Introduction to Tao's Java^{TM} Solution

VERSION 1.20

1. INTRODUCTION 3
1.1 Hello World 3
2. THE JAVA™ PLATFORM 4
2.1 Java Byte Code 4
2.2 The Java Virtual Machine 4
2.2.1 The Registers 4
2.2.2 The Method Area 4
2.2.3 The Stack 4
2.2.4 The Garbage-Collected Heap 4
2.3 The Libraries 5
3. OBJECT-ORIENTED PROGRAMMING 6
3.1 Classes 6
3.2 Subclasses 6
3.2.1 Single Inheritance 6
3.3 Class Variables and Class Methods 7
3.4 Access Control 7
3.4.1 Packages 7
4. DYNAMIC ASPECTS OF JAVA TECHNOLOGY 8
4.1 The JIT (Just In Time Compiler) 8
4.2 Solving the Constant Recompilation Problem 8
4.3 Garbage Collection 8
4.4 Java Technology for Embedded Systems 8
4.5 Real-Time Behaviour 9
4.6 Memory 9
5. THE INTENT PLATFORM 10
5.1 Small Footprint 10
5.2 Object-Based Programming 10
5.3 Dynamic Aspects of intent 11
5.4 Real-Time and Multithreaded 11

1. Introduction

The Java™ programming language is an object-oriented, architecture-neutral dynamic language. It was originally designed, when C++ proved unsuitable, for use in a project to develop advanced software for consumer electronics.

The syntax of the Java language is similar to C/C++, and thus enables programmers familiar with C and C++ to learn the language with minimal training. Many of the less well understood features of C++ have been omitted, primarily those concerned with operator overloading (although the Java language does support method overloading), multiple inheritance and extensive automatic coercions. Another source of complexity in C and C++ applications, storage management, is dealt with in Java technology by automatic garbage collection; the freeing of memory not being referenced.

Developers have found that the language's simplicity, combined with its powerful object-oriented features, make it a productive language for developing applications programs, and its architecture neutrality (as implemented by the Java Virtual Machine) has made it the ideal language for the World Wide Web and Internet.

1.1 Hello World

Below is a Java language version of the popular demonstration program 'Hello World'. This can be found in the directory *demo/example/j/.*

```
package demo.example.j

public class Hello
{
     public static void main (String args[])
         {
         System.out.println("Hello from Java");
         }
}
```

The resulting output will be the words 'Hello from Java' printed to the system's standard output device.

This program declares a class called *Hello* and a method called *main* in the package *demo.example.j*. As the Java language does not support stray variables or global functions, the skeleton of every application must be a class definition. When a Java application is run it is necessary to specify the class to be run. The interpreter then invokes the *main()* method defined within that class. This method controls the flow of the program, allocates whatever resources are needed, and runs any other methods necessary. The package definition allows classes within the same package to access one another's members.

Note that *System.out* is not 100% pure Java technology, as not all Java platforms have the concept of standard input/output streams. The alternative would be use of a GUI.

2. The Java™ Platform

The Java platform is what makes it possible to run Java applications on many different types of machines with minimal rewriting of code. The portability of the Java language is achieved by two mechanisms: the translation into Java Byte Code (JBC) and the implementation of the Java Virtual Machine (JVM), as well as some aspects of the language itself.

2.1 Java Byte Code

Java technology was designed to support applications on networks. Since networks are often composed of a variety of systems with a variety of CPU and operating system architectures, the Java language compiler generates an architecture-neutral object file format.

The compiler does this by converting Java language source (*.java*) files into bytecode instructions which are easy to interpret on any machine, and easily translated into native machine code on the fly. These bytecodes are then placed into class (*.class*) files. One class file is generated for each class in the source.

2.2 The Java Virtual Machine

The JVM is an abstract computer that runs compiled Java programs (i.e. JBC). It is 'virtual' because it is generally implemented in software on top of a real hardware platform and operating system. As all Java programs are compiled for the JVM, programs will only run on a platform if the JVM has been implemented for that platform.

