Comparison of Java and C++

This is a **comparison of the Java programming language with the C++ programming language**.

1 Design aims

The differences between the C++ and Java programming languages can be traced to their heritage, as they have different design goals.

C++ Was designed for systems and applications programming (a.k.a. infrastructure programming), extending the C programming language. To this procedural programming language designed for efficient execution, C++ has added support for statically typed object-oriented programming, exception handling, lifetime-based resource management (RAII), generic programming, and template metaprogramming, in particular. It also added a standard library which includes generic containers and algorithms (STL), as well as many other general purpose facilities.

Java Is a general-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible. It relies on a Java virtual machine to be secure and highly portable. It is bundled with an extensive library designed to provide a complete abstraction of the underlying platform. Java is a statically typed object-oriented language that uses similar (but incompatible) syntax to C++. It includes a documentation system called Javadoc.

The different goals in the development of C++ and Java resulted in different principles and design tradeoffs between the languages. The differences are as follows :

2 Language features

2.1 Syntax

See also: Java syntax and C++ syntax

- Java syntax has a context-free grammar that can be parsed by a simple LALR parser. Parsing C++ is more complicated. For example, Foo<1>(3); is a sequence of comparisons if Foo is a variable, but creates an object if Foo is the name of a class template.

- C++ allows namespace-level constants, variables, and functions. In Java, such entities must belong to some given type, and therefore must be defined inside a type definition, either a class or an interface.

- In C++, objects are values, while in Java they are not. C++ uses *value semantics* by default, while Java always uses *reference semantics*. To opt for reference semantics in C++, either a pointer or a reference can be used.

- In C++, it is possible to declare a pointer or reference to a const object in order to prevent client code from modifying it. Functions and methods can also guarantee that they will not modify the object pointed to by a pointer by using the “const” keyword. This enforces const-correctness.

- In Java, for the most part, const-correctness must rely on the semantics of the class’ interface, i.e., it isn't strongly enforced, except for public data members that are labeled final.

- C++ supports goto statements, which may lead to Spaghetti programming. With the exception of the goto statement (which is very rarely seen in real code and highly discouraged), both Java and C++ have basically the same control flow structures, designed to enforce structured control flow, and relies on break and continue statements to provide some goto-like functionality. Some commenters point out that these labelled flow control statements break the single point-of-exit property of structured programming.[4]

- C++ provides low-level features which Java lacks. In C++, pointers can be used to manipulate specific memory locations, a task necessary for writing low-level operating system components. Similarly, many C++ compilers support an inline assembler. In Java, such code must reside in external libraries, and can only be accessed through the Java Native Interface, with a significant overhead for each call.

