. The operand stack is restored -- the argument values are added back and if the invoke was not `invokestatic`, `objectref` is added back as well. The Java virtual machine program counter is restored to the opcode of the invoke instruction.

Since JDWP version 1.4. Requires canPopFrames capability - see CapabilitiesNew.

Out Data

| | | |
|---|---|---|
| threadID | *thread* | The thread object ID. |
| frameID | *frame* | The frame ID. |

Reply Data

(None)

Error Data

| | |
|---|---|
| INVALID_THREAD | Passed thread is null, is not a valid thread or has exited. |
| INVALID_OBJECT | thread is not a known ID. |
| INVALID_FRAMEID | Invalid jframeID. |
| THREAD_NOT_SUSPENDED | If the specified thread has not been suspended by an event. |
| NO_MORE_FRAMES | There are no more Java or JNI frames on the call stack. |
| INVALID_FRAMEID | Invalid jframeID. |

| | |
|---|---|
| NOT_IMPLEMENTED | The functionality is not implemented in this virtual machine. |
| VM_DEAD | The virtual machine is not running. |

## ClassObjectReference Command Set (17)

### ReflectedType Command (1)

Returns the reference type reflected by this class object.

Out Data

| | | |
|---|---|---|
| classObjectID | *classObject* | The class object. |

Reply Data

| | | |
|---|---|---|
| byte | *refTypeTag* | Kind of following reference type. |
| referenceTypeID | *typeID* | reflected reference type |

Error Data

| | |
|---|---|
| INVALID_OBJECT | If this reference type has been unloaded and garbage collected. |
| VM_DEAD | The virtual machine is not running. |

## Event Command Set (64)

### Composite Command (100)

Several events may occur at a given time in the target VM. For example, there may be more than one breakpoint request for a given location or you might single step to the same location as a breakpoint request. These events are delivered together as a composite event. For uniformity, a composite event is always used to deliver events, even if there is only one event to report.

The events that are grouped in a composite event are restricted in the following ways:

- Only with other thread start events for the same thread:
  - Thread Start Event
- Only with other thread death events for the same thread:
  - Thread Death Event
- Only with other class prepare events for the same class:
  - Class Prepare Event
- Only with other class unload events for the same class:
  - Class Unload Event
- Only with other access watchpoint events for the same field access:
  - Access Watchpoint Event
- Only with other modification watchpoint events for the same field modification:
  - Modification Watchpoint Event
- Only with other Monitor contended enter events for the same monitor object:
  - Monitor Contended Enter Event

- Only with other Monitor contended entered events for the same monitor object:
  - Monitor Contended Entered Event
- Only with other Monitor wait events for the same monitor object:
  - Monitor Wait Event
- Only with other Monitor waited events for the same monitor object:
  - Monitor Waited Event
- Only with other ExceptionEvents for the same exception occurrance:
  - ExceptionEvent
- Only with other members of this group, at the same location and in the same thread:
  - Breakpoint Event
  - Step Event
  - Method Entry Event
  - Method Exit Event

The VM Start Event and VM Death Event are automatically generated events. This means they do not need to be requested using the EventRequest.Set command. The VM Start event signals the completion of VM initialization. The VM Death event signals the termination of the VM.If there is a debugger connected at the time when an automatically generated event occurs it is sent from the target VM. Automatically generated events may also be requested using the EventRequest.Set command and thus multiple events of the same event kind will be sent from the target VM when an event occurs.Automatically generated events are sent with the requestID field in the Event Data set to 0. The value of the suspendPolicy field in the Event Data depends on the event. For the automatically generated VM Start Event the value of suspendPolicy is not defined and is therefore implementation or configuration specific. In the Sun implementation, for example, the suspendPolicy is specified as an option to the JDWP agent at launch-time.The automatically generated VM Death Event will have the suspendPolicy set to NONE.

