CICS JVM Application Isolation Utility

The IBM® Persistent Reusable Java™ Virtual Machine Version 1.4.2 is the last version of Java on the z/OS platform to support the resettable mode of operation. In the IBM® Developer Kit and Runtime Environment, Java™ 2 Technology Edition, Version 5, there is no resettable JVM. Because of this, CICS Transaction Server for z/OS Version 3 Release 1 is the last release of CICS to support the resettable JVM.

In a resettable JVM, the state of the JVM is reset after each use, so that no application transaction (i.e. non-trusted middleware code) can affect the operation of subsequent transactions. The JVM reset cleans up the JVM's storage heaps, reinitializes shareable application classes, and discards and reloads nonshareable application classes, meaning that no non-trusted static middleware objects can persist in the JVM from one use of the JVM to the next. Also, if an application causes a change to a part of the JVM's own state which cannot be reset, then the JVM is terminated. Examples of such state changes include setting a system property, or loading a native library.

The alternative to the resettable JVM in CICS is the continuous JVM. The continuous JVM does not reset the JVM's state between uses. This enables the persistence of static objects across tasks, which can be a powerful tool when used deliberately. For example, an application developer can use caching techniques to avoid reinitializing objects on each use. It can also, however, be a source of unexpected and erroneous behaviour unless it is handled carefully.

Because there is no reset operation in the continuous JVM, applications which were designed to execute in a resettable JVM might exhibit changed behaviour when they execute in a continuous JVM. You might need to make changes to an application in order to preserve its original behavior while running in a continuous JVM.

Example 1: altering `static` variables

The most common type of state change that an application can make is to alter the value of a `static` variable. `static` variables are shared by all instances of a class, unlike non-static variables which are allocated separately for each instance.

In a resettable JVM, when a class is first loaded, the JVM takes a copy of the initial value of each `static` variable and uses it to restore the variable to its original state at the end of each transaction. Consider the following trivial case:

```
public class HelloWorld
{
    public static int count = 0;

    public static void main(String args[])
    {
        count++;
        System.out.println("Hello World, count is " + count);
    }
}
```

In a resettable JVM, the `static` variable `count` is reset to zero by the JVM after each invocation of the HelloWorld `main()` method. The message therefore shows that count is 1 each time HelloWorld is invoked.

In a continuous JVM, however, `count` is not reset to its original value before the next invocation of the `main()` method, and the old, shared, value persists. The message therefore shows the count increasing by 1 on each invocation in subsequent transactions.

To preserve the original behaviour while running in a continuous JVM, the HelloWorld class could be changed to make `count` an instance variable and initialise it on each invocation in a constructor:

```
public class HelloWorld
{
    public int count = 0;

    public static void main(String args[])
    {
        HelloWorld hw = new HelloWorld();
        hw.count++;
        System.out.println("Hello World, count is " + hw.count);
    }

    HelloWorld()
    {
        count = 0;
    }
}
```

Example 2: altering the contents of `static` objects

A more subtle type of issue can arise when the `static` variable is an object reference whose internal state may change, as in this example:

```
import java.util.Hashtable;
import java.util.Enumeration;

class StaticHash
{
    private static final Hashtable myHashtable = new Hashtable();

    public static void main(String[] args)
    {
        int count = myHashtable.size();
        myHashtable.put("key" + count, "value" + count);

        Enumeration keys = myHashtable.keys();
        while (keys.hasMoreElements())
        {
            Object key = keys.nextElement();
            System.out.println("Found this key in the Hashtable: " + key);
        }
    }
}
```

In a resettable JVM, a new instance of `myHashtable` is created every time the JVM is reset, and it will only ever contain a single key, “key0”. In a continuous JVM, however, only one instance of `myHashtable` is created, and each time the class is run, a new key is added to it. The issue could be solved in a similar manner to the first example, by making `myHashtable` an instance variable and creating the new `Hashtable` in a constructor. Alternatively, `myHashtable` could be left as a `static` reference and be

reset each time by adding a constructor containing an invocation of `myHashtable.clear()`.

Auditing applications for the use of static variables

The CICS JVM Application Isolation Utility is provided to help system administrators and application programmers discover `static` variables in Java applications running in their CICS Transaction Server for z/OS regions. The application developers should then review the findings of the Utility to determine whether the application might exhibit unintended behaviour when executed in a continuous JVM. The Utility can also be used when migrating Java workloads from resettable to continuous JVMs.

The CICS JVM Application Isolation Utility is a code analyser tool which inspects Java bytecodes in Java Archive (jar) files and class files. It does not alter any Java bytecodes. It is provided as a means to help identify potential issues before they arise in a continuous JVM under CICS Transaction Server.

The Application Isolation Utility is delivered as a jar file. To install it, copy the file `dfhjaiu.jar` to a convenient directory .

To run the Utility, log in to a z/OS Unix System Services shell, and enter:

```
java –cp dfhjaiu.jar CicsIsoUtil [-verbose] filename [filename…filename]
```

The Utility can inspect Java bytecodes in class files and jar files. Wildcard (glob) characters can be used in the file name.

For example, to inspect the HelloWorld class file used in Example 1 above, enter:

```
java –cp dfhjaiu.jar CicsIsoUtil HelloWorld.class
```

The report produced by the Utility is written to System.out, which may be redirectedto another destination as required. For the HelloWorld class file used in Example 1 above, the report looks like this:

```
CicsIsoUtil: CICS JVM Application Isolation Utility

Copyright (C) IBM Corp. 2006

Reading Class File: HelloWorld.class

    Method: public static void main(java.lang.String[])
      Static fields written in this method:
        public static int count

    Method: <clinit> (Class Initialization)
      Static fields written in this method:
        public static int count

  Number of methods inspected       : 3
  Total static writes for this class: 2

Number of Jar Files inspected    : 0
Number of Class Files inspected  : 1
```

The report shows that the **static** field **count** is written to during Class Initialization, and in the **main()** method. This indicates that **count** might behave differently when the class is used in a continuous JVM, than in a resettable JVM. The application programmer should examine the source code to decide whether **count** really will behave differently.

For the StaticHash class file used in Example 2 above, the CICS JVM Application Isolation Utility report is as follows:

```
CicsIsoUtil: CICS JVM Application Isolation Utility

Copyright (C) IBM Corp. 2006

Reading Class File: StaticHash.class

    Method: <clinit> (Class Initialization)
      Static fields written in this method:
        private static final java.util.Hashtable myHashtable

  Number of methods inspected       : 3
  Total static writes for this class: 1

Number of Jar Files inspected    : 0
Number of Class Files inspected  : 1
```

Note that the **static** variable **myhashtable** is only written to during Class Initialization, yet the internal state of the **Hashtable** changes on each invocation. This is a more difficult issue to assess. The output of the Utility identifies that a static object exists. The application developer must then check the source code of the application to ensure that the state of the **static** object (and the entire graph of other objects that may be referenced from the original **static** object) is not changed in a way that will unintentionally affect subsequent invocations of the class in a continuous JVM.

Normally, the Utility does not print details of methods which do not write to **static** variables, or details of **static final String** variables. With the –verbose option specified, the Utility does print these extra details, and also lists all static method invocations made. This additional information can identify other potential issues with your applications. For example:

```
      Static methods invoked by this method:
        boolean isResettableJVM()
           (defined in class: com.ibm.jvm.ExtendedSystem)
```

All methods in the **com.ibm.jvm.ExtendedSystem** class are related to the resettable JVM. They have all been deprecated, and should be removed from any application code.