2.2 Semantics

- C++ allows default values for arguments of a function/method. Java does not. However, method overloading can be used to obtain similar results in Java but generate redundant stub code.
- The minimum of code you need to compile for C++ is a function. The minimum for Java is a class.
- C++ allows a range of implicit conversions between native types (including some narrowing conversions), and also allows the programmer to define implicit conversions involving user-defined types. In Java, only widening conversions between native types are implicit; other conversions require explicit cast syntax.
    - A consequence of this is that although loop conditions (if, while and the exit condition in for) in Java and C++ both expect a boolean expression, code such as if(a = 5) will cause a compile error in Java because there is no implicit narrowing conversion from int to boolean. This is handy if the code was a typo for if(a == 5). Yet current C++ compilers usually generate a warning when such an assignment is performed within a conditional expression. Similarly, standalone comparison statements, e.g. a==5;, without a side effect generate a warning.
- For passing parameters to functions, C++ supports both pass-by-reference and pass-by-value. In Java, primitive parameters are always passed by value. Class types, interface types, and array types are collectively called reference types in Java and are also always passed by value.[5][6][7]
- Java built-in types are of a specified size and range defined by the language specification. In C++, a minimal range of values is defined for built-in types, but the exact representation (number of bits) can be mapped to whatever native types are preferred on a given platform.
    - For instance, Java characters are 16-bit Unicode characters, and strings are composed of a sequence of such characters. C++ offers both narrow and wide characters, but the actual size of each is platform dependent, as is the character set used. Strings can be formed from either type.
    - This also implies that C++ compilers can automatically select the most efficient representation for the target platform (i.e., 64bit integers for a 64bit platform), while the representation is fixed in Java, meaning the values can either be stored in the less-efficient size, or must pad the remaining bits and add code to emulate the reduced-width behavior.
- The rounding and precision of floating point values and operations in C++ is implementation-defined (although only very exotic or old platforms depart from the IEEE 754 standard). Java provides an optional strict floating-point model that guarantees more consistent results across platforms, though possibly at the cost of slower run-time performance, however, Java does not provide strict compliance to the IEEE 754 standard. Most C++ compilers will, by default, partially comply to IEEE 754 standard (usually excluding strict rounding rules and raise exceptions on NaN results), but provide options for stricter compliance as well as less strict compliance (to allow for some optimizations).[8][9] If we label those options from least compliant to most compliant as *fast*, *consistent* (Java's *strictfp*), *near-IEEE*, and *strict-IEEE*, we can say that most C++ implementations default to *near-IEEE*, with options to switch to *fast* or *strict-IEEE*, while Java defaults to *fast* with an option to switch to *consistent*.
- In C++, pointers can be manipulated directly as memory address values. Java references are pointers to objects.[10] Java references do not allow direct access to memory addresses or allow memory addresses to be manipulated with pointer arithmetic. In C++ one can construct pointers to pointers, pointers to ints and doubles, and pointers to arbitrary memory locations. Java references only access objects, never primitives, other references, or arbitrary memory locations.
- In C++, pointers can point to functions or member functions (function pointers). The equivalent mechanism in Java uses object or interface references.
- Through the use of stack-allocated objects, C++ supports scoped resource management, a technique used to automatically manage memory and other system resources that supports deterministic object destruction. While scoped resource management in C++ cannot be guaranteed (even objects with proper destructors can be allocated using new and left undeleted) it provides an effective means of resource management. Shared resources can be managed using shared_ptr, along with weak_ptr to break cyclic references. Java supports automatic memory management using garbage collection which can free unreachable objects even in the presence of cyclic references, but other system resources (files, streams, windows, communication ports, threads, etc.) must be explicitly released because garbage collection is not guaranteed to occur immediately after the last object reference is abandoned.
- C++ features user-defined operator overloading. Operator overloading allows for user-defined types to support operators (arithmetic, comparisons, etc.) like primitive types via user-defined implementa-

tions for these operators. It is generally recommended to preserve the semantics of the operators. Java does not support any form of operator overloading (although its library uses the addition operator for string concatenation).

