

Wikiprint Book

Title: Inspector user guide

Subject: Java Path Finder - projects/jpf-inspector/userguide

Version: 35

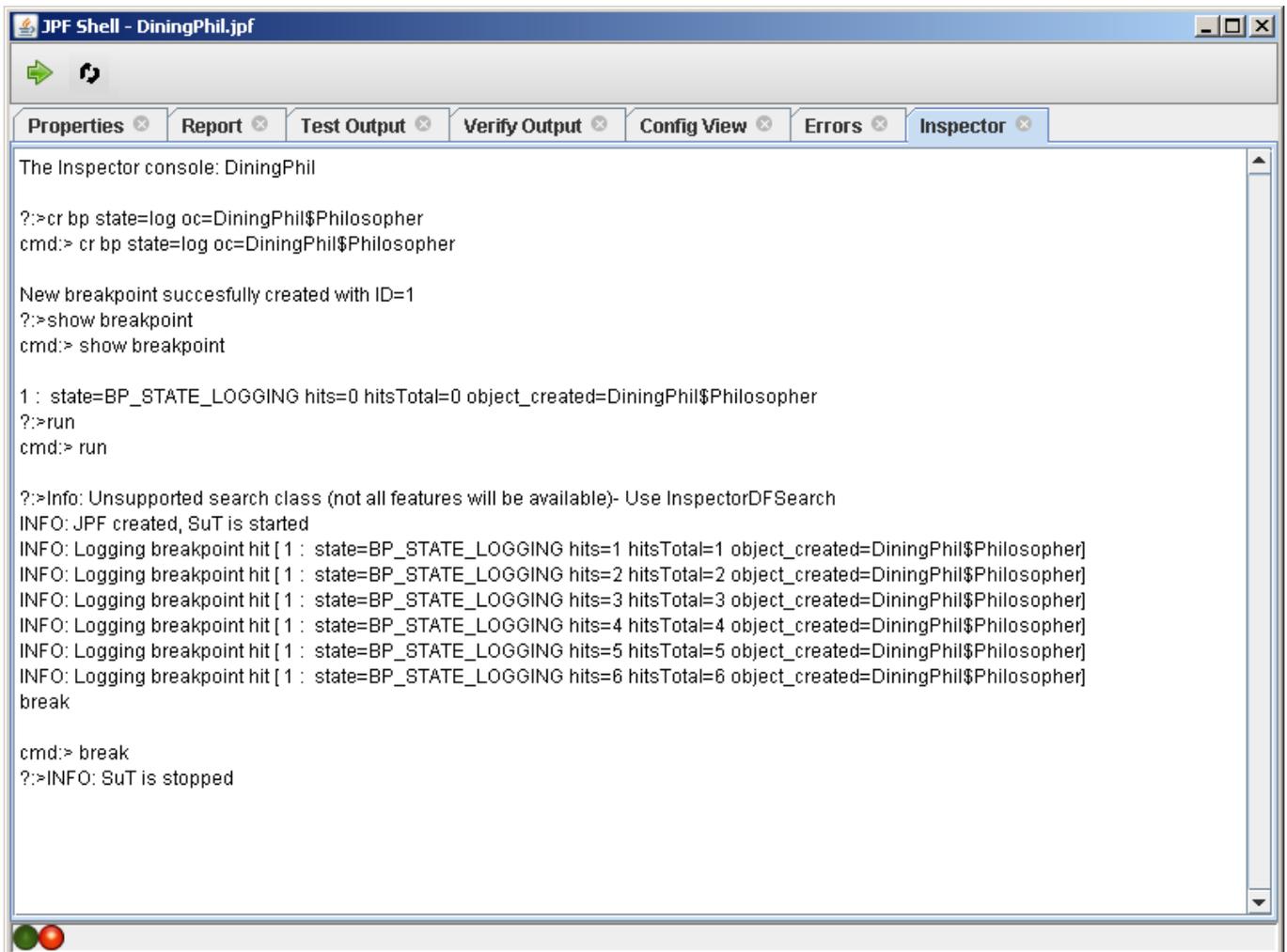
Date: 03/15/2013 05:38:57 PM

Table of Contents

Inspector user guide	3
Breakpoints	3
Breakpoint type specifications:	4
Single stepping	4
Program state inspection	4
Understanding output	5
Threads	5
Objects and Methods	5

Inspector user guide

Main user interface of the JPF Inspector is a textual console that is embedded into a new shell panel. The console behaves in an expected way - a user writes textual commands and inspector prints results (command output). Current interface is shown on the figure below.



