JavaCI User's Guide

Introducing JavaCI

JavaCI is a fusion of a Java Virtual Machine and the MPE/iX Command Interpreter. The primary purpose of JavaCI is to provide faster startup time for Java processes. This has been done creating a single Java class which can load and execute other Java classes, but which can also recognize and execute MPE/iX Command Interpreter commands. This class essentially loops forever, writing out prompts and reading back in the requested command. If a command to invoke Java is seen, the running VM can in most cases load and execute the requested Java class without needing to start a new Java VM. Non-Java commands will be executed by the MPE/iX Command Interpreter via the HPCICOMMAND intrinsic.

Ideally, the environment provided would by 100% compatible with the MPE/iX Command Interpreter, and provide 100% compatibility with Java as invoked from a non-JavaCI environment. In reality, there are some functional differences from both sides.

JavaCI and the Command Interpreter

Several commands provided by the MPE/iX Command Interpreter are not executable programmatically via the HPCICOMMAND intrinsic. Here is a list of those commands, and what JavaCI will do in each case:

- `ABORT` Not supported; See discussion on BREAK below
- `BYE` Exits the JavaCI
- `CHGROUP` Does a `CHDIR` instead. This will put the user in the desired directory, but will not alter the group logged on to for purposes of accounting, or change the group reported by a `:SHOWME` command.
- `DATA` Not supported by JavaCI.
- `DO` See Discussion on `REDO` below
- `EOD` Not supported by JavaCI
- `EOJ` Not supported by JavaCI
- `EXIT` Exits the JavaCI
- `HELLO` Not supported by the JavaCI
- `JOB` Not supported by the JavaCI
- `LISTREDO` See discussion on `REDO` below
- `SETCATALOG` Not supported by the JavaCI
- `REDO` See discussion on `REDO` below
- `RESUME` Not supported; see discussion on BREAK below

Break, Abort, and Resume in the JavaCI

JavaCI does not do anything to intercept or handle break. In cases where JavaCI is started from the MPE CI, this means that BREAK will be handled by the underlying Command Interpreter process. If JavaCI is started with the `:NEWCI` command, BREAK will be disabled.

It would certainly be desirable for a CI-workalike such as JavaCI to provide the same functionality as BREAK within the MPE CI, but the interfaces that must be called to implement this functionality cannot be called from non-privileged-mode programs such as Java.

Redo, Do, and Listredo in the JavaCI

The MPE CI redo stack is not programmatically accessible, but redo functionality is such an important part of the CI that the functionality has been duplicated within the JavaCI. The actual editing of command lines is done via the CI's `EDIT()` evaluator function. Maintenance of the redo stack and selecting entries from the stack for `DO` or `REDO` by position or pattern match are all done in Java code.

The `HPREDOSIZE` CI variable, which sets and reports the size of the redo stack, is not programmatically readable. Therefore, when JavaCI starts up it selects an arbitrary size of 50 entries for the redo stack. If the `HPREDOSIZE` variable is changed via `:SETVAR` within the JavaCI, both the JavaCI's redo stack size and the redo stack size of the 'real' MPE CI will be changed to the requested value.

Command files and UDCs in the JavaCI

When you enter an MPE command at the JavaCI prompt, MPE's normal rules apply for invoking UDCs, command files, and built-in commands. Note that any UDCs or Command Files will be executed solely by the MPE Command Interpreter, not by JavaCI. Therefore, the blindingly fast execution of Java commands experienced from the JavaCI prompt will not be seen for Java commands that are part of command files or UDCs.

The JavaCI prompt

The Java prompt is based on the `HPPROMPT` environment variable, but is modified by prepending the string (JAVACI) to the prompt. This is done so that users cannot 'forget' they are in the JavaCI environment, especially in cases where JavaCI behavior might be different from the MPE CI.

Command Interpreter Extensions

There is opportunity with JavaCI to add new MPE commands or otherwise enhance or extend the capabilities of the MPE Command Interpreter. We have for the most part resisted this temptation, as our intent is to provide specific Java-related capabilities, not to build an incompatible CI replacement. At this point, there are only two points where we deliberately added commands or functionality to the JavaCI environment that is not available in the MPE CI.

The first of these is an implied run capability for Java. If a command passes all the way through the normal CI processing (checking for commands, UDCs, and command files) without matching anything, then we test to see if it is a Java class. The case-sensitive class name is checked for in each directory and jar file specified by the `CLASSPATH` CI variable. If the command name (plus '.class' extension) is found in any of the directories or jars, then the class is loaded and executed. If no class is found, then the Unknown Command Name (CIERR 975) error is reported.

The second extension is a new MPE command, `:RESETCLASSLOADER`. This command will be introduced and explained under the 'classloading' heading in the next section.

JavaCI and Java

As mentioned at the start, the objective of JavaCI is to provide the same capabilities and behavior seen when Java is invoked from a CI or shell prompt, only with greater performance. The greater performance comes in two areas. The first is load time, and is quite dramatic. Since the JavaVM and system classes are

already loaded, when a command such as '`java HelloWorld`' is encountered, it is only necessary to load the HelloWorld class and invoke its main method. On low-end systems where the JavaVM startup for such a command might be in excess of 10 seconds, executing the same command from the JavaCI will complete in under one second.

If the same command is executed yet again, performance will be even faster. This is because the target class (HelloWorld in the above example) has already been loaded, and may have been compiled by the just-in-time or the HotSpot compiler. In the case of HotSpot, you may observe increasing performance across a number of runs of the same program, until it reaches a steady-state where the majority of the code within the program has been compiled. (This is more likely to be observed for programs that have short run times; for long duration programs, most compilable code will have been compiled after the program runs once).