- Java features standard API support for reflection and dynamic loading of arbitrary new code.
- C++ supports static and dynamic linking of binaries.
- Java has generics, whose main purpose is to provide type-safe containers. C++ has compile-time templates, which provide more extensive support for generic programming and metaprogramming. Java has annotations, which allow adding arbitrary custom metadata to classes and metaprogramming via an annotation processing tool.
- Both Java and C++ distinguish between native types (these are also known as "fundamental" or "built-in" types) and user-defined types (these are also known as "compound" types). In Java, native types have value semantics only, and compound types have reference semantics only. In C++ all types have value semantics, but a reference can be created to any type, which will allow the object to be manipulated via reference semantics.
- C++ supports multiple inheritance of arbitrary classes. In Java a class can derive from only one class, but a class can implement multiple interfaces (in other words, it supports multiple inheritance of types, but only single inheritance of implementation).
- Java explicitly distinguishes between interfaces and classes. In C++, multiple inheritance and pure virtual functions make it possible to define classes that function almost like Java interfaces do, with a few small differences.
- Java has both language and standard library support for multi-threading. The synchronized keyword in Java provides simple and secure mutex locks to support multi-threaded applications. Java also provides robust and complex libraries for more advanced multi-threading synchronization. Only as of C++11 is there a defined memory model for multi-threading in C++, as well as library support for creating threads and for many synchronization primitives. There are also many third-party libraries for this purpose.
- C++ member functions can be declared as virtual functions, which means the method to be called is determined by the run-time type of the object (a.k.a. dynamic dispatching). By default, methods in C++ are not virtual (i.e., *opt-in virtual*). In Java, methods are virtual by default, but can be made non-virtual by using the final keyword (i.e., *opt-out virtual*).
- C++ enumerations are primitive types and support implicit conversion to integer types (but not from integer types). Java enumerations can be public static enum{enumName1,enumName2} and are used like classes. Another way is to make another class that extends java.lang.Enum<E>) and may therefore define constructors, fields, and methods as any other class. As of C++11, C++ also supports strongly-typed enumerations which provide more type-safety and explicit specification of the storage type.
- Unary operators '++' and '--': in C++ "The operand shall be a modifiable lvalue. [skipped] The result is the updated operand; it is an lvalue...",[11] but in Java "the binary numeric promotion mentioned above may include unboxing conversion and value set conversion. If necessary, value set conversion {and/or [...] boxing conversion} is applied to the sum prior to its being stored in the variable.",[12] i.e. in Java, after the initialization "Integer i=2;", "++i;" changes the reference i by assigning new object, while in C++ the object is still the same.

2.3 Resource management

- Java offers automatic garbage collection, which may be bypassed in specific circumstances via the Real time Java specification. Memory management in C++ is usually done through constructors, destructors, and smart pointers. The C++ standard permits garbage collection, but does not require it; garbage collection is rarely used in practice.
- C++ can allocate arbitrary blocks of memory. Java only allocates memory through object instantiation. Arbitrary memory blocks may be allocated in Java as an array of bytes.
- Java and C++ use different idioms for resource management. Java relies mainly on garbage collection, which can reclaim memory, while C++ relies mainly on the RAII (Resource Acquisition Is Initialization) idiom. This is reflected in several differences between the two languages:
  - In C++ it is common to allocate objects of compound types as local stack-bound variables which are destroyed when they go out of scope. In Java compound types are always allocated on the heap and collected by the garbage collector (except in virtual machines that use escape analysis to convert heap allocations to stack allocations).
  - C++ has destructors, while Java has finalizers. Both are invoked prior to an object's deallocation, but they differ significantly. A C++ object's destructor must be implicitly (in the case of stack-bound variables) or explicitly invoked

to deallocate the object. The destructor executes synchronously just prior to the point in the program at which the object is deallocated. Synchronous, coordinated uninitialization and deallocation in C++ thus satisfy the RAII idiom. In Java, object deallocation is implicitly handled by the garbage collector. A Java object's finalizer is invoked asynchronously some time after it has been accessed for the last time and before it is actually deallocated. Very few objects require finalizers; a finalizer is only required by objects that must guarantee some cleanup of the object state prior to deallocation — typically releasing resources external to the JVM.

- With RAII in C++, a single type of resource is typically wrapped inside a small class that allocates the resource upon construction and releases the resource upon destruction, and provide access to the resource in between those points. Any class that contain only such RAII objects do not need to define a destructor since the destructors of the RAII objects are called automatically as an object of this class is destroyed. In Java, safe synchronous deallocation of resources can be performed deterministically using the try/catch/finally construct.
- In C++, it is possible to have a dangling pointer—a stale reference to an object that has already been deallocated. Attempting to use a dangling pointer typically results in program failure. In Java, the garbage collector will not destroy a referenced object.
- In C++, it is possible to have uninitialized primitive objects. Java enforces default initialization.
- In C++, it is possible to have an allocated object to which there is no valid reference. Such an unreachable object cannot be destroyed (deallocated), and results in a memory leak. In contrast, in Java an object will not be deallocated by the garbage collector *until* it becomes unreachable (by the user program). (Note: *weak references* are supported, which work with the Java garbage collector to allow for different *strengths* of reachability.) Garbage collection in Java prevents many memory leaks, but leaks are still possible under some circumstances.[13][14][15]

2.4 Libraries

- C++ provides cross-platform access to many features typically available in platform-specific libraries. Direct access from Java to native operating system and hardware functions requires the use of the Java Native Interface.

