

White paper

Why Java?

Is Java better than other programming languages?



SNIJDER

Snijder Micro Systems
Visser 25, NL-5751 BL, Deurne
P.O.Box 300, NL-5750 BH, Deurne
The Netherlands

Phone: +31-(0)493-351020
Fax: +31-(0)493-351530
E-mail: info@snijder.com
<http://www.snijder.com>

Copyright © 2000 Snijder Micro Systems.
All information contained in this manual is proprietary to Snijder Micro Systems.

1. Introduction

This document provides an overview of the main advantages of Java over other traditional programming languages, like C/C++ or others. This is not an in-depth discussion of all features of the Java language – that would be far beyond the scope of this document, and besides, there are already many good books on that topic. Instead, this document is meant to be a brief summary of the most important features of the Java language, from a programmer's standpoint, and bearing in mind the specifics of the Intent Java platform where it makes a difference. Intent is based on the Elate RTOS (Real Time Operating System) from Tao Group Ltd.

2. Main features of the Java language

2.1. Platform independence

Possibly the most important feature of the Java language, and for sure the most popular one, is its platform independence or 'portability', which is referred to by Sun with the motto "Write Once, Run Anywhere".

Although other languages also claim to be portable (C and C++ spring to mind) it is important to realize that the term 'portability' has a very different meaning in Java. First, languages such as C or C++ are portable at the source level, that is, applications must be recompiled or rebuilt for each target platform. On the other hand, Java classes are *binary* portable. Once a Java class is compiled into a stream of bytecodes, it can be executed on any Java-compliant platform. One of the results of this feature is that technologies like automatic upgrades, remote execution, etc. are easier to implement.

Second, in languages such as C or C++, portability is very limited, even at the source level. For example, the sizes of primitive types (byte, int, float ...) are not defined in the ANSI/ISO standard, and thus they might be different on each implementation. This makes it more difficult to write portable applications. The same happens with endianness – applications cannot rely on the underlying platform being big endian or little endian, as this feature is implementation-dependent. None of these problems exists in Java.

2.2. Comprehensive standard libraries

Portability does not only depend on the language itself, but also on the completeness of the standard libraries for that language. A real world application will need to deal with issues like Internet connectivity, multithreading, timers, internationalization, keyboard handling, GUI, etc. If this functionality is not part of the standard libraries, portability is severely compromised as it will be necessary to rely on homebrew or 3rd party libraries to bridge that gap. This is the case in C/C++ where the standard libraries are rather limited. On the other hand, Java includes a very comprehensive set of libraries and APIs which are guaranteed to be available on any Java-compliant platform. Furthermore, there are many optional packages for more specific purposes which are implemented as pure, platform-independent Java code and thus will work on any Java-compliant platform.

Obviously, this rich availability of frameworks, libraries and packages not only helps with the portability issue, but also reduces development time and maintenance needs, since developers can instead concentrate on the features that are really specific to each particular application. Moreover, since the basic libraries are standardized and all Java code is binary portable, it is possible to reuse existing objects and components effortlessly. Availability of such components is guaranteed due to the growing popularity of the Java language.

A good example of all this is the construction of the GUI for a given application. Using the Java AWT (Abstract Windowing Toolkit) standard libraries as a basis, complex graphical widgets can be created, bought, developed anywhere, redistributed. With other languages it would be necessary to rely on proprietary, platform-specific GUI toolkits.

2.3. Dynamic loading

An important feature of the Java language is its ability to dynamically load and execute Java classes. This enables technologies such as remote software upgrades, hot deployment, seamless support for application extensions through plug-ins, etc.

For example, an embedded application might be easily extended with new functionality which could not be foreseen at system design time. Large packages, such as diagnostic or configuration tools, could be loaded on demand and unloaded afterwards, thus freeing all allocated system resources as soon as the tools are not in use anymore. Added-value tools and services may be sold later to extend the basic functionality of the applications, and this can be done online with no extra effort.

