A Cinnober white paper

The benefits of using Java as a high-performance language for mission critical financial applications

Cinnober Financial Technology AB
Research & Technology, 8 October 2012

Cinnober's practical experience from a decade of development

Table of contents

1 Executive summary

Trading systems today require fast execution to meet ever stricter latency requirements. For this reason we use Java as our implementation platform. Java executes as fast as or faster than any other language. Cinnober has demonstrated this by achieving single-digit microsecond latency "door-to-door" in its TRADExpress Ultra matching engine (www.cinnober.com/whitepapers).

Java provides the means for rapid development of robust, large and complex applications that are easy to extend, ensuring short time-to-market of initial system deliveries and throughout the lifetime of the system. This ensures investment protection, low maintenance and low total cost of ownership.

Cinnober is continuously following and furthering state of art of computer programming and Java allows us to use both proven and emerging technologies while meeting rigorous customer demands.

Java enables us to develop exceptionally fast software exceptionally fast.

Cinnober has focused on developing in Java from the outset in 1998. Adopting the at the time pioneering language proved to be fortuitous, helping us both to quickly take advantage of business opportunities and adapt to evolving customer demands. Our staff has over 3000 man-years of experience from developing trading and clearing systems, and we have delivered over 60 customer implementations.

These include server systems, desktop applications, web applications, as well as gateways and integration components, on a variety of platforms and languages, spanning from PCs to mainframes.

Being early adopters of Java has given us deep and extensive experience with the language and the expertise required to meet the most stringent demands.

Key areas

The key areas where Java provides advantages over other languages are as follows:

Fast execution

Java executes rapidly, since the native code of a Java program is continuously re-optimized, not only for the executing hardware, but also with regard to program usage, i.e. what parts of the software which are actually used, and how, at any given moment. There's no other commercial language that would be faster.

Time-to-market

We develop speedily with Java, since Java easily lends itself to a robust programming style. It gives high quality code from relatively short development cycles, and enables us to offer very short time-to-market periods.

There is also a great selection of high-quality development and productivity tools available for Java, all of which ease development and maintenance.

Other language environments only offer a fraction of the range of tools that are available for Java.

Large systems

Java is especially suited for large systems with long life cycles. Java encourages a lucid programming style which makes Java programs easy to maintain and support with low total cost of ownership. Cinnober has extensive experience and a world-class track record of implementing complex systems for large organizations.

Flexibility

Furthermore, the continuing development of the Java Virtual Machine (JVM) means that our products can often take advantage of new Java performance enhancements without changing anything in our code, without even re-compiling it, by simply deploying it using a new JVM.

The same holds for new hardware and operating system architectures. We have effortlessly moved from older architectures to modern ones with only superficial updates of our deployment processes.

We are confident the above will continue to hold for the foreseeable future.

White paper outline

This white paper aims to give a balanced view of the Java language and its continued role in the domain of financial software. We do not focus exclusively on the strengths of Java, but also on how Cinnober leverages Java-specific characteristics that others might call drawbacks.

Chapter 2, *Java as a high-performance language*, summarizes our views on Java.

Chapter 3, *Overview of the Java platform*, is a short introduction to the Java language from a performance perspective.

Chapter 4, *Java execution characteristics*, delves more deeply into Java.

Chapter 5, *Java and the JVM ecosystem,* provides additional material for those interested in Java.

2 Java as a high-performance language

High-performance definition

In the context of this document, high-performance refers to, but not exclusively, the characteristics:

- Execution speed
- Resource management

We will focus on these two in the rest of the paper.

Evaluation and comparison of languages

In general, it is possible to write high-quality, high-performance code in practically any language.

A problem with any comparison between programming languages is that they tend to be emotionally loaded and only prove what the author set out to prove.

When comparing different programming languages, C/C++ is often used as a benchmark since it is well known and generally provides good performance.

So let us be clear from the very beginning: there are benchmarks where Java will outperform other languages and there are benchmarks where a traditional language will be better than Java. For examples of comparative benchmarks, see *References* at the end of this chapter.

Criteria

When evaluating a programming language there are several criteria of interest, e.g. (but not limited to):

Criteria	Example
1. Run-time performance	Speed
2. Ease of development	Clarity of code, protection against mistakes, debugging
3. Ease of maintenance	Monitoring, error trapping, post-mortem debugging
4. Portability	Across time (will old code still compile and run?) Across hardware/operating system (will old code run on new systems?)

We believe that Java is outstanding on all points.