2.5 Runtime

- Due to its unconstrained expressiveness, low level C++ language features (e.g. unchecked array access, raw pointers, type punning) cannot be reliably checked at compile-time or without overhead at run-time. Related programming errors can lead to low-level buffer overflows and segmentation faults. The Standard Template Library provides higher-level RAII abstractions (like vector, list and map) to help avoid such errors. In Java, low level errors either cannot occur or are detected by the JVM and reported to the application in the form of an exception.
- The Java language requires specific behavior in the case of an out-of-bounds array access, which generally requires bounds checking of array accesses. This eliminates a possible source of instability but usually at the cost of slowing down execution. In some cases, especially since Java 7, compiler analysis can prove a bounds check unnecessary and eliminate it. C++ has no required behavior for out-of-bounds access of native arrays, thus requiring no bounds checking for native arrays. C++ standard library collections like std::vector, however, offer optional bounds checking. In summary, Java arrays are "usually safe; slightly constrained; often have overhead" while C++ native arrays "have optional overhead; are slightly unconstrained; are possibly unsafe."

2.6 Templates vs. generics

Both C++ and Java provide facilities for generic programming, templates and generics, respectively. Although they were created to solve similar kinds of problems, and have similar syntax, they are actually quite different.

2.7 Miscellaneous

- Java and C++ use different techniques for splitting up code in multiple source files. Java uses a package system that dictates the file name and path for all program definitions. In Java, the compiler imports the executable class files. C++ uses a header file source code inclusion system for sharing declarations between source files.
- Compiled Java code files are generally smaller than code files in C++ as Java bytecode is usually more compact than native machine code and Java programs are never statically linked.
- C++ compilation features an additional textual preprocessing phase, while Java does not. Thus some users add a preprocessing phase to their build

process for better support of conditional compilation.

- Java's division and modulus operators are well defined to truncate to zero. C++ (prior to C++11) does not specify whether or not these operators truncate to zero or "truncate to -infinity". $-3/2$ will always be -1 in Java and C++11, but a C++03 compiler may return either -1 or -2, depending on the platform. C99 defines division in the same fashion as Java and C++11. Both languages guarantee (where a and b are integer types) that (a/b)*b + (a%b) == a for all a and b (b != 0). The C++03 version will sometimes be faster, as it is allowed to pick whichever truncation mode is native to the processor.

- The sizes of integer types are defined in Java (int is 32-bit, long is 64-bit), while in C++ the size of integers and pointers is compiler and ABI dependent within given constraints. Thus a Java program will have consistent behavior across platforms, whereas a C++ program may require adaptation for certain platforms, but may run faster with more natural integer sizes for the local platform.

An example comparing C++ and Java exists in Wikibooks.

3 Performance

In addition to running a compiled Java program, computers running Java applications generally must also run the Java virtual machine (JVM), while compiled C++ programs can be run without external applications. Early versions of Java were significantly outperformed by statically compiled languages such as C++. This is because the program statements of these two closely related languages may compile to a few machine instructions with C++, while compiling into several byte codes involving several machine instructions each when interpreted by a JVM. For example:

Since performance optimization is a very complex issue, it is very difficult to quantify the performance difference between C++ and Java in general terms, and most benchmarks are unreliable and biased. And given the very different natures of the languages, definitive qualitative differences are also difficult to draw. In a nutshell, there are inherent inefficiencies as well as hard limitations on optimizations in Java given that it heavily relies on flexible high-level abstractions, however, the use of a powerful JIT compiler (as in modern JVM implementations) can mitigate some issues. And, in any case, if the inefficiencies of Java are too much to bear, compiled C or C++ code can be called from Java by means of the JNI.