Event Data

| | | |
|---|---|---|
| byte | *suspendPolicy* | Which threads where suspended by this composite event? |
| int | *events* | Events in set. |
| Repeated *events* times: | | |
| byte | *eventKind* | Event kind selector |
| Case VMStart - if *eventKind* is JDWP.EventKind.VM_START: | | Notification of initialization of a target VM. This event is received before the main thread is started and before any application code has been executed. Before this event occurs a significant amount of system code has executed and a number of system classes have been loaded. This event is always generated by the target VM, even if not explicitly requested. |

| | | | |
|---|---|---|---|
| | int | *requestID* | Request that generated event (or 0 if this event is automatically generated. |
| | threadID | *thread* | Initial thread |
| Case SingleStep - if *eventKind* is JDWP.EventKind.SINGLE_STEP: | | | Notification of step completion in the target VM. The step event is generated before the code at its location is executed. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Stepped thread |
| | location | *location* | Location stepped to |
| Case Breakpoint - if *eventKind* is JDWP.EventKind.BREAKPOINT: | | | Notification of a breakpoint in the target VM. The breakpoint event is generated before the code at its location is executed. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which hit breakpoint |
| | location | *location* | Location hit |
| Case MethodEntry - if *eventKind* is JDWP.EventKind.METHOD_ENTRY: | | | Notification of a method invocation in the target VM. This event is generated before any code in the invoked method has executed. Method entry events are generated for both native and non-native methods.<br><br>In some VMs method entry events can occur for a particular thread before its thread start event occurs if methods are called as part of the thread's initialization. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which entered method |
| | location | *location* | The initial executable location in the method. |

| | | | |
|---|---|---|---|
| Case MethodExit - if *eventKind* is JDWP.EventKind.METHOD_EXIT: | | | Notification of a method return in the target VM. This event is generated after all code in the method has executed, but the location of this event is the last executed location in the method. Method exit events are generated for both native and non-native methods. Method exit events are not generated if the method terminates with a thrown exception. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which exited method |
| | location | *location* | Location of exit |
| Case MethodExitWithReturnValue - if *eventKind* is JDWP.EventKind.METHOD_EXIT_WITH_RETURN_VALUE: | | | Notification of a method return in the target VM. This event is generated after all code in the method has executed, but the location of this event is the last executed location in the method. Method exit events are generated for both native and non-native methods. Method exit events are not generated if the method terminates with a thrown exception.<br><br>Since JDWP version 1.6. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which exited method |
| | location | *location* | Location of exit |
| | value | *value* | Value that will be returned by the method |
| Case MonitorContendedEnter - if *eventKind* is JDWP.EventKind.MONITOR_CONTENDED_ENTER: | | | Notification that a thread in the target VM is attempting to enter a monitor that is already acquired by another thread. Requires canRequestMonitorEvents capability - see CapabilitiesNew. |

| | | | |
|---|---|---|---|
| | | | Since JDWP version 1.6. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which is trying to enter the monitor |
| | tagged-objectID | *object* | Monitor object reference |
| | location | *location* | Location of contended monitor enter |
| Case MonitorContendedEntered - if *eventKind* is JDWP.EventKind.MONITOR_CONTENDED_ENTERED: | | | Notification of a thread in the target VM is entering a monitor after waiting for it to be released by another thread. Requires canRequestMonitorEvents capability - see CapabilitiesNew.<br><br>Since JDWP version 1.6. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which entered monitor |
| | tagged-objectID | *object* | Monitor object reference |
| | location | *location* | Location of contended monitor enter |
| Case MonitorWait - if *eventKind* is JDWP.EventKind.MONITOR_WAIT: | | | Notification of a thread about to wait on a monitor object. Requires canRequestMonitorEvents capability - see CapabilitiesNew.<br><br>Since JDWP version 1.6. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which is about to wait |
| | tagged-objectID | *object* | Monitor object reference |
| | location | *location* | Location at which the wait will occur |
| | long | *timeout* | Thread wait time in milliseconds |
| Case MonitorWaited - if *eventKind* is JDWP.EventKind.MONITOR_WAITED: | | | Notification that a thread in the target VM has finished waiting on |