For run-time performance Java relies on the continuous profiling and re-optimization that takes place as the code is executed in the Java Virtual Machine, traditional languages on pre-execution analysis during the compilation phase. This is further described in chapter three, see *Overview of the different execution models*.

We believe that the continuous re-optimization in the JVM is the key facet of Java which gives it a decisive performance edge over traditional languages.

Language strategies for achieving high performance

High-performance Java

To achieve high performance, Java relies on:

- Static optimizations done in the source-to-byte code compiler
- Dynamic optimization done in the JIT compiler, based on run-time profiling
- Sophisticated memory management algorithms that reduce the workload

It is important to note that in Java, bad optimizations based on false assumptions (i.e. they were once true, but are now false due to changed circumstances) can always be reassessed; see also *Interpretation and compilation* in chapter four.

Modern memory management algorithms minimize the performance impact of any memory turnover; see also *Memory management* in chapter four.

High-performance C/C++

To achieve high performance, C/C++ relies on:

- Simplicity of the data/object model (no complex, implicit overhead required by the language)
- Highly advanced optimization strategies in the compiler

Taking the GNU gcc compiler as an example, the default optimization level is roughly equivalent to the one offered by Java, but it also has higher levels which Java hasn't.

This kind of optimization is called static, since it is done only once under fixed conditions.

On the other hand, the optimizations, regardless of level, rely on assumptions which, in the absence of actual run-time profiling, by necessity must be conservative. Thus, static optimization cannot be optimal.

Other languages

While many different programming languages exist today, Java and C/C++ stand out when developing server applications. Other languages might have their use in niche applications, and the future may bring new developments; see also *Using the JVM for other languages than Java* in chapter five.

Cinnober follows academic and industry developments in this area.

Cinnober strategies for achieving high performance

The Cinnober strategies for achieving high performance are mostly not specific to Java. We use common multi-threading strategies to avoid data or I/O contention in the system and achieve high parallelism.

Java-specific strategies include:

- Warm-up of the profiler
- Pre-allocating data pools
- Avoiding patterns specifically bad for Java, e.g. multi-dimensional arrays

References

Azul Systems, Java vs. C Performance...Again, http://www.azulsystems.com/blog/cliff/2009-09-06-java-vs-c-performanceagain

3 Overview of the Java platform

The following is not intended as a Java tutorial. Its purpose is to call attention to the most salient points that make us favor Java.

ALGOL syntax family

The syntax of expressions and comments belongs to the ALGOL family and is similar to C and C++. Statements of the language are immediately recognizable by a C/C++ programmer even without prior Java experience. What differences there are, can be easily learned in a short time, but Java:

- does away with many unsafe features of C/C++
- adds stronger type checking
- is fundamentally object-oriented

Java guards against many kinds of mistakes even experienced programmers make now and then. C/C++ on the other hand, while ostensibly more flexible in the sense that it gives the programmer more freedom, has minimal checks and the freedom also enables the programmer to create spectacular program bugs.

Class object model

The Java programming language is an object-oriented language. All code and data in Java, except primitives, are held by class objects. In contrast to C, which was designed to supplant assembly language, there is no provision in Java for directly accessing computer memory. There are library calls for unsafe pointer manipulation, and Real-time Java supports device driver mappings.

- Even for seemingly low-level constructs, like arrays, the language provides enforced bounds checking
- There are no pointers or references to data or parameters, only object handles, which are not directly mutable

It is thus nearly impossible to create "run-away pointers" or inadvertently overwrite memory in the way most C/C++ programmers are sadly familiar with.

Execution models

Execution model background – major types

Compiled languages

In most traditional programming languages, including C, C++, ALGOL, Fortran and COBOL, a *compiler* parses the source text and produces *machine code* appropriate for a particular computer. The compiler may employ extensive analysis of the code to produce highly optimized executable code.

Interpreted languages

An alternative approach is to have a run-time environment which either directly interprets and executes the source code, or translates the source text into some kind of *intermediate code* which is not directly executable but instead requires some dedicated environment to run it. These languages are called *interpreted* and the run-time environment an *interpreter.* The original versions of BASIC and LISP were pure interpreters.

Interpretation refinements

A refinement of interpretation is to retain the intermediate code for possible later execution. A further refinement is to add a compiler.

Dynamic languages

Dynamic languages are compiled at run time, i.e. when the program is activated. The advantage is that the compilation can be optimized for each actual run. In Java, this is implemented by using two compilers:

- An initial compiler/translator which converts the source text to intermediate code. In Java terminology, this code is called *bytecode*, and it is stored in *class files*
- *A Java Virtual Machine (JVM)*, which executes bytecode by using both:
    1. Interpretation
    2. Profiling
    3. Iterative, on-demand, compilation

The last step is known as *Just-In-Time (JIT) compilation*.