Certain inefficiencies that are inherent to the Java language itself include, primarily:

- All objects are allocated on the heap. For functions using small objects this can result in performance degradation and heap fragmentation, while stack allocation, in contrast, costs essentially zero. However, modern JIT compilers mitigate this problem to some extent with escape analysis or escape detection to allocate objects on the stack, since Oracle JDK 6.

- Methods are virtual by default (although they can be made final), usually leading to an abuse of virtual methods, adding a level of indirection to every call. This also slightly increases memory usage by adding a single pointer to a virtual table per each object. It also induces a start-up performance penalty, since a JIT compiler must perform additional optimization passes for de-virtualization of small functions.

- A lot of run-time casting required even using standard containers induces a performance penalty. However, most of these casts are statically eliminated by the JIT compiler.

- Safety guarantees come at a run-time cost. For example, the compiler is required to put appropriate range checks in the code. Guarding each array access with a range check is not efficient, so most JIT compilers will try to eliminate them statically or by moving them out of inner loops (although most native compilers for C++ will do the same when range-checks are optionally used).

- Lack of access to low-level details prevents the developer from improving the program where the compiler is unable to do so.[19]

- The mandatory use of reference-semantics for all user-defined types in Java can introduce large amounts of superfluous memory indirections (or jumps) (unless elided by the JIT compiler) which can lead to frequent cache misses (a.k.a. cache thrashing). Furthermore, cache-optimization, usually via cache-aware or cache-oblivious data structures and algorithms, can often lead to orders of magnitude improvements in performance as well as avoiding time-complexity degeneracy that is characteristic of many cache-pessimizing algorithms, and is therefore one of the most important forms of optimization; reference-semantics, as mandated in Java, makes such optimizations impossible to realize in practice (by neither the programmer nor the JIT compiler).

- Garbage collection,[20] as this form of automatic memory management introduces memory overhead.[21]

However, there are a number of benefits to Java's design, some realized, some only theorized:

- Java garbage collection may have better cache coherence than the usual usage of *malloc*/*new* for memory allocation. Nevertheless, arguments exist that both allocators equally fragment the heap and neither exhibits better cache locality. However, in C++, allocation of single objects on the heap is rare, and large quantities of single objects are usually allocated in blocks via an STL container and/or with a small object allocator.[22][23]

- Run-time compilation can potentially use information about the platform on which the code is being executed to improve code more effectively. However, most state-of-the-art native (C, C++, etc.) compilers generate multiple code paths to employ the full computational abilities of the given system.[24] Additionally, the inverse argument can be made that native compilers can better exploit architecture-specific optimizations and instruction sets than multi-platform JVM distributions.

- Run-time compilation allows for more aggressive virtual function inlining than is possible for a static compiler, because the JIT compiler has more information about all possible targets of virtual calls, even if they are in different dynamically loaded modules. Currently available JVM implementations have no problem in inlining most of the monomorphic, mostly monomorphic and dimorphic calls, and research is in progress to inline also megamorphic calls, thanks to the recent invoke dynamic enhancements added in Java 7.[25] Inlining can allow for further optimisations like loop vectorisation or loop unrolling, resulting in a huge overall performance increase.

- In Java, thread synchronization is built into the language, so the JIT compiler can potentially, through escape analysis, elide locks,[26] significantly improve the performance of naive multi-threaded code. This technique was introduced in Sun JDK 6 update 10 and is named biased locking.[27]

Additionally, some performance problems exist in C++ as well:

- Allowing pointers to point to any address can make optimization difficult due to the possibility of interference between pointers that alias each other. However, the introduction of *strict-aliasing* rules largely solves this problem.[28]

- Since the code generated from various instantiations of the same class template in C++ is not shared (as with type-erased generics in Java), excessive use of templates may lead to significant increase of the executable code size (a.k.a. code bloat). However, because function templates are aggressively inlined, they can sometimes reduce code bloat but more importantly allow for more aggressive static analysis and code optimization by the compiler, more often making them more efficient than non-templated code, while, by contrast, Java generics are necessarily less efficient than non-genericized code.