| | | | |
|---|---|---|---|
| | | | Requires canRequestMonitorEvents capability - see CapabilitiesNew. a monitor object.<br><br>Since JDWP version 1.6. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread which waited |
| | tagged-objectID | *object* | Monitor object reference |
| | location | *location* | Location at which the wait occured |
| | boolean | *timed_out* | True if timed out |
| Case Exception - if *eventKind* is JDWP.EventKind.EXCEPTION: | | | Notification of an exception in the target VM. If the exception is thrown from a non-native method, the exception event is generated at the location where the exception is thrown. If the exception is thrown from a native method, the exception event is generated at the first non-native location reached after the exception is thrown. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Thread with exception |
| | location | *location* | Location of exception throw (or first non-native location after throw if thrown from a native method) |
| | tagged-objectID | *exception* | Thrown exception |
| | location | *catchLocation* | Location of catch, or 0 if not caught. An exception is considered to be caught if, at the point of the throw, the current location is dynamically enclosed in a try statement that handles the exception. (See the JVM specification for details). If there is such a try statement, the catch location is the first location in the appropriate catch clause. |

| | | | |
|---|---|---|---|
| | | | If there are native methods in the call stack at the time of the exception, there are important restrictions to note about the returned catch location. In such cases, it is not possible to predict whether an exception will be handled by some native method on the call stack. Thus, it is possible that exceptions considered uncaught here will, in fact, be handled by a native method and not cause termination of the target VM. Furthermore, it cannot be assumed that the catch location returned here will ever be reached by the throwing thread. If there is a native frame between the current location and the catch location, the exception might be handled and cleared in that native method instead.<br><br>Note that compilers can generate try-catch blocks in some cases where they are not explicit in the source code; for example, the code generated for `synchronized` and `finally` blocks can contain implicit try-catch blocks. If such an implicitly generated try-catch is present on the call stack at the time of the throw, the exception will be considered caught even though it appears to be uncaught from examination of the source code. |
| Case ThreadStart - if *eventKind* is JDWP.EventKind.THREAD_START: | | | Notification of a new running thread in the target VM. The new thread can be the result of a call to `java.lang.Thread.start` or the result of attaching a new thread to the VM though JNI. The notification is generated by the new thread some time before its execution starts. Because of this timing, it is possible to receive other events for |

| | | | |
|---|---|---|---|
| | | | the thread before this event is received. (Notably, Method Entry Events and Method Exit Events might occur during thread initialization. It is also possible for the VirtualMachine AllThreads command to return a thread before its thread start event is received.<br><br>Note that this event gives no information about the creation of the thread object which may have happened much earlier, depending on the VM being debugged. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Started thread |
| Case ThreadDeath - if *eventKind* is JDWP.EventKind.THREAD_DEATH: | | | Notification of a completed thread in the target VM. The notification is generated by the dying thread before it terminates. Because of this timing, it is possible for {@link VirtualMachine#allThreads} to return this thread after this event is received.<br><br>Note that this event gives no information about the lifetime of the thread object. It may or may not be collected soon depending on what references exist in the target VM. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Ending thread |
| Case ClassPrepare - if *eventKind* is JDWP.EventKind.CLASS_PREPARE: | | | Notification of a class prepare in the target VM. See the JVM specification for a definition of class preparation. Class prepare events are not generated for primtiive classes (for example, java.lang.Integer.TYPE). |
| | int | *requestID* | Request that generated event |