Breakpoints

Inspector supports five basic commands related to breakpoints. Each command has several alternative names - the main one is always followed by shortcuts in brackets. Each breakpoint has its own unique ID that is assigned automatically - the ID is printed when the breakpoint is created and by the `show bp` command.

The `run [continue]` command starts the execution of JPF on the SuT (instead of the Verify button) or continues with the execution (after the breakpoint was hit). The `break` command stops JPF immediately.

The `show breakpoint [show bp]` command prints all existing breakpoints and their status.

The `delete breakpoint [del bp]` command removes a breakpoint. Example: `del bp 10` (removes the breakpoint with ID 10).

The `create breakpoint [cr bp]` command is used to create new breakpoints.

- Each breakpoint may have a name that is not processed by the Inspector (`name = Identifier`).
- Each breakpoint must have a state that is defined in the creating command. Possible options are:
 - `disable [dis]` - the breakpoint is deactivated and only the hit counts are computed
 - `log` - if the breakpoint hits, then a log message appears in the console, but the SuT is not stopped
 - `enable [en]` - execution of JPF is stopped when the breakpoint hits

Each breakpoint has 2 hit counters associated with it - a total hit counter and a path hit counter. A path hit counter behaves as hit counters in ordinary debuggers. Limits related to the associated action (log to console or stopping JPF) may be defined for each breakpoint. These limits are compared to the path hit counter. You may set either the lower or the upper bound, or both bounds. The keyword to use is `hit_count [hc]`. Some examples follow:

- `10 <= hit_count` - an action is executed for the 10th hit, 11th hit, and so on
- `hc < 50` - an action is executed for the initial 49 hits
- `3 <= hc <= 3` - an action is executed only for the third hit of the breakpoint

Previous settings are common to all breakpoints and they have to take place before the breakpoint type specification in the `create breakpoint` command.

Breakpoint type specifications:

- `property_violated [pv]` - the breakpoint hits when some property is violated by SuT
- `position [pos] = file name : line number` - the breakpoint hits when the first instruction on the given line is executed. Example: `cr bp state=log pos=*/JPF/JPF.java:122`.
Note: It is possible to use `*` in the `file name` argument as a wildcard for any part of the file path. This is especially useful to avoid writing directory (package) names - for example, you can write `*/ThreadStateError3.java:51` instead of `gov/nasa/jpf/rtembed/memory/ThreadStateError3.java`.
- `field_access [fa], field_read [fr] or field_write [fw] = class name : field name` - the breakpoint hits if the given field of any instance of the given class is accessed (read, written). Example: `create breakpoint state=enable field_write=Racer:d`.
- `method_invoke [mi] = class name : method name` - the breakpoint hits if the given method of any instance of the given class is invoked. Example: `create breakpoint state=enable method_invoke=MyList:add`.
- `instruction_type [inst_type, it] = any / invoke [inv] / return [ret] / field_access [fa] / field_read [fr] / field_write [fw] / condition [cond] / lock / array` - the breakpoint hits if an instruction of the given type is executed. The `condition` type hits on the following instructions: `IF_ACMPI*`, `IF_ICMPI*`, `IF*`, `IFNONNULL`, and `IFNULL`. Bytecode instructions corresponding to other types are obvious (e.g., `PUTFIELD` matches `field_access` and `field_write`).
- `object_created [oc] = class name` - the breakpoint hits if the specified class (with the same name) is created. Example: `cr bp state=en 3<=hc<=3 object_created=java.lang.Thread` - breakpoint hits when the 3rd thread is created.
- `object_released [or] = class name` - the breakpoint hits if an object of the specified class is freed by the garbage collector. Example: `cr bp state=dis object_released=java.lang.String` - counts freed strings.
- `exception_thrown [et] = class name` - the breakpoint hits if an exception with the given type is thrown. Example: `cr bp state=en et=java.lang.RuntimeException` - hits when `RuntimeException` is thrown.
- `thread_scheduled [ts] = in / out / both : thread number` - the breakpoint hits when the specified thread is scheduled. The `: ThreadNumber` argument can be omitted, in this case the breakpoint hits if any thread not running in previous transition is scheduled. Example: `cr bp state=log ts=both:0` - hits if the main thread is scheduled in or out.
- `garbage_collection [gc] = begin / end / both` - the breakpoint hits if the garbage collection is started, stopped or in both cases. Example: `cr bp state=log garbage_collection=both`.
- `choice_generator [cg] = data / scheduling [sched] / both` - the breakpoint hits if a choice of the specified type is used (a new value is generated or a new thread is scheduled). Example: `cr bp state=en 50<=hit_count choice_generator=data`.