- Because dynamic linking is performed after code generation and optimization in C++, function calls spanning different dynamic modules cannot be inlined.

- Because thread support is generally provided by libraries in C++, C++ compilers cannot perform thread-related optimizations. However, since the introduction of multi-threading memory models in C++11, modern compilers have the necessary language features to implement such optimizations. Furthermore, many optimizing compilers, such as the Intel compiler, provide several language extensions and advanced threading facilities for professional multi-threading development.

4 Official standard and reference of the language

4.1 Language specification

The C++ language is defined by *ISO/IEC 14882*, an ISO standard, which is published by the *ISO/IEC JTC1/SC22/WG21* committee. The latest, post-standardization draft of C++11 is available as well.[29]

The C++ language evolves through an open steering committee called the C++ Standards Committee. The committee is composed of the creator of C++ Bjarne Stroustrup, the convener Herb Sutter, and other prominent figures, including many representatives of industries and user-groups (i.e., the stake-holders). Being an open committee, anyone is free to join, participate, and contribute proposals for upcoming releases of the standard and technical specifications. The committee now aims to release a new standard every few years, although in the past strict review processes and discussions have meant longer delays between publication of new standards (1998, 2003, and 2011).

The Java language is defined by the *Java Language Specification*,[30] a book which is published by Oracle.

The Java language continuously evolves through a process called the Java Community Process, and the world's programming community is represented by a group of people and organizations - the Java Community members[31]— which is actively engaged into the enhancement of the language, by sending public requests - the Java Specification Requests - which must pass formal and public reviews before they get integrated into the language.

The lack of a firm standard for Java and the somewhat more volatile nature of its specifications have been a con-

stant source of criticism by stake-holders wanting more stability and more conservatism in the addition of new language and library features. On the other hand, C++ committee also receives constant criticism for the opposite reason, i.e., being too strict and conservative, and taking too long to release new versions.

4.2 Trademarks

"C++" is not a trademark of any company or organization and is not owned by any individual.[32] "Java" is a trademark of Oracle Corporation.[33]

5 References

[1] "The Java Tutorials: Passing Information to a Method or a Constructor". Oracle. Retrieved 17 February 2013.

[2] "The Java Tutorials: Object as a Superclass". Oracle. Retrieved 17 February 2013..

[3] "XMPP Software » Libraries". xmpp.org. Retrieved 13 June 2013.

[4] Robert C. Martin (January 1997). "Java vs. C++: A Critical Comparison" (PDF).

[5] "Reference Types and Values". *The Java Language Specification, Third Edition*. Retrieved 9 December 2010.

[6] Horstmann, Cay; Cornell, Gary (2008). *Core Java* **I** (Eighth ed.). Sun Microsystems. pp. 140–141. ISBN 978-0-13-235476-9. Some programmers (and unfortunately even some book authors) claim that the Java programming language uses call by reference for objects. However, that is false. Because this is such a common misunderstanding, it is worth examining a counterexample in some detail... This discussion demonstrates that the Java programming language does not use call by reference for objects. Instead *object references are passed by value*.

[7] Deitel, Paul; Deitel, Harvey (2009). *Java for Programmers*. Prentice Hall. p. 223. ISBN 978-0-13-700129-3. Unlike some other languages, Java does not allow programmers to choose pass-by-value or pass-by-reference—all arguments are passed by value. A method call can pass two types of values to a method—copies of primitive values (e.g., values of type int and double) and copies of references to objects (including references to arrays). Objects themselves cannot be passed to methods.