| | | | |
|---|---|---|---|
| | threadID | *thread* | Preparing thread. In rare cases, this event may occur in a debugger system thread within the target VM. Debugger threads take precautions to prevent these events, but they cannot be avoided under some conditions, especially for some subclasses of java.lang.Error. If the event was generated by a debugger system thread, the value returned by this method is null, and if the requested suspend policy for the event was EVENT_THREAD all threads will be suspended instead, and the composite event's suspend policy will reflect this change.<br><br>Note that the discussion above does not apply to system threads created by the target VM during its normal (non-debug) operation. |
| | byte | *refTypeTag* | Kind of reference type. See JDWP.TypeTag |
| | referenceTypeID | *typeID* | Type being prepared |
| | string | *signature* | Type signature |
| | int | *status* | Status of type. See JDWP.ClassStatus |
| Case ClassUnload - if *eventKind* is JDWP.EventKind.CLASS_UNLOAD: | | | Notification of a class unload in the target VM.<br><br>There are severe constraints on the debugger back-end during garbage collection, so unload information is greatly limited. |
| | int | *requestID* | Request that generated event |
| | string | *signature* | Type signature |
| Case FieldAccess - if *eventKind* is JDWP.EventKind.FIELD_ACCESS: | | | Notification of a field access in the target VM. Field modifications are not considered field accesses. |

| | | | |
|---|---|---|---|
| | | | Requires canWatchFieldAccess capability - see CapabilitiesNew. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Accessing thread |
| | location | *location* | Location of access |
| | byte | *refTypeTag* | Kind of reference type. See JDWP.TypeTag |
| | referenceTypeID | *typeID* | Type of field |
| | fieldID | *fieldID* | Field being accessed |
| | tagged-objectID | *object* | Object being accessed (null=0 for statics |
| Case FieldModification - if *eventKind* is JDWP.EventKind.FIELD_MODIFICATION: | | | Notification of a field modification in the target VM. Requires canWatchFieldModification capability - see CapabilitiesNew. |
| | int | *requestID* | Request that generated event |
| | threadID | *thread* | Modifying thread |
| | location | *location* | Location of modify |
| | byte | *refTypeTag* | Kind of reference type. See JDWP.TypeTag |
| | referenceTypeID | *typeID* | Type of field |
| | fieldID | *fieldID* | Field being modified |
| | tagged-objectID | *object* | Object being modified (null=0 for statics |
| | value | *valueToBe* | Value to be assigned |
| Case VMDeath - if *eventKind* is JDWP.EventKind.VM_DEATH: | | | |
| | int | *requestID* | Request that generated event |

## Error Constants

| | | |
|---|---|---|
| NONE | 0 | No error has occurred. |
| INVALID_THREAD | 10 | Passed thread is null, is not a valid thread or has exited. |
| INVALID_THREAD_GROUP | 11 | Thread group invalid. |
| INVALID_PRIORITY | 12 | Invalid priority. |
| THREAD_NOT_SUSPENDED | 13 | If the specified thread has not been suspended by an event. |
| THREAD_SUSPENDED | 14 | Thread already suspended. |
| THREAD_NOT_ALIVE | 15 | Thread has not been started or is now dead. |
| INVALID_OBJECT | 20 | If this reference type has been unloaded and garbage collected. |
| INVALID_CLASS | 21 | Invalid class. |
| CLASS_NOT_PREPARED | 22 | Class has been loaded but not yet prepared. |
| INVALID_METHODID | 23 | Invalid method. |
| INVALID_LOCATION | 24 | Invalid location. |
| INVALID_FIELDID | 25 | Invalid field. |
| INVALID_FRAMEID | 30 | Invalid jframeID. |
| NO_MORE_FRAMES | 31 | There are no more Java or JNI frames on the call stack. |
| OPAQUE_FRAME | 32 | Information about the frame is not available. |
| NOT_CURRENT_FRAME | 33 | Operation can only be performed on current frame. |
| TYPE_MISMATCH | 34 | The variable is not an appropriate type for the function used. |
| INVALID_SLOT | 35 | Invalid slot. |
| DUPLICATE | 40 | Item already set. |