Single stepping

Inspector supports single-step execution in a GDB-like manner. Currently it is possible to step over a source code line with or without skipping of nested method calls, and to step over an instruction - all that only in the forward direction. All commands have a long name and a shortcut.

The `step_over [so]` command tells JPF to execute all instructions on the current source code line, including all nested method calls.

The `step_in [si]` command tells JPF to execute all instructions on the current source code line, but to stop when a control steps in a nested method call.

The `step_instruction [sins]` command tells JPF to execute the next instruction and then stop.

Be aware of these two facts: (1) breakpoints are not disabled by single-step execution and (2) breakpoint conditions are tested before each instruction is executed. Consequently, a step command may not move the program's execution forward if a breakpoint is hit on the current instruction (to be executed).

Program state inspection

Inspector can print the current value of various components of a program state. Note that the Java program (upon which JPF is applied) must be compiled with debugging information for most of these commands to work. The `#` characters are needed to distinguish names of local variables and

keywords.

The `thread [ti]` command prints the basic information (name, status, priority) of each thread. When a thread index is provided as an argument (optional), information is printed only for a single thread with the given index.

The `thread_pc [thpc]` command prints the source code line corresponding to the current program counter (current instruction) for all threads. When a thread index is provided as an optional argument, the source code line is printed only for a single thread with the given index.

The `print #thread[i]` command prints basic information about the i-th thread and its call stack (method names and source code lines). Information for the current thread is displayed, if no thread number is given.

The `print #thread[i].#stackFrame[j]` command prints all local variables (names, types, and values) in the j-th stack frame of the i-th thread.

The `print #thread[i].#stackFrame[j].#localVariable[k]` command prints values of all instance fields of the object pointed to by the k-th local variable on j-th stack frame of i-th thread. It is possible to use the term `#this` instead of `#localVariable[0]`.

The `print #heap[i]` command prints values of all fields of a heap object with the index i. It is also possible to print all heap objects of a given type by using the `print #heap[class name]` command. The given class name can contain * to match any arbitrary number (including zero) of characters. This is useful to avoid writing package name. Example: `print #heap[*]` prints all objects on the heap and `print #heap[*.String]` prints all strings stored on the heap.

It is also possible to navigate through the heap via a chain of field names, and print values of all fields of corresponding objects. For example, commands like `print #thread[2].#stackFrame[1].#localVariable[1].left` and `print #heap[427].schedParams.allocTh` are valid.

A few shortcuts are supported too. The `print` command with no arguments prints all local variables in the top stack frame of the current thread - it is a shortcut for `print #thread[i].#stackFrame[0]`, where i is the index of the current thread. The `print this` command prints values of all fields of the receiver object ("this") for the currently executed method.

Understanding output

We show the commands' output on the `DinningPhil` example from `jpf-core`.

Threads

The `print #thread` command

```
?:>print #thread[0]
cmd:> print #thread[0]
0 : main state=RUNNING priority=5
  DiningPhil.main(String[]) - DiningPhil.java:54 -      for (int i = 0; i < N; i++) {
```

Where:

0 - thread index

main - thread name

state and priority are clear

In the following indented lines(line) the function call stack is printed.

`DiningPhil.main(String[])` - specifies class, method name and parameter types

`DiningPhil.java:54` - specifies file name and **line** with actual position or call (in deeper stack frames)

`for (int i = 0; i < N; i++) {` - is line from the source file which is executed

Objects and Methods

The `print #thread[0].#stackFrame[0]` (or the 'print' command without any parameters) or 'print #heap[315]' etc.

```
?:>print
cmd:> print
DiningPhil.main(String[]) - DiningPhil.java:54 -      for (int i = 0; i < N; i++) {
    0 : args (java.lang.String[]) =[Ljava.lang.String;@137
    1 : forks (DiningPhil$Fork[]) =[LDiningPhil$Fork;@293
    2 : i (int) =5
```

The structure of the first line ("DiningPhil.main(String[]) - DiningPhil.java:54 - for (int i = 0; i < N; i++) {") is same as in the previous example.

Then follows an indented list of local variables and parameters (in the method case) or fields (in the case of objects).

Take the first variable ("0 : args (java.lang.String[]) =[Ljava.lang.String;@137")

0 - index of the variable (Index can be used in subsequent print commands like 'print #localVariable[0]')

args - the variable name

java.lang.String[] - is a type of the variable

[Ljava.lang.String;@137 - a value of the variable.

- Primitive type - the value of the variable (see the 3rd entry with int type).
- Reference type (object or array) - the value compounds of actual type (in internal format - "[Ljava.lang.String;") followed by @ and index in the heap where the reference points to. You can use the 'print #heap[137]' command to see internal structure to follow the reference.