The whole applet paradigm also relies upon these technologies – users can download small applications and run them in their browsers without having to go through any recompilation or installation. The same holds true for Java servlets, which can be seen as Java-based plug-ins that dynamically extend the functionality of web servers.

2.4. Development environment

The platform independence of the Java language also simplifies the development process and reduces its cost. Other languages/platforms, like C/C++ on OS9 or VxWorks, require special (and often expensive) development environments. With Java, it is possible to use any PC or workstation with Windows, Linux or any other OS supporting Java as a development system, and once the code is compiled and ready it can just be transferred to the target system and it will run without modifications.

2.5. Very high-level nature

Java is a very high-level language, much more so than other languages like C or C++. It features a syntax based on that of C/C++, but cleaner, with built-in exception handling, automatic garbage collection (it is well known that memory management was one of the most common sources of bugs and problems in C and C++), namespaces, built-in synchronization mechanisms, reflection, RTTI, single inheritance and interfaces instead of multiple inheritance, and a long etc. All this means that the language itself is easier to use and less error-prone than other alternatives.

2.6. Security and safety for 3rd party code

If a system is to be opened for 3rd party Java applications, this means that there will be non-trusted code running on that system. This code could be malicious or maybe just poorly written, and this in turn might mean a potential danger for the stability of the system. This raises a few questions about security and resource usage.

Java provides a sandboxing security model that can be used to validate and restrict rights of non-trusted code. This way it is impossible to crash or mess up the system, as this security model is built in the Java language itself. Lower level languages such as C, C++, VP or any native assembler cannot ensure safe execution of non-trusted code, simply because they have not been designed this way (and this is something that can't be added later). On the other hand, Java was designed with security in mind, and the language itself provides a security model which allows executing non-trusted code without worrying about the stability or safety of the system. This is done through three levels of defense:

Static bytecode verification: all bytecode which is to be executed in a Java Virtual Machine is validated and checked at a number of different levels *before it is allowed to run*. This includes checking that the format of the bytecode fragment is correct, making sure that the code does not forge pointers, violate access restrictions, or access objects using incorrect type information. This way the JVM makes sure that it is not possible for a "hostile compiler" to create bytecodes that would crash the JVM.

Class loading: the class loading model is designed in such a way that it is impossible to replace core classes in the Java runtime environment by code coming from other parties.

Security manager: the security manager restricts the way in which applications can use visible interfaces from the Java runtime. This is done by performing run-time checks on "dangerous" methods. These checks can be specified by the system developer so that unwanted operations are vetoed without compromising system security.

These three parts together make up the sandbox.

This is the same approach that web browsers use to restrict rights for the Java applets they run. For example, applets cannot do any file I/O, cannot connect to a host different to the one they come from, etc. All these restrictions are enforced by means of the Java security API.

3. Why not Java?

3.1. Speed

Despite their multiple advantages, interpreted languages have always had a serious drawback: they tend to be slow, due to the performance cost of run-time interpretation. This was also the case in early Java systems. Later, JIT (Just In Time) compilers got around this problem by compiling Java bytecodes to native machine code at run-time. Execution of a Java class would invoke the JIT compiler (built into the JVM) which would then compile the bytecodes to native instructions. Thus, execution would be a bit slower the first time a certain piece of code is reached, but much faster for all subsequent accesses. JIT compilers are usually optimized

towards a certain platform. Still, the overall performance is typically worse than that of compiled languages, such as C or C++.

With Intent, a JIT is no longer necessary. Java classes can be translated to native code either statically, at system design time, or dynamically, at load time. Static translation makes Java as fast as optimized C/C++ code, while dynamic translation can be used where it is necessary or convenient to retain benefits that result from the platform-neutral bytecode representation. The Intent JVM, which is certified by Sun, is believed to be the fastest in market.