The 'virtual hardware' of the JVM can be divided into four basic parts - the registers, the stack, the garbage-collected heap and the method area. These parts must exist in every JVM implementation.

2.2.1 The Registers

The Java Virtual Machine has a program counter and three registers that manage the stack. It has few registers because the bytecode instructions operate primarily on the stack: this stack-oriented design helps keep the JVM's instruction set and implementation small.

The program counter (or *pc* register) is used to keep track of the next instruction to be executed. The other three registers (*optop* register, *frame* register and *vars* register) point to various parts of the stack frame of the currently executing method.

2.2.2 The Method Area

The bytecodes for a program are contained in the method area. The program counter always points to some byte in the method area, and following the execution of an instruction is set to the address of the instruction which follows the previous one.

2.2.3 The Stack

On the Java platform, the stack is used to keep the state of each method invocation, to store the parameters for and results of bytecode instructions, to return values from methods etc. The stack frame of a currently executing method keeps track of the state of that method invocation using three sections: the local variables section, the execution environment section and the operand stack. These are pointed to by the *vars*, *frame* and *optop* registers respectively.

It should be noted that the *optop* register points to the top of the operand stack, and as this is the uppermost stack section the *optop* register therefore points to the top of the entire Java platform stack.

2.2.4 The Garbage-Collected Heap

While the stack is concerned with methods, the heap contains the objects on which the methods act. When memory for an object is allocated with the *new* operator, it comes from the heap. Although in the Java language memory may not be freed explicitly, the runtime environment automatically frees those objects which are no longer being referenced. This process is known as garbage collection and is explained more fully in Chapter 4.3.

2.3 The Libraries

Several libraries of utility classes and methods are available in the complete Java system. These libraries include the basic Java language classes, the Input/Output package, the Java language utilities package and the Abstract Window Toolkit. These core libraries are implemented by Tao using the compact and architecture-neutral VP code (see Chapter 5 on The int**e**nt Platform), and are smaller than the Java technology implementation, which uses its large class libraries with neither size nor speed in mind.

The Java Language package contains the collection of base types that are always imported into any given compilation unit. This includes the declarations of Object and Throwable, the base classes for objects and exceptions, as well as threads and 'wrapper' classes for the primitive data types. The Input/Output Package contains the rough equivalent of the Standard I/O Library found on most UNIX systems. Classes in the Utility Package include common storage classes and special use classes such as Date. Transferring Java applications easily from one window system to another is enabled by the Abstract Window Toolkit, which contains classes for basic interface components such as colours, fonts, windows and panels.

3. Object-Oriented Programming

An object is an encapsulated piece of code whose state can be manipulated by methods specific to the class to which it belongs. An object is an instance of the class to which it belongs. Figure 1 is a conceptual representation of an object.

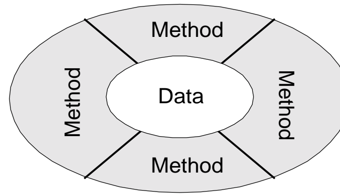

Figure 1: A Conceptual Representation of an Object

The variables belonging to the instance are encapsulated within the object. An object's methods are the only means by which other objects can access or alter its instance variables, although classes may declare their instance variables to be directly accessible by other objects. This is discussed later in Chapter 3.4 on Access Control.

3.1 Classes

A class is a software construct that defines the data (state) and methods (behaviour) of the specific objects constructed from that class. Objects are obtained by instantiating a previously defined class, and many objects may be instantiated from one class definition. Any other object may create an object by declaring a variable to refer to a point object and then allocating an instance of the object.

3.2 Subclasses

A subclass is a mechanism whereby new objects which are similar to already-defined objects can be defined in terms of those existing objects. A subclass inherits the methods of a single parent class, and also contains additional methods of its own. It may have subclasses of its own. Methods in the subclass override those in the parent class. The code for a subclass includes the name of its parent (or base) class. All classes in Java technology ultimately inherit from the class *Object*.