| | | |
|---|---|---|
| NOT_FOUND | 41 | Desired element not found. |
| INVALID_MONITOR | 50 | Invalid monitor. |
| NOT_MONITOR_OWNER | 51 | This thread doesn't own the monitor. |
| INTERRUPT | 52 | The call has been interrupted before completion. |
| INVALID_CLASS_FORMAT | 60 | The virtual machine attempted to read a class file and determined that the file is malformed or otherwise cannot be interpreted as a class file. |
| CIRCULAR_CLASS_DEFINITION | 61 | A circularity has been detected while initializing a class. |
| FAILS_VERIFICATION | 62 | The verifier detected that a class file, though well formed, contained some sort of internal inconsistency or security problem. |
| ADD_METHOD_NOT_IMPLEMENTED | 63 | Adding methods has not been implemented. |
| SCHEMA_CHANGE_NOT_IMPLEMENTED | 64 | Schema change has not been implemented. |
| INVALID_TYPESTATE | 65 | The state of the thread has been modified, and is now inconsistent. |
| HIERARCHY_CHANGE_NOT_IMPLEMENTED | 66 | A direct superclass is different for the new class version, or the set of directly implemented interfaces is different and canUnrestrictedlyRedefineClasses is false. |
| DELETE_METHOD_NOT_IMPLEMENTED | 67 | The new class version does not declare a method declared in the old class version and canUnrestrictedlyRedefineClasses is false. |
| UNSUPPORTED_VERSION | 68 | A class file has a version number not supported by this VM. |

| | | |
|---|---|---|
| NAMES_DONT_MATCH | 69 | The class name defined in the new class file is different from the name in the old class object. |
| CLASS_MODIFIERS_CHANGE_NOT_IMPLEMENTED | 70 | The new class version has different modifiers and and canUnrestrictedlyRedefineClasses is false. |
| METHOD_MODIFIERS_CHANGE_NOT_IMPLEMENTED | 71 | A method in the new class version has different modifiers than its counterpart in the old class version and and canUnrestrictedlyRedefineClasses is false. |
| NOT_IMPLEMENTED | 99 | The functionality is not implemented in this virtual machine. |
| NULL_POINTER | 100 | Invalid pointer. |
| ABSENT_INFORMATION | 101 | Desired information is not available. |
| INVALID_EVENT_TYPE | 102 | The specified event type id is not recognized. |
| ILLEGAL_ARGUMENT | 103 | Illegal argument. |
| OUT_OF_MEMORY | 110 | The function needed to allocate memory and no more memory was available for allocation. |
| ACCESS_DENIED | 111 | Debugging has not been enabled in this virtual machine. JVMTI cannot be used. |
| VM_DEAD | 112 | The virtual machine is not running. |
| INTERNAL | 113 | An unexpected internal error has occurred. |
| UNATTACHED_THREAD | 115 | The thread being used to call this function is not attached to the virtual machine. Calls must be made from attached threads. |

| | | |
|---|---|---|
| INVALID_TAG | 500 | object type id or class tag. |
| ALREADY_INVOKING | 502 | Previous invoke not complete. |
| INVALID_INDEX | 503 | Index is invalid. |
| INVALID_LENGTH | 504 | The length is invalid. |
| INVALID_STRING | 506 | The string is invalid. |
| INVALID_CLASS_LOADER | 507 | The class loader is invalid. |
| INVALID_ARRAY | 508 | The array is invalid. |
| TRANSPORT_LOAD | 509 | Unable to load the transport. |
| TRANSPORT_INIT | 510 | Unable to initialize the transport. |
| NATIVE_METHOD | 511 | |
| INVALID_COUNT | 512 | The count is invalid. |

