Using Java™ Technology With int**e**nt®

Version 1.25

1. OVERVIEW OF INTENT™, JAVA™ TECHNOLOGY EDITION 3

2. JAVA TECHNOLOGY AND INTENT 3

2.1 THE JAVA PLATFORM 3
2.2 THE JAVA LANGUAGE 3
2.3 THE JAVA VIRTUAL MACHINE 4
2.4 THE JAVA LIBRARIES 4
2.5 MULTITHREADING 4

3. USING INTENT, JAVA TECHNOLOGY EDITION 5

3.1 JAVA™ BYTE CODE TRANSLATION AND EXECUTION 5
3.2 MIXING DYNAMIC AND STATIC EXECUTION 5
3.3 DYNAMIC RUN-TIME EXECUTION OF JAVA BYTE CODE 5

4. RUNNING THE APPLICATION 6

4.1 STATIC BUILD-TIME TRANSLATION OF JAVA BYTE CODE. 6
4.2 GARBAGE COLLECTION 6

1. Overview of int**e**nt®, Java^{TM} Technology Edition

int**e**nt® enables compelling interactive content even on low end devices. It incorporates the intent multimedia toolkit and the engines necessary to run the content, such as intent, Java^{TM} Technology Edition.

int**e**nt, JTE is a implementation in Virtual Processor Code of the Personal Java API class libraries, combined with a JVM and runtime environment. It also includes a translator to convert Java Byte Codes into VP byte codes. As a Sun Microsystem's authorized VM provider, Tao makes use of Sun's test suite to ensure the compliance of int**e**nt JTE to Sun's own specifications.

Tao's Sun authorised implementation allows the class libraries to be implemented in a memory efficient manner, thereby allowing int**e**nt, Java Technology Edition to function in embedded environments where size and performance issues would normally prove this to be prohibitive. The advantage of translation is simply that a translated system is near to one hundred percent efficiency compared with only a few percent for a interpreted system, as commonly used by Java Virtual Machines. All translation can be carried out at system build time, or dynamically at run-time. Run-time Java technology facilities such as garbage collection and exception handling are also included as part of int**e**nt, JTE.

It should be noted that most execution environments for the Java language are written for a specific machine type. For example the implementation of the maths libraries will be optimised for a specific host machine and the back end of any JIT (just in time compiler) will be specific to a processor or chip set. The vast majority of int**e**nt has been written in the Virtual Processor (VP) code, machine neutral format used extensively by the Elate operating system and the int**e**nt media platform. This means that the entirety of int**e**nt, JTE can be transparently moved from processor to processor, and from system to system. The only porting required is at the level of other system components, such as the Platform Isolation Interface, as documented in the '*System Programming Guide*.'

This document details basic usage information for int**e**nt, JTE. It was written as a companion to the document "*Introduction to Tao's Java^{TM} Solution,"* which provides more basic details concerning both Java technology and int**e**nt JTE.

2. Java Technology and int**e**nt

The translation of the Java Byte Code is entirely transparent as far as the programmer need be concerned, in that the functionality of the standard library functions (API) conforms with the PersonalJava^{TM} API. However the re-implementation of the Java Byte Code as VP code allows for significantly greater speed and minimal size. This section of the document describes these alterations, but for more detailed information concerning the operation of Tao's technology, the reader is referred to the appropriate manuals. The most relevant are as follows:

- Elate Tool Programming Guide (VP)
- Elate Object Based Programming Guide
- VP Reference Manual
- The Reference Manual for the int**e**nt™ kernel
- The Reference Manual for the Sysbuild utility

2.1 The Java Platform

The Java Platform may be divided into three basic components - the language, the virtual machine and the libraries.

2.2 The Java Language

int**e**nt, JTE has the capacity to work with any compiler that produces conformant class files.

2.3 The Java Virtual Machine

The Java Virtual Machine is a description of a software machine that must conform to the standard specification produced by Sun Microsystems. int**e**nt, JTE implements the Standard Java Virtual Machine.

However, the techniques and details of the implementation vary greatly from a standard JVM implemented, for example, using Just In Time (JIT) compilation. int**e**nt makes no use of interpretation. The internalisation of classes comprises the translation of Java Byte Code into native code, which is subsequently executed. This native code contains all of the required information, so that the original Java Byte Code may then be freed. In addition, the compilation of all code ensures greater consistency of performance.

Note that int**e**nt, JTE permits multiple virtual machines to be used on the same platform.

2.4 The Java Libraries

The Java libraries provide the Java environment with all of the services available on the particular host platform. All core libraries are implemented in Tao's Virtual Processor (VP) code, with most optional libraries being implemented in the Java language itself.

2.5 Multithreading

Java technology has threading built into the core of the language, in fact, threading is an underlying assumption of the Java object model. That is, the Java class Object has threading methods defined in its interface. This means that any Java operating environment must implement a thread or lightweight process model of some kind.

Because it provides built-in support for lightweight processes at its core, int**e**nt, JTE can directly map Java threads onto the lightweight process model. Thread scheduling can be tuned on a per platform basis to achieve high reliability and performance.