[8] "Semantics of Floating Point Math in GCC". GNU Foundation. Retrieved 20 April 2013.

[9] "Microsoft c++ compiler, /fp (Specify Floating-Point Behavior)". Microsoft Corporation. Retrieved 19 March 2013.

[10] "Java Language Specification 4.3.1: Objects". Sun Microsystems. Retrieved 9 December 2010.

[11] Standard for Programming Language C++ '11, 5.3.2 Increment and decrement [expr.pre.incr].

[12] The Java™ Language Specification, Java SE 7 Edition, Chapters 15.14.2 , 15.14.3, 15.15.1, 15.15.2, http://docs.oracle.com/javase/specs/

[13] Satish Chandra Gupta, Rajeev Palanki (16 August 2005). "Java memory leaks -- Catch me if you can". IBM DeveloperWorks. Archived from the original on 2012-07-22. Retrieved 2015-04-02.

[14] How to Fix Memory Leaks in Java by Veljko Krunic (Mar 10, 2009)

[15] Creating a memory leak with Java on stackoverflow.com

[16] http://en.cppreference.com/w/cpp/language/type_alias

[17] http://en.cppreference.com/w/cpp/language/variable_template

[18] Boost type traits library

[19] Clark, Nathan; Amir Hormati; Sami Yehia; Scott Mahlke (2007). "Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping". *HPCA'07*: 216–227.

[20] Hundt, Robert (2011-04-27). "Loop Recognition in C++/Java/Go/Scala" (PDF; 318 kB). Stanford, California: Scala Days 2011. Retrieved 2012-11-17. *Java shows a large GC component, but a good code performance. [...] We find that in regards to performance, C++ wins out by a large margin. [...] The Java version was probably the simplest to implement, but the hardest to analyze for performance. Specifically the effects around garbage collection were complicated and very hard to tune*

[21] Matthew Hertz, Emery D. Berger (2005). "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management" (PDF). OOPSLA 2005. Retrieved 2015-03-15. *In particular, when garbage collection has five times as much memory as required, its runtime performance matches or slightly exceeds that of explicit memory management. However, garbage collection's performance degrades substantially when it must use smaller heaps. With three times as much memory, it runs 17% slower on average, and with twice as much memory, it runs 70% slower.*

[22] Alexandrescu, Andrei (2001). Addison-Wesley, ed. *Modern C++ Design: Generic Programming and Design Patterns Applied. Chapter 4*. pp. 77–96. ISBN 978-0-201-70431-0.

[23] "Boost Pool library". Boost. Retrieved 19 April 2013.

[24] Targeting IA-32 Architecture Processors for Run-time Performance Checking

[25] Fixing The Inlining "Problem" by Dr. Cliff Click | Azul Systems: Blogs

[26] Oracle Technology Network for Java Developers

[27] Oracle Technology Network for Java Developers

[28] Understanding Strict Aliasing - CellPerformance

[29] "Working Draft, Standard for Programming Language C++" (PDF).

[30] The Java Language Specification

[31] The Java Community Process(SM) Program - Participation - JCP Members

[32] Bjarne Stroustrup's FAQ: Do you own C++?

[33] ZDNet: Oracle buys Sun; Now owns Java.

6 External links

- Object Oriented Memory Management: Java vs. C++
- Chapter 2:How Java Differs from C, chapter from Java in a Nutshell by David Flanagan
- Java vs. C++ resource management comparison - Comprehensive paper with examples
- Java vs C performance... again... - In-depth discussion of differences between Java and C / C++ with regard to performance.