Overview of the different execution models

The picture below illustrates the differences between the interpreted, compiled, and Java dynamic execution models.

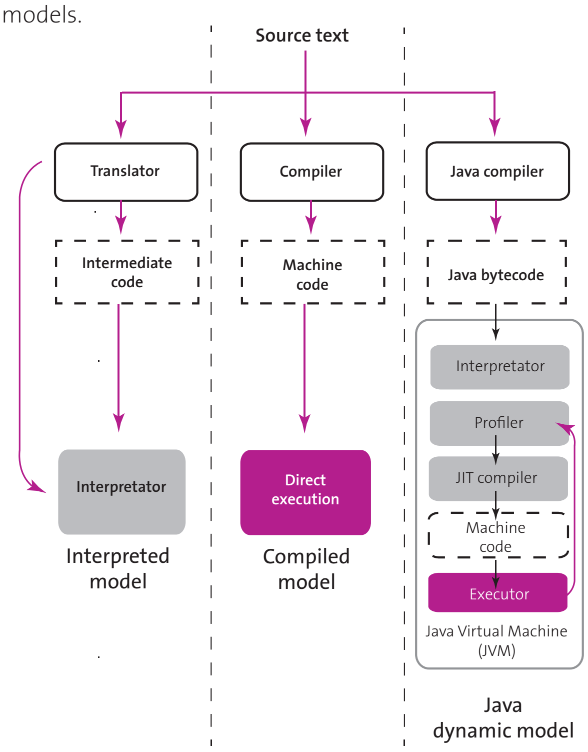

Figure 1. The differences between the interpreted, compiled, and Java dynamic execution models.

The JVM first executes the bytecode as an interpreter. After sufficient data has been collected by the profiler, the JIT compiler converts the bytecode, using metrics from the profiler, to machine code that is executed directly on the CPU. The profiler continues to work and the code may be recompiled if the program usage changes.

This continuous profiling and (re-)compilation of byte-code into machine code is the focal feature of the JVM, which enables it to achieve better execution speeds than statically compiled languages.

Some JVMs, e.g. JRockit, skip the initial interpretation altogether and instead JIT compiles directly.

Characteristics of interpretation

A pure interpreter reads each source code statement, or instruction, and immediately tries to execute it in some kind of run-time environment which keeps track of all variables and states of the program. The interpreter proceeds in this way through the entire program, executing each statement in turn as it reads.

Interpretation advantages

The chief advantages of interpretation are

- Portability
- Ease of use
- Speed of development

Interpreters are easy to port to different computer architectures or operating systems since there is no machine code involved. Producing machine code is not a trivial task; producing high-quality machine code is exceptionally complex. For this reason, programming languages are often prototyped as interpreters even if the goal is to make a compiled language.

They are easy to learn and use since it is often possible to interactively experiment with a program, or even single statements. This also leads to short development times.

Interpretation disadvantages

The chief disadvantage of interpretation is slow execution speed, compared to direct execution of machine code.

Intermediate code

Using intermediate code will speed execution, but unless the interpreter is able to retain direct machine code, an interpreted language will always be slower than a compiled language.

Exceptional cases

Please note it is always possible to demonstrate cases where a compiler can’t speed up execution, e.g. when the solution involves dynamically generating code in the language itself. Its proponents love to espouse LISP for this reason.

Characteristics of compilation

Compilation offers the opportunity to apply very sophisticated analysis of the program in order to produce high quality machine code, either optimized for speed, size or any other desired criterion. Since size is seldom an issue, unless the target is some kind of embedded computer, high speed is the most commonly desired goal.

Code optimizations

Many of the more advanced optimization techniques rely on the compiler being able to make assumptions about the execution of the code, e.g.:

- What actual CPU variant is used? Even within the same CPU family there are very significant differences between, e.g., an old Pentium 386 and a modern Nehalem-class CPU, or between Intel Xeon and AMD Opteron. By knowing the exact CPU used, the compiler may choose to use specialized instructions simply not available on other variants.
- To replace a virtual method call, where the actual method is not known, with the real call, the compiler has to examine all possible invocations in the entire source code mass. In the case of a public library, this optimization is impossible.
- Optimization might rely on the number of processors used to execute the code. This is also not possible to know in advance, unless the developer specifically gives a constraint at compile time.