Within the Tao implementation, each Java thread for a Java Virtual Machine has the same memory objects, environment list, signal handlers and statics. As each Java thread is a separate process they can then be assigned priorities individually. Tao's Java implementation builds on its realtime kernel, with each Java thread being a kernel thread. The int**e**nt process that starts the VM maps to NORM_PRIORITY in Java. MIN_PRIORITY is NORM_PRIORITY-4, MAX_PRIORITY is NORM_PRIORITY+5, thus giving a range of 10 intent process priorities for Java threads in this Virtual Machine. Other VM's may be started with the same or a different intent priority, thus possibly having a different range of int**e**nt priorities.

3. Using int**e**nt, Java Technology Edition

3.1 Java™ Byte Code Translation and Execution

Java technology is inherently dynamic (for example, as with downloaded applets and applications, which may be directly invoked). To make these technologies viable the underlying operating environment must also be dynamic in nature.

int**e**nt, JTE has support for dynamic class and method loading at its core. It uses dynamic tool loading, among other features, to implement the Java run time environment, so there is no conceptual gap between int**e**nt, JTE's functions and facilities, and the dynamic environment which Java technology demands.

Legacy operating systems tend to be forced to implement many features that are not native or inherent in the underlying environment, in order to produce a realistic Java Engine. This has impact upon not only memory footprint but also performance, as the semantics of the implementation vary through legacy concepts and features.

3.2 Mixing Dynamic and Static Execution

int**e**nt makes no assumptions about the way in which individual platforms will take advantage of its features. It is possible to "brew" specific Java execution environments for each platform on which int**e**nt, JTE is implemented. This often amounts to mixing dynamic and static components of an environment to match specific needs. The following section describes how Java Byte Code can be dynamically executed in int**e**nt, JTE and how it handles the ability to "blend" in Java classes at build time if required.

3.3 Dynamic Run-Time Execution of Java Byte Code

int**e**nt JTE supports the PersonalJava™ 1.1.3 and 1.2 specifications. Java™ applets and applications may be invoked through the scripts discussed in the document '*Getting Started with int**e**nt'* on either Windows® or Linux®. Otherwise, the following command is used to load and if necessary translate a Java class, so for example to run this demonstration application type:

```
intent_java_stdio demo.example.j.Hello
```

It is also possible to format this command as follows:

```
intent_java_stdio demo/example/j/Hello
```

The java command starts a Java Virtual Machine (JVM) and proceeds to run a Java application within it. Please note that the name given to the Java command needs to be the absolute name of the class file. Execution starts in the main method of <classname>. Zip and jar files may be specified on the command line. The classpath option simply specifies the classpath to be used when loading classes from the local filesystem. Many directories or jar files can be specified, each separated by a ';'

The int**e**nt Virtual Machine has a notion of two different types of classes, system classes and application classes (which comprise of classes loaded through the classpath or by a classloader). System classes are those classes comprising the Java libraries, and are loaded using a built in classpath of '/' and will stay resident in memory (i.e. translated and bound) during the lifetime of a VM and after the VM exits, until a tool flush event occurs. Conversely, as far as application classes are concerned, the classpath is valid for the lifetime of the VM, while a classloader is only valid during its own lifetime.

When looking for a class the VM first searches the system classpath (i.e. '/'), and then the application classpath. The classpath used for application classes is specified by using the classpath option on the command line; any mention of '/' on the application classpath will be silently ignored. . There is no mechanism for modifying the classpath used for system classes. For more information about other options to this command, please see the file *app/stdio/java.html.*

4. Running The Application

The method described below is used as an entry point to the class specified in the java command:

```
public static void main (String [ ] args)
```

The name given to the java commands needs to be the absolute name of the file. This pre-translates Java classes.

4.1 Static Build-Time Translation of Java Byte Code.

Some uses of Java technology will include embedded applications wherein all of the Java classes required will be known when the application or device is manufactured. In this case it is possible to internalise only the required classes for the particular device's application by using int**e**nt's build time tools, by using Tao's sysgen utility. For more information upon this please see the '*Sysgen Reference Manual'.*

4.2 Garbage Collection

The Java garbage collector uses a mark-sweep algorithm to detect allocations which are no longer being referenced and then frees them (i.e. it scans dynamic memory areas for objects and marks those that are referenced). The garbage collector can be configured to run whenever a set amount of memory has been allocated following the last garbage collection. The Tao garbage collector's parameters can be dynamically altered to make best use of system resources. For example, it can be aborted when keys are pressed, to ensure that Java applications (and applets) respond quickly to user actions.

The garbage collector will allocate memory for both object and buffers of data to be used internally. Each allocation can be flagged to be searched for pointers by the garbage collector during the mark phase; i.e. byte and char buffers will never contain pointers to other memory allocations and therefore do not need to be searched.

The garbage collector mark phase starts with the contents of the registers and stacks of each Java thread, and searching for pointers to allocated memory. Each allocation found is then recursively searched for pointers to other allocations until a maximal set is found. Any allocations not found by this phase are marked and freed by the sweep phase.

An allocation is considered 'live' if there is a pointer to it held in another live allocation, the stack of a Java thread or a register of a Java thread. The pointer can be to anywhere in the interior of the allocation or to the immediately preceding word.