## EventKind Constants

| | | |
|---|---|---|
| SINGLE_STEP | 1 | |
| BREAKPOINT | 2 | |
| FRAME_POP | 3 | |
| EXCEPTION | 4 | |
| USER_DEFINED | 5 | |
| THREAD_START | 6 | |
| THREAD_DEATH | 7 | |
| THREAD_END | 7 | obsolete - was used in jvmdi |
| CLASS_PREPARE | 8 | |
| CLASS_UNLOAD | 9 | |
| CLASS_LOAD | 10 | |
| FIELD_ACCESS | 20 | |

| | | |
|---|---|---|
| FIELD_MODIFICATION | 21 | |
| EXCEPTION_CATCH | 30 | |
| METHOD_ENTRY | 40 | |
| METHOD_EXIT | 41 | |
| METHOD_EXIT_WITH_RETURN_VALUE | 42 | |
| MONITOR_CONTENDED_ENTER | 43 | |
| MONITOR_CONTENDED_ENTERED | 44 | |
| MONITOR_WAIT | 45 | |
| MONITOR_WAITED | 46 | |
| VM_START | 90 | |
| VM_INIT | 90 | obsolete - was used in jvmdi |
| VM_DEATH | 99 | |
| VM_DISCONNECTED | 100 | Never sent across JDWP |

### ThreadStatus Constants

| | | |
|---|---|---|
| ZOMBIE | 0 | |
| RUNNING | 1 | |
| SLEEPING | 2 | |
| MONITOR | 3 | |
| WAIT | 4 | |

### SuspendStatus Constants

| | | |
|---|---|---|
| SUSPEND_STATUS_SUSPENDED | 0x1 | |

### ClassStatus Constants

| | | |
|---|---|---|

| | | |
|---|---|---|
| VERIFIED | 1 | |
| PREPARED | 2 | |
| INITIALIZED | 4 | |
| ERROR | 8 | |

### TypeTag Constants

| | | |
|---|---|---|
| CLASS | 1 | ReferenceType is a class. |
| INTERFACE | 2 | ReferenceType is an interface. |
| ARRAY | 3 | ReferenceType is an array. |

### Tag Constants

| | | |
|---|---|---|
| ARRAY | 91 | '[' - an array object (objectID size). |
| BYTE | 66 | 'B' - a byte value (1 byte). |
| CHAR | 67 | 'C' - a character value (2 bytes). |
| OBJECT | 76 | 'L' - an object (objectID size). |
| FLOAT | 70 | 'F' - a float value (4 bytes). |
| DOUBLE | 68 | 'D' - a double value (8 bytes). |
| INT | 73 | 'I' - an int value (4 bytes). |
| LONG | 74 | 'J' - a long value (8 bytes). |
| SHORT | 83 | 'S' - a short value (2 bytes). |
| VOID | 86 | 'V' - a void value (no bytes). |
| BOOLEAN | 90 | 'Z' - a boolean value (1 byte). |
| STRING | 115 | 's' - a String object (objectID size). |
| THREAD | 116 | 't' - a Thread object (objectID size). |
| THREAD_GROUP | 103 | 'g' - a ThreadGroup object (objectID size). |

| | | |
|---|---|---|
| CLASS_LOADER | 108 | 'l' - a ClassLoader object (objectID size). |
| CLASS_OBJECT | 99 | 'c' - a class object object (objectID size). |

### StepDepth Constants

| | | |
|---|---|---|
| INTO | 0 | Step into any method calls that occur before the end of the step. |
| OVER | 1 | Step over any method calls that occur before the end of the step. |
| OUT | 2 | Step out of the current method. |

### StepSize Constants

| | | |
|---|---|---|
| MIN | 0 | Step by the minimum possible amount (often a bytecode instruction). |
| LINE | 1 | Step to the next source line unless there is no line number information in which case a MIN step is done instead. |

### SuspendPolicy Constants

| | | |
|---|---|---|
| NONE | 0 | Suspend no threads when this event is encountered. |
| EVENT_THREAD | 1 | Suspend the event thread when this event is encountered. |
| ALL | 2 | Suspend all threads when this event is encountered. |

### InvokeOptions Constants

The invoke options are a combination of zero or more of the following bit flags:

| | | |
|---|---|---|
| INVOKE_SINGLE_THREADED | 0x01 | otherwise, all threads started. |
| INVOKE_NONVIRTUAL | 0x02 | otherwise, normal virtual invoke (instance methods only) |