Conservative assumptions

Assumptions made by a static compiler have to be conservative. On the other hand, compilation time is probably not critical. Hypothetically, we might be prepared to let a compiler run for hours, or even days, provided the speed benefits are significant.

Thus, the chief advantage of compilation is:

- Highly optimized code,

at the cost of

- slow compilation, using
- (possibly erroneous) conservative assumptions.

Characteristics of the Java dynamic model

The Java dynamic model aims to achieve all the advantages of both the interpreted and the compiled model, with the additional twist of using a built-in profiler to avoid the disadvantages of erroneous assumptions in the optimizations.

Aggressive optimizations

The JIT compiler tends to use aggressive assumptions. There is almost no drawback to this, since any erroneous assumptions will be caught by the profiler, and the code re-compiled.

Just-in-Time compilation

The compilation time is reduced by compiling only the pieces of the code which is actually used.

Whenever the JVM executes any piece of bytecode, it:

1. Checks if this piece is already compiled into machine code.
   — If so, execution is directly transferred to this code.
2. If not, the bytecode is passed to the interpreter.
3. The profiler collects data on the execution, and passes the bytecode, with profiling data, to
4. the JIT compiler.
5. The resulting machine code is cached and retained for the next time this piece of bytecode is run.

Even if a particular piece of bytecode is interpreted for a long time (e.g. stuck in a loop), this does not preclude compilation. After a given time interval, the code will be compiled into machine code, and the interpretation interrupted and replaced by direct execution of machine code.

Execution profiling

Note that profiling is not only done for interpreted code. Executed machine code is also profiled, and subject to recompilation. This process continues throughout the entire run of the program. Java execution never stops adapting to changing circumstances.

References

Java Language Specification, ISBN 978-0-321-24678-3

http://en.wikipedia.org/wiki/Compiled_language

http://en.wikipedia.org/wiki/Interpreted_language

http://en.wikipedia.org/wiki/Dynamic_compilation

http://docs.oracle.com/javase/tutorial/

http://www.javabeginner.com/

http://www.javatutorialhub.com/java-tutorial.html

http://www.javatutorialhub.com/java-virtual-machine-jvm.html

4 Java execution characteristics

The Java Virtual Machine

Hypothetical computer

It is apt to view the JVM as a virtual implementation of a hypothetical computer, which executes bytecode, rather than as an execution framework for a particular programming language, viz. Java.

Which means that although it was originally created to execute Java only, today the JVM is used for several other languages, e.g. Scala, Groovy, Jython, etc.; see also *Using the JVM for other languages than Java* in chapter five.

Write once, run anywhere

Since the bytecode is independent of the actual target hardware, the JVM fulfills one of the main goals of the Java language: To create a language truly free of hardware dependence, and which once written, could be run on any computer.

The Java authors called this feature *"Write Once, Run Anywhere"*, in contrast with the C/C++ goal of *"Write Once, Compile Anywhere"*. Compiled Java code is supposed to run on any computer which has a JVM.

Memory management

The JVM not only executes code, it also contains one or more Garbage Collectors (GC), which manage memory allocation, deallocation and defragmentation.

The requirement of the JVM to have a GC has been the source of much criticism. It is, however, our firm belief that the GC is not a problem per se. It is a solution to a problem which exists in any programming language implementing a dynamic memory model, i.e. where memory can be allocated on demand at run-time.

If memory is not managed by the language itself, it must be handled within the application logic. There are a number of ways to do this, from simple design patterns via generic "pointer classes" to full-blown memory management class libraries. This adds to the complexity of the code, and checks must be established to ensure that all code really uses the selected model.

Interpretation and compilation

Just-In-Time compilation

As outlined in *Overview of the different execution models* in chapter three, a typical JVM executes bytecode first by interpretation, then by direct execution after profiling and compilation. The JVM does not subject the entire program to this process all at once, but rather on demand: Each piece of the program is fetched and compiled as needed by the execution flow. The JVM interprets the bytecode until sufficient profiling has been collected, and the piece is then compiled.

This does not mean that there is a massive compilation phase before the system starts up. It is aptly called Just-In-Time compilation; only code really needed is compiled, and code is executed, albeit by interpretation, while compilation is performed in the background.

Start-up delay

While this strategy greatly reduces the start-up time of Java applications, it also implies that it might take a while until a program achieves full speed. For a desktop application the time is negligible and will most certainly not be noticed by the user. For a transaction processing system the delay might mean the difference between response times on the order of milliseconds compared to microseconds for an HFT trading system.