3.2.1 Single Inheritance

A subclass may inherit from only one parent class. In C++ a subclass could inherit from multiple parent classes. This could cause problems in cases where variables or methods in the parent classes had the same names. Whereas single inheritance simplifies things, it also causes the problem that where methods in another class might be useful to a subclass, they cannot be inherited. Java technology solves this by use of interfaces (see Figure 2). An interface declares methods, but does not include instance variables or implementation code. As the implementation code is contained within the class implementing the interface, the implementation may be altered without affecting the interface itself.

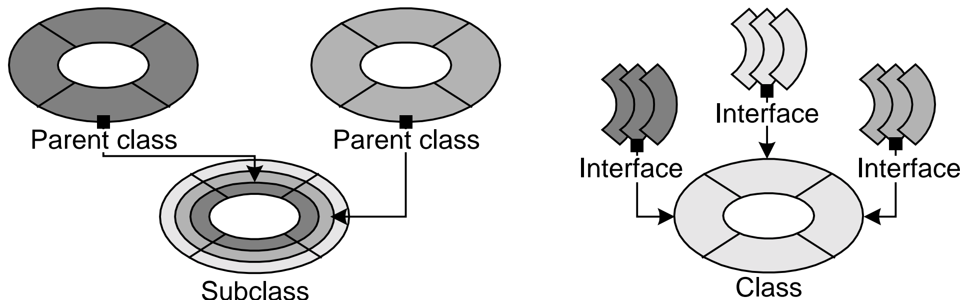

Figure 2: Multiple Inheritance vs Interfaces

Thus classes may implement as many interfaces as are needed, but may inherit from only one parent class (see Figure 3). One drawback to this is that a class must implement all methods of a given interface, a requirement which often leads to empty methods and code bloat.

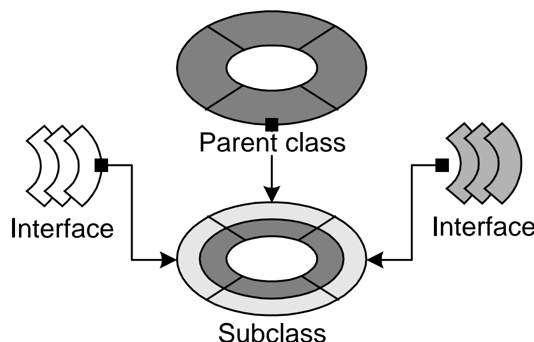

Figure 3: Inheriting from a Single Parent Class

Note that the inability to inherit the implementation of an inherited class means that less code is re-used than in a multiple inheritance model.

The same problem is solved in Tao's technology by the use of tools rather than classes and methods (see Chapter 5.2 on Object Based Programming) .

3.3 Class Variables and Class Methods

The Java language follows the conventions from other object-oriented languages in providing class methods and class variables. Unlike instance variables, of which there is one in each separate object created from the class, the class variable is local to the class itself and the single copy is shared by every object instantiated from the class.

Class methods are used when a particular behaviour is common to every member of a class, especially if knowledge is required which can only be obtained from other instances of the class.

3.4 Access Control

When declaring a new class it is possible to set the level of access permitted to its instance variables and methods.

The access level may be public, protected, package or private. These indicate whether subclasses of the class, classes in the same package of the class, and any other classes may access the member variable. The default is package, which allows members of the same package to access members.

3.4.1 Packages

Packages are collections of classes and interfaces which are related to each other in some useful way. They are created by storing the source files for the classes and interfaces of each package in a separate directory in the file system.

The primary benefit of packages is that they allow many class definitions to be organised into a single unit. The secondary benefit is that the useful class members are available to the classes within the package which might need them, but not to classes defined outside the package.

4. Dynamic Aspects of Java Technology

The Java language was designed to adapt to changing environments. Classes are linked in as required, and incoming code is verified before being interpreted. Since the Java language is an interpreted language, more compile-time information is available at runtime.

4.1 The JIT (Just In Time Compiler)

Despite the advantages of run-time interpretation, it has the drawback that it can be very slow. The JIT gets around this problem by compiling Java bytecodes to native machine code at runtime. Thus execution of a Java class invokes the JIT compiler (built into the interpreter) which compiles the bytecodes to native instructions. A JIT is usually optimised towards a certain platform.