Note that in order for classes to be run in the existing VM, it is important that no VM options be passed on the command line. Options such as –verbose, -Xms16m (to change heap size), etc. specify attributes that cannot be changed once a virtual machine is running. Since JavaCI is inside an already-running JVM, it cannot change these attributes of the JVM it is running inside of. If any VM options are passed on a Java command line, a new instance of the JVM will be created to execute that command. Subsequent commands that do not specify command-line options will continue to be executed by the JavaCI's VM.

Classloading

JavaCI includes a customized classloader (known as the JCIClassLoader). There were two reasons why a customized class loader was required.

The first reason has to do with the `CLASSPATH` variable. The normal Java classloader reads the `CLASSPATH` variable at the time the JVM is started, and then will not check it again. Changes to the `CLASSPATH` variable or to the corresponding java.class.path property will not be recognized. Because of the way JavaCI will typically be used, it is important that the user be able to do a `:SETVAR` on the `CLASSPATH` variable, and have this new `CLASSPATH` used in any subsequent invocations of the classloader. The JavaCI program will reset the JCIClassLoader's classpath each time Java is invoked.

The second reason is to support class re-loading. In a typical Java environment, if a class is loaded then it will not be re-loaded even if the underlying file (.class or .jar) changes. This is unacceptable in the JavaCI environment, since the user could be editing a .java file, compiling it, testing it, and then repeating the cycle. If the class was not reloaded, none of the changes made to the class would take effect until the user completely exited JavaCI.

The rules for Java class loaders make this difficult; in particular, you cannot selectively reload a single class. The only way to get any classes at all to reload is to completely replace the classloader, thus reloading all classes that were loaded with that classloader. Although this is more expensive that reloading a single class, it is still far more efficient than restarting the entire Java VM.

We've tried to optimize when we force a 'reload of all classes'. We currently do so in two situations. The first is when the java compiler (javac) is invoked on a class that we have previously loaded through the JCIClassLoader. This is a pretty clear-cut indication that the class contents have changed and we should reload. The second case is whenever a user does a `:SETVAR` on the `CLASSPATH` variable. In this case, it's less clear whether a reload will actually be required, but it is entirely possible that the new `CLASSPATH` setting may cause some classes to be found at different locations that we previously loaded them from.

There are all sorts of situations where we will not catch changes to class files or to the classpath that would necessitate a reload. Any commands executed from UDCs or Command Files are not seen by JavaCI, and these may affect the environment. If new classes are installed on the system, via FTP or other means, we will not recognize that these might be in positions where they should be loaded in place of our cached classes. To handle all these unforeseeable cases, we have added a `:RESETCLASSLOADER` command that

will cause a new instance of the JCIClassLoader to be created, and all classes previously loaded will be reloaded if they are needed again.

Note that all of this behavior only works for classes loaded by our own JCIClassLoader. We cannot reload classes that are loaded by the default system class loader. For this reason, the recommended way to start the JavaCI program is with a classpath of null specified on the command line. This way, the system class loader will not have access to anything on the `CLASSPATH`. The system classes and JavaCI itself will be loaded by the system class loader, everything else will be loaded by the JCIClassLoader.

Security

As noted above, the JavaCI uses its own classloader. It also uses its own SecurityManager, in order to prevent programs that call System.exit() from terminating the entire JavaCI program. Installing ClassLoaders and SecurityManagers in the Java VM is a privileged operation, and requires special permission via the .java.policy file. In order for JavaCI to work on your system, you must install a policy file which grants JavaCI these capabilities.

Security files can be installed either systemwide or on a per-user basis. The system-wide security policy file is /usr/local/java/<version>/jre/lib/security/java.policy. The user-specific security policy is $HOME/.java.policy.

The following file (at either user or system-wide level) is currently being used to test JavaCI:

```
grant {
      permission java.security.AllPermission;
};
```

We are working on finding a more granular approach to give only the specific permissions required by JavaCI. If you install this system-wide, it is recommended that you save the default java.policy file (perhaps renaming it to java.policy.default) so that it can be restored later.

JavaCI and the POSIX shell

A logical thing to ask at this point is whether we can provide the same JavaCI functionality for the POSIX shell. The answer is probably yes. While we believe it can be done, the approach must be different, because the POSIX system() command is not nearly as elegant and useful as HPCICOMMAND(). With HPCICOMMAND, we can set environment variables and have them persist. But the POSIX system() call, which allows us to execute shell functions programmatically, is a one-shot deal; none of the context from an invocation of the command is preserved for later invocations. So environment variables, .profile scripts, etc. are completely useless.

The most likely design for implementing a JavaSh (or jsh) would be to have a shell process running at all times, with its stdin/stdout/stderr redirected to pipes, sockets, or message files. The jsh would then provide the user interface and pass all non-java commands off to the shell process for execution.

Since we have limited bandwidth within the lab to work on projects of this nature, we'd like to understand how important jsh functionality is, in relation to enhancements to JavaCI or other java-related functionality users might desire.

Questions for Alpha Testers

- Are any of the unimplemented commands important to handle in JavaCI?
- Are there additional extensions to JavaCI that are desirable?
- How important is a jsh (Java Shell) program similar to JavaCI?
- How much does JavaCI help java startup performance in your environment?
- Does JavaCI perceptibly improve Java runtime performance?
- Is the JavaCI documentation (this document) adequate?