Priming the line

Thus, to achieve consistent performance from the very start of a system, we must ensure that all relevant code is already compiled when the first crucial transactions are to be processed. This can be achieved in several different ways:

It might not be an issue at all.

Many systems get enough exercise just by starting, connecting to other components, accepting logins, etc. For trading systems, pre-market phases might provide enough exercise and still not be time-critical.

You can supply a test load during a warm-up phase.

It might however be problematic to implement a test load that really exercises the system without producing any unwanted side effects. It would not make sense having the system discard the test load and not perform any business logic.

You can design the system to warm-up itself during start-up.

This adds to the complexity of the code.

Some JVMs provide a mechanism to pre-compile byte-code.

While these mechanisms might seem compelling to use, by necessity they can only use conservative optimization techniques, just like traditional static compilers. They are thus only provided by real-time JVMs, which need to produce consistent execution times across the lifetime of the program, as opposed to the highest possible performance. Also to increase consistency they disable the continuous profiling/recompilation, which would normally provide Java its distinctive performance edge.

You can run the system in interpreted mode only.

Abysmal for performance, but at least it will provide consistent execution times.

At Cinnober, we have tested all of the above and continue to test new JVM features as they are released. In general, we have found it mostly to be of no practical importance. This is not a performance issue.

Memory management

Memory management is a very general term for any programming framework used to simplify memory allocation and deallocation tasks. This is also called dynamic memory in contrast with static memory, where all memory has to be sized and allocated at compilation time. Classic languages, e.g. Fortran, are static. Any language that offers dynamic memory must have some memory management framework.

C/C++ memory management

In C, memory is allocated by the `malloc()` library routine and freed by the `free()` library routine. C++ introduced the operators `new` and `delete` for this, but since C++ is designed as a superset of C, you can still also use `malloc()` and `free()`. You shouldn't mix these methods, unless you really know what you're doing.

C/C++ characteristics

The benefits of the C/C++ model lies in its simplicity. There is very little overhead in allocating/deallocating, unless some defragmentation has to be done.

Thus, allocation and deallocation are spread throughout the code. This will incur a small but continuous execution speed penalty.

Fragmentation will eventually occur, even if the system tries to merge adjacent free memory.

The defragmentation logic, although simple, makes allocation/deallocation times non-deterministic.

Stationary objects

Objects allocated in C/C++ stay in the place they have been created. The object handles returned by the memory allocation methods are really memory addresses, which enables the programmer to do whatever he/she wishes to do with them. There is intentionally very little in the language to prevent a programmer from overwriting memory with arbitrary data.

Common application coding issues

In fact, there is very little support in C/C++ for memory management. All of the following error conditions arise from failure to correctly implement C/C++ memory management.

Memory leak

The programmer must take care to explicitly free any allocated data which is no longer needed.

Failure to do so means discarded memory cannot be re-used. If a program were to only allocate and never deallocate any data, memory would soon be exhausted.

Memory leaks are common, but since most systems are regularly restarted, they might or might not pose a problem. They can be hard to track down.

Dangling pointer

There should be no pointers left still referencing freed memory. If there are, they must not be used.

If a data object is referenced from many pieces of code or data, care must be taken to either erase all of these references or prevent them from being used. Otherwise some part of the program might access a part of memory that is no longer valid, either causing an immediate exception, or worse, reading invalid data or overwriting valid, then causing very hard-to-find errors in other parts of the system.

A dangling pointer is one of the most dreaded errors, since the actual coding error and the place where a problem occurs are probably completely unrelated.

Casting error

In C/C++ it is very common to "cast", i.e. convert pointers from one type to another. Care must be taken not to use cast pointers incorrectly.

It is very easy in C/C++, and often necessary, to cast pointers between different types. When a pointer has been cast, it is completely up to the programmer to ensure the pointer is used in a meaningful way.

Run-away index or pointer

The programmer must take utmost care not to access arrays or objects beyond their allocation limits.

An array is a consecutive sequence of objects, accessed by a pointer and an index, e.g. `obj[17]`, meaning the 18th (the first is number 0) instance in a sequence allocated at the address "`obj`". The language does not check if there really are at least 18 objects at "`obj`". There might be none or any number.

This kind of error can cause the same type of problem and be just as hard to track down as a "dangling pointer".

In general, any pointer in C/C++ can be manipulated, e.g. added to, multiplied, etc. Array indexing is just an easy way of expressing "pointer arithmetic", which is considered a very powerful feature.