Tao's Java platform does not require a JIT as such, as this functionality is inherent in Tao's standard translation mechanisms.

4.2 Solving the Constant Recompilation Problem

The 'fragile superclass' or 'constant recompilation' problem occurs in C++ as a side-effect of the way the object-oriented features are normally implemented. Any time a new member is added to a class, those classes which reference that class may break as a result of the change.

The Java language solves the constant recompilation problem in several stages. The major step is in Java language compilation - references are passed to the interpreter as symbolic references, rather than being converted into numeric values. It is the Java language interpreter which performs final name resolution and determines the storage layout of objects. Thus owing to the dynamic nature of Java technology, which interprets bytecodes on the fly, changes to the source do not require recompilation of all affected classes. A similar approach to translation solves this problem in int**e**nt

4.3 Garbage Collection

Garbage collection is central to the Java technology programming paradigm. It achieves high performance by taking advantage of the natural pauses in user behaviour. During these idle periods the Java run-time system automatically runs the garbage collector in a low-priority thread. Unused memory is gathered and compacted, dynamically freeing memory resources.

The Java technology memory management model is based on objects and references to objects. All references to allocated storage are through symbolic 'handles'. The Java memory manager keeps track of references to objects - garbage collection frees the memory used by objects which are no longer being referenced.

The dynamic and automatic nature of memory management in Java technology eliminates the need for explicit memory management, which in C and C++ has proved a major source of bugs, memory leaks and poor performance. Within int**e**nt JTE, garbage collection can be interrupted by system events.

4.4 Java Technology for Embedded Systems

It is critical that embedded or real-time systems are designed to cope with the timing constraints imposed by the world outside the computer - random, short-lived external signals must be responded to before the data contained within them is lost. Such systems normally have few resources and require an operating system with a small footprint. Sun's EmbeddedJava™ technology has shrunk the large footprint of Java technology, and the Java language has the advantage over C (currently used in most embedded programs) in that while C has permissive operations which can lead to undisciplined coding practice and unstable execution, the Java language requires explicit declarations and tight coding. Also, the universal standards provided by the Java language for multithreading and shared data protection make the program easy to transfer to alternate platforms. However, the behaviour of the int**e**nt JTE is entirely identical across platforms.

4.5 Real-Time Behaviour

The translation between Java language source and JBC gives Java technology platform independence, but at the cost of performance. Run-time interpretation can be slow, whereas often embedded systems require fast, known response times. Although the automatic garbage collector saves programmers effort and eliminates a major source of bugs, its tendency to impose long delays at unpredictable intervals interferes with the ability to comply with real-time constraints. The int**e**nt gc is interruptible, which has the effect of alleviating these pressures.

int**e**nt, Java Technology Edition translates rather than interprets the Java language, and does so at load time rather than during runtime, thus giving better runtime performance (see Chapter 5.3 on the Dynamic Aspects of int**e**nt).

4.6 Memory

Java technology was designed for the desktop applications market, which has very different economies from the embedded market. As such, current implementations of the Java technology are not usually space efficient. This is much less of a problem in int**e**nt, as the use of individual tools rather than classes is much more memory efficient.

5. The intent® Platform

intent® provides manufacturers with the opportunity to create devices with attractive user interfaces, complex applications and Internet access tools while minimising usage of system resources and getting their products to market faster than ever.

intent is a content platform combining both multimedia tools (gathered together in the intent media libraries) and the engines necessary to run the content, such as intent, Java Technology Edition.

The intent platform achieves portability by defining a virtual processor layer. The Virtual Processor defines a language, VP, and a platform isolation interface which allows a minimal set of platform dependent features to be isolated from the rest of intent. The basic translation mechanism is to load VP byte codes and translate these into native code. Normally this process takes place at tool load time, but can also be done at system build time.

intent runs Java applications using intent, Java Technology Edition, which is a full implementation of the Java Virtual Machine. It translates Java byte codes into VP byte codes and so can work with any development environment that produces Java classes in the standard byte code format. Unlike Java technology, intent, JTE was developed with the embedded market in mind, making small footprint and speed a priority.