3.2. Footprint

Although the Java language was originally designed for use in embedded consumer electronic applications, the first Java implementations were targeted at the desktop applications market, which has very different economies from the embedded market. Thus, most existing Java implementations are not space efficient. However, Sun's PersonalJava (PJava) and EmbeddedJava technologies have managed to shrink this large footprint down to a size which is much more suited to these kind of systems.

The Intent OS is built on a object-based programming model in which the fundamental unit is the individual *tool*. To some extent, a tool is like a function: a small section of executable code. However, whereas traditional object oriented programming requires that for a particular method to be used the entire class must be loaded, in Intent each method is an individual tool which can be called by any object. The entire set of core libraries conforming the PersonalJava technology API has been implemented using thousands of small and compact tools which are loaded and bound on demand. The net result is that the system is highly tuneable, allowing Java applications to be built and run even in the most memory constrained environment.

For example, it was possible to squeeze a complete servlet-enabled Java web server, including all necessary device drivers and support tools in a 2 MB flash device, while still leaving more than 0.5 MB for HTML pages, PDF files, servlets, applets and all kinds of user data.

3.3. Real-time behaviour?

It is critical that embedded or real-time systems are designed to cope with the timing constraints imposed by the external environment. In order to be able to deal with random, short-lived signals, these systems require fast and deterministic response times. This is a problem in Java due to two different reasons: the lack of control over the scheduling model, and the non-deterministic nature of the garbage collection process.

Lack of control over scheduling: the threading model used in Java does not provide mechanisms for fine-grained control over scheduling and dispatching, something that is very important in real-time systems. This is partly due to the high level nature of the language and to the desire to keep implementation dependencies to a minimum.

In Intent, Java threads directly map to Elate lightweight processes. Thus, it is possible to accurately define which scheduling policy to use, or even define different algorithms for different thread priority ranges, as with any other regular Elate processes.

Non-deterministic garbage collection: the memory management model in Java is based upon the garbage collection paradigm. The garbage collector (GC) is an entity that keeps track of which objects are being used (referenced) at any given time, and frees memory and resources used by those objects which are not in use anymore. Although garbage collection saves development effort and eliminates one of the main sources of bugs, it might occasionally impose long delays during program execution, something that typically interferes with real-time software requirements.

In Intent, the garbage collector is run by the JVM as a low-priority thread that tries to take advantage of idle periods resulting from natural pauses in user behaviour. The GC can be triggered on a number of different conditions: at predefined time intervals, when the working set of allocated memory reaches a high water mark, on demand, etc. A feature that is unique to Intent's JVM is that the GC process is preemptable. In other words, if there is an event that needs to be serviced while the GC is running (e.g., a network event), then the GC is interrupted, the results discarded, and the GC scheduled to resume later. The only exception is when the GC is running due to a request for memory that cannot be satisfied immediately. The fact that the GC process is preemptable alleviates to a great extent the pressures of real-time software constraints.

4. Conclusion

The fact that Java offers a huge number of advantages over other programming languages is out of question. However, most Java implementations from these last years also suffer from a number of problems, such as low speed or large memory footprint. For many applications, these problems even outweighed the potential benefits, making Java unsuitable for the case at hand.

With Intent, this scenario has undergone important changes. Many of the most important problems of traditional Java implementations have been solved or greatly alleviated with the revolutionary architecture of the Intent RTOS. The Intent JVM is believed to be the fastest in market, and its compliance with the Java published specs has been certified by Sun Microsystems. This makes Java the language of choice for fast and robust, fail-resilient, Internet-enabled applications.

Elate[®]™ is a registered trademark of Tao Group Limited.

Intent[™] is a trademark of Tao Group Limited.

Java[™] and all Java[™]-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. or other countries. Snijder Micro Systems is independent of Sun Microsystems, Inc.

OS-9[®] is a registered Trademark of Microware Systems Corporation.

VxWorks[®] is a registered Trademark of Wind River Systems Inc.

All other brand or product names are trademarks or registered trademarks of their respective holders.