Memory fragmentation

It is absolutely mandatory that a C/C++ programmer avoid the coding errors above. Memory fragmentation, on the other hand, is not possible to avoid unless the programmer implements or uses some additional memory management strategy, above and beyond what is natively provided by C/C++.

Memory fragmentation happens when individual objects, which were not allocated adjacent to each other, are freed. The resulting free memory is then not continuous. This means that even if the total amount of free memory is large, there might not be large enough pieces of free memory to allocate new objects. Note that adjacent free memory can still be defragmented.

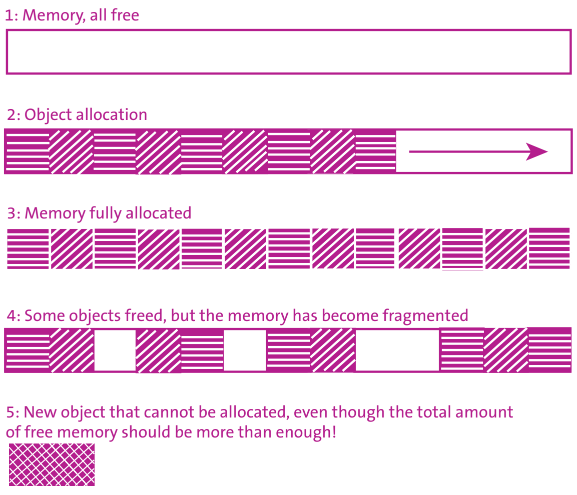
1: Memory, all free
2: Object allocation
3: Memory fully allocated
4: Some objects freed, but the memory has become fragmented
5: New object that cannot be allocated, even though the total amount of free memory should be more than enough!

Figure 2. Memory fragmentation

Memory fragmentation avoidance

The key to memory fragmentation is that allocated memory is *not movable*. If memory were to move, all references to the memory must be updated, and since C/C++ cannot keep track of memory references, it is not possible to do.

While not absolutely necessary, and generally not strictly possible, the programmer should take care to avoid memory fragmentation.

This can only be done by

- Pre-allocating objects in pools,
- avoiding dynamic allocation altogether, or
- using some memory pointer encapsulation, e.g. "smart pointers", "reference counters", etc., or even a complete Garbage Collector.

In the third case, care must be taken that the program does not mix memory models, or worse, access the pointers in a way which overrides the encapsulation.

Java memory management

Java characteristics

In summary, Java memory allocation is consistently fast as long as there is enough free memory.

Deallocation executes independently of the application.

Defragmentation executes on demand, mostly in parallel with the application, and the Java memory management does away with all the errors mentioned in *C/C++ memory management*.

Hiding pointers from the programmer

Java addresses the C/C++ issues above by completely hiding the memory pointers from the programmer.

Pointer encapsulation

All objects are referenced by "handles", which encapsulates the memory pointers. There is no way for the programmer to override the encapsulation.

This implies that there is no way to do pointer arithmetic in Java.

Memory automatically freed

Java only has a `new` operator. There is no `delete`. Memory is automatically freed when it is no longer used, i.e. when there are no handles left referencing it. Thus, a "memory leak" in the C/C++ sense cannot occur.

It is still possible to allocate memory and, by keeping unnecessary handles to the objects, prevent the memory from being freed.

Dangling pointers impossible

Since memory is only freed when no one references it, a "dangling pointer" cannot occur.

Compatible casts only

It is also not possible to convert ("cast") a handle to an incompatible type.

Casts are always checked at run-time. Incompatible casts will raise an immediate `ClassCastException`.

Array accesses always checked

Arrays can never be accessed beyond their limits.

Array element accesses are always checked at run-time. Erroneous indexing will raise an immediate `ArrayIndexOutOfBounds` exception.

Automatic defragmentation

The pointer encapsulation makes it possible to move Java objects in memory.

This is the key to how Java does memory defragmentation. Java not only knows where objects are allocated, it also keeps track of all references to them, so if an object has to be moved, all handles to it are updated too.

Defragmentation is also performed in parallel with the application logic.

The mechanism which implements memory deallocation and defragmentation is called the Garbage Collector, also known simply as the GC.

Java memory management strategies

Different strategies for different applications

There are also drawbacks to the Java model. The pointer encapsulation, various run-time checks and the execution of the Garbage Collector do incur a performance overhead.

The pointer encapsulation overhead can be mitigated by optimizations in the JIT compiler.

The GC overhead can be mitigated by choosing a GC appropriate to the problem at hand. There are a number of strategies provided by modern JVMs.