5.1 Small Footprint

Rather than classes and objects, intent, JTE relies upon thousands of compact 'tools' which are loaded and bound upon demand. This means that Java™ technology can now be implemented even in the most memory constrained environments.

A set of core class libraries conforming to the PersonalJava technology API standard has been implemented in Tao's own portable VP code, which is highly memory efficient and produces code with high speed of execution. This enables a viable Java technology solution for the embedded environment.

5.2 Object-Based Programming

Object based programming differs from object oriented programming in that its fundamental unit is the individual tool. Tao tools are functions, small sections of executable code. Whereas object oriented programming requires that for a particular method to be used the entire class must be loaded, in object based programming each method is an individual tool, which may be called by any object. Tools can be used to build classes and objects, and thus program in an object oriented manner (see Figure 4).

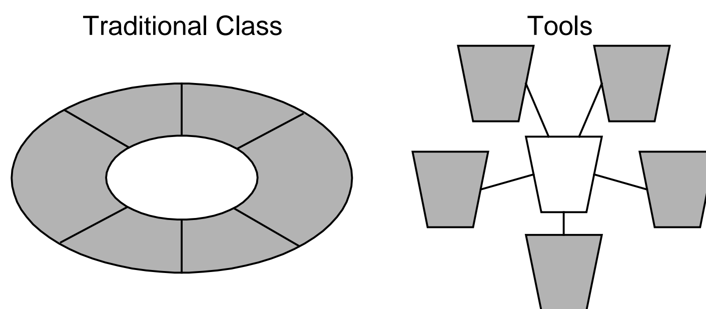

Figure 4: Building Classes with Tools

The intent implementation of Tao's Java engine implements Java language methods as individual tools. As in the Java technology, there is no fragile superclass problem. Binding (replacing the toolname with a pointer to the tool’s code) occurs after translation, similar to the way the Java language interpreter, rather than the compiler, performs final name resolution and determines the storage layout of objects.

Virtual Processor code, in which int**e**nt's tools are written, occupies a middle ground between the expressive but space-consuming objects used in object oriented programming, and low level assembler language.

5.3 Dynamic Aspects of int**e**nt

As in Java technology, int**e**nt tool loading is dynamic, in that tools are loaded when required then remain in memory until no longer referenced. Any tool not being referenced may be flushed from memory if the memory is needed.

The Jcode garbage collector scans dynamic memory areas for objects and marks those which are no longer referenced. These allocations are then freed. The garbage collector is configured to run whenever a set amount of memory has been allocated following the last garbage collection.

5.4 Real-Time and Multithreaded

int**e**nt has been designed to support real-time behaviour. Message passing between tools can be either synchronous or asynchronous depending upon task requirements, and various other objects of this kind, such as mutexes, signals and semaphores, are supported.

int**e**nt supports multi-tasking and will permit the simultaneous implementation of a range of scheduling policies.

In an environment where no underlying threads are used, it is usually necessary to build a thread model into the JVM. int**e**nt, JTE directly maps Java language threads onto the lightweight process model at its core, ensuring that thread scheduling can be tuned according to the particular platform on which the threads are to be executed.

© Tao Group Ltd or Tao Systems Ltd. 2000, 2001. All Rights Reserved.

Copyright in the software either belongs to Tao Group Ltd or Tao Systems Ltd. The software may not be used, sold, licensed, transferred, copied or reproduced in whole or in part or in any manner or form other than in accordance with the licence agreement provided with the software or otherwise without the prior written consent of either Tao Group Ltd or Tao Systems Ltd.

No part of this publication may be reproduced in any material form (including photocopying or storing it in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication) without the written permission of the copyright owner.

*Elate®, int**e**nt® and the Tao logo are registered trademarks of Tao Group Ltd.*
Digital Heaven™ is a trademark of Tao Group Ltd.
The rights of third party trademark owners are acknowledged.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries, and are used under license.