Stop-the-world GCs

The first GCs, used in LISP and Basic, were "stop-the-world", which periodically halted execution of the user program while the GC tidied up the memory. This was acceptable since these languages were (and are) typically used in applications without performance constraints. A stop-the-world GC might still be appropriate for batch-oriented Java applications, since they usually provide excellent throughput.

Concurrent GCs

Concurrent GCs sub-divide their tasks in phases which mostly run parallel with the user code. Only at certain synchronization points is the user program paused while the GC performs some critical memory update.

There are a number of concurrent GCs available in modern JVMs. They usually exhibit an excellent balance between throughput and latency.

Real-time GCs

Real-time GCs, used in Real-Time Java JVMs for real-time process control applications, operate under soft or hard real-time constraints where the critical synchronization points execute within specified time limits that are guaranteed not to exceed the scheduling limits of the user program.

Leveraging the GC at Cinnober

At Cinnober, we believe the GC is a solution to a problem which, without a GC, must be handled anyway, and within the application logic.

Understand the JVM

Any performance issues with the GC are due to limited understanding of how memory allocation occurs in Java.

Avoid memory turnover

Specifically, even in the presence of a GC, objects should still not be allocated unnecessarily. Avoid memory turnover if you're coding a high-performance application.

Tunable GC

Garbage Collection can be tuned, and by tuning the GC parameters we are able to achieve consistent GC behavior.

Zero GC

If GCs have to be completely avoided, you can adopt zero-memory-turnover programming patterns and do away with GC altogether, see references below.

References

Hirt/Lagergren, "Oracle JRockit", ISBN 978-1-847198-06-8

The JVM Specification, ISBN 978-0-201-43294-7

Boehm Garbage Collector for C/C++, http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Scheduling Garbage Collection in Embedded Systems, http://cs.lth.se/kontakt/roger_henriksson/thesis/

The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems, http://www.research.ibm.com/people/d/dfb/papers/Bacon03Metronome.pdf

Wikipedia: Garbage Collection, http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

ZeroGC, http://macgarden.dyndns.org/ZeroGCJava.html

Wikipedia: Memory leak, http://en.wikipedia.org/wiki/Memory_leak

Wikipedia: Just-in-time compilation, http://en.wikipedia.org/wiki/Just-in-time_compilation

Wikipedia: JVM, http://en.wikipedia.org/wiki/JVM

5 Java and the JVM ecosystem

Java was from the beginning designed to be a commercial language. As such, the designers wanted it to be well defined and reliable, using features that already were proven to work. The sponsor, Sun, decided to set up and fund a rigorous standardization framework, which made the specifications available to anyone but also kept firm control over the language.

These decisions made possible the emergence of a rich selection of development and productivity tools, multiple Java implementations and alternative languages taking advantage of the existing Java framework. The JVM had become an ecosystem where not only Java, but also a lot of other languages could thrive.

Development and productivity tools

The strict specifications for Java, including bytecode, class files and the JVM, not only define how to execute the code, but also provide a framework to edit, debug and profile it. Based on these common specifications, there are now a vast range of development and productivity tools available for Java. Many of these provide additional frameworks for yet another level of productivity tools, available as third-party plug-ins.

Integrated Development Environments

Integrated Development Environments (IDEs) for Java are available from a number of vendors. The major competitors are Oracle Netbeans, IBM Eclipse and Borland JBuilder, but there are many other.

A modern IDE is a highly configurable and extendable framework, into which you can install any number of native and third-party plugins for source control, code coverage, profiling, documentation, etc.

For other languages, the only major contenders are the Microsoft Visual products, and then only on Windows. Even there, the range and breadth of features come far behind the Java ecosystem.

Profiling and debugging

The JVM provides facilities for profiling and debugging which can be utilized by IDEs, plugins, or stand-alone tools. A JVM can, by its very nature as a sophisticated virtual machine, by default provide execution metrics that are not available to other languages even as extensions. After all, a JVM must have an execution analysis framework in order to do JIT compiling, which can then be adopted for external debuggers and profilers.

Java facilities like profiling, remote debugging and execution monitoring, which are natural parts of the JVM ecosystem, are only available as complicated options in other languages, if available at all.

Different JVM vendors/implementations

There are many JVM implementations to choose from. In this list we only mention a few, chiefly the more notable ones available commercially.

Oracle Hotspot, OpenJDK et al

Under Suns stewardship, the original Java implementation had evolved into several branches, Hotspot, OpenJDK and Real-time Java. Oracle received all these when Sun was acquired by Oracle. It was already an independent JVM vendor, having obtained JRockit when Oracle acquired BEA. Oracle has so far continued to support all of its Java implementations. The features of JRockit and Hotspot are supposed to gradually merge, starting from JDK 1.7.

- Hotspot is available free of charge for Windows, Solaris, Linux and Apple OSX. Support contracts are available.
- JRockit, a Real-time Java implementation, is only licensed commercially. JRockit is notable for never interpreting bytecode; it uses its JIT compiler directly.
- OpenJDK is free, available in source code, and the basis for many third-party ports.
- Sun Real-time Java is only available from Oracle's OEM team. It has been superseded by JRockit and is no longer actively marketed.

Azul

Azul was founded by some ex-Sun employees and began by implementing a specialized Java appliance, with a proprietary CPU dubbed the Vega. Its strengths are very large memory heaps (hundreds of GB) and a large number of cores. In essence, these appliances are multi-core Java co-processors that can offload Java execution from a standard host.

Azul have recently begun selling a standalone JVM, the ZingLX, which is marketed as particularly apt for low-latency applications.

- ZingLX is only licensed commercially.

IBM

IBM is notable for being one of the first JVM implementers independent of Sun/Oracle. They have provided JVMs for their own platforms from the very beginning, especially PowerPC-based, but also for Windows and Linux in the last few years.

- These are available free of charge for non-commercial applications.

Especially noteworthy are IBM's efforts to implement a Real-time JVM. They currently offer two real-time JVMs, one for soft real-time requirements, the other for hard real-time requirements.

- These are only licensed commercially.

IBM's JVMs have many interesting features. They offer

- Pre-compilation, called "ahead-of-time" (AOT)
- File-based sharing of JIT-compiled classes between JVMs
- A soft and hard real-time capable GC, dubbed "Metronome"

Also note that IBM's naming of their Java products is particularly dense:

- IBM Java is only marketed as components of their WebSphere family of products. This makes it almost meaningless to search for WebSphere Java at IBM's site, since you can't avoid getting thousands of irrelevant hits. In particular,
    - the IBM Java is called "WebSphere Java",
    - the soft real-time Java is called "WebSphere RT" or "WebSphere Real-Time", and
    - the hard real-time Java is called "WebSphere RT for RT Linux".

Using the JVM for other languages than Java

While the JVM was conceived as an execution framework for the Java language, it was almost immediately realized that the bytecode was general enough to execute any language. Proof-of-concept compilers were written, just to prove the point, but with no intent of providing a commercially viable product. There were also some features of the JVM that limited or hampered its use as a completely language-independent virtual processor.

The JVM specification has since evolved to provide better support for dynamic and other non-Java languages, and there are today a number of languages available which run on the JVM and which furthermore can be seamlessly integrated with Java code, e.g. Scala, Groovy and Clojure.

Emerging languages like these might someday prove to be commercially successful. Running Java on the JVM enables us to adopt cutting edge technologies as soon as they have shown their mettle.

References

OpenJDK,
http://openjdk.java.net/

Oracle/Hotspot,
http://www.oracle.com/technetwork/java/index.html

Oracle/JRockit,
http://www.oracle.com/jrockit

IBM/Java

- http://www.ibm.com/developerworks/java/
- http://www.ibm.com/software/webservers/real-time/
- http://www.ibm.com/developerworks/java/jdk/linux/download.html

Azul/ZingLX, http://www.azulsystems.com/products/zing

Azul/Vega, http://www.azulsystems.com/products/vega

Wikipedia on JVM languages,

Passion for change | Cinnober is the world's leading independent provider of innovative marketplace and clearing technology. Our solutions are tailored to handle high transaction volumes with assured functionality and low latency.

We are passionate about one thing: applying advanced financial technology to help trading and clearing venues seize new opportunities in times of change.

Our customers include Alpha Exchange, Borsa Italiana, BM&FBOVESPA, Burgundy, Deutsche Börse, Eurex, Hong Kong Mercantile Exchange, London Metal Exchange, Markit BOAT, NYSE Liffe, Quadriserv and Stock Exchange of Thailand.

Cinnober's solutions are based on the TRADExpress™ Platform incorporating everything needed for mission-critical solutions in terms of performance, robustness and flexibility. The portfolio of offerings includes price discovery and matching, real-time risk management, clearing and settlement, index calculation, data distribution and surveillance.

Our track record says it all. We help our customers turn change into a competitive advantage.

Cinnober®

Cinnober Financial Technology AB
Kungsgatan 36
SE-111 35 Stockholm
Sweden
+46 8 503 047 00
cinnober.com