7 Text and image sources, contributors, and licenses

7.1 Text

- **Comparison of Java and C++** *Source:* https://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B?oldid=698742353 *Contributors:* Damian Yerrick, Derek Ross, Wesley, Bryan Derksen, Tarquin, Sjc, Ed Poor, Sfmontyo, Hirzel, Frecklefoot, Edward, Tim Starling, Kwertii, Darkwind, Zmandel, Pengo, Giftlite, Ævar Arnfjörð Bjarmason, Netoholic, Alterego, Curps, CyborgTosser, Prosfilaes, Neilc, OverlordQ, Saucepan, Rdsmith4, Karl Dickman, Squash, Grunt, Rodrigostrauss, Guppyfinsoup, Discospinster, CanisRufus, Marco255, Root4(one), Spoon!, Themusicgod1, Nigelj, War Famine Pestilence Death, Mathieu, Martin Fuchs, Molteanu, Wtmitchell, Schapel, Forderud, Kbolino, Lkinkade, Dustball, Urod, Esben~enwiki, Pfunk42, E090, BD2412, Rjwilmsi, MarSch, XP1, Gudeldar, Mazzmn, Mirror Vax, Ysangkok, Kmorozov, DevastatorIIC, Miffy900, Chobot, Mysekurity, Peterl, DerrickOswald, Wavelength, Eelis.net, Barefootguru, Bovineone, Michael Trigoboff, Catamorphism, JulesH, Zzuuzz, Theda, Cedar101, Juliano, Donhalcon, Curpsbot-unicodify, Chip Zero, David Wahler, Erik Sandberg, SmackBot, Kilo-Lima, ActiveSelective, Super Quinn, Gilliam, Kazkaskazkasako, Chris the speller, Bluebot, Flums, Nbarth, Gamahucheur, Michael.Pohoreski, Kittybrewster, Cybercobra, Decltype, SeverityOne, JonathanWakely, Weregerbil, Bdiscoe, Daquell, JavaKid, Doug Bell, Harryboyles, Loadmaster, Rainwarrior, Dicklyon, Hans Bauer, Saxton, RekishiEJ, Buckyboy314, Rengolin, Aandu, Requestion, Ezrakilty, Chrisahn, Cydebot, Brindy666, Peterdjones, Pascal.Tesson, Alexnye, Christian75, Johntabularasa, Hervegirod, Pcu123456789, EmeryD, Medinoc, CmdrRickHunter, MartinDK, Ees, Ivec, King Mir, Tiagofassoni, A3nm, Gwern, Hsz, Nono64, Tgeairn, Veritas Blue, Alex Heinz, Solarswordsman, Cfeet77, Bdodo1992, Mikon, Adamd1008, Vqwj, Ranjyad, Lear's Fool, Dangiankit, Crashie, Jayaram ganapathy, EverGreg, SlitherM, Wrldwzrd89, EdSquareCat, Kexyn, Jespdj, Flyer22 Reborn, Timgurto, Ctxppc, AlanUS, Mwn3d, Tronic2, The Thing That Should Not Be, Garyzx, Mild Bill Hiccup, Mpd en, TobiasPersson, Daniel-Pharos, Aprock, Bjdehut, Thymefromti, DumZiBoT, Mjharrison, Cardatron, Dthomsen8, Owl order, Addbot, Mortense, CodedoC, Diptanshu.D, Teles, Jarble, Luckas-bot, Yobot, Fraggle81, Tundra010, AnomieBOT, 1exec1, Citation bot, Xqbot, The Elves Of Dunsimore, Reckjavik, Oddityoverseer13, Dhendriks123, Shattered Gnome, Alainr345, Wikiresearchman, Shadowjams, Xehpuk, FrescoBot, LCID Fire, Thenarrowmargin, McNepp, Vertexua, Sebastiangarth, Sae1962, I dream of horses, GeirGrusom, Hgb asicwizard, LogAntiLog, Sussexonian, KardinalKill, Reaper Eternal, EYOLs, Amkilpatrick, Specs112, LoStrangolatore, RjwilmsiBot, XNinto, WildBot, Zertyz, Faolin42, Klbrain, Wikitoov, Cogiati, Aavindraa, Donner60, Ipsign, Zafarella, ClueBot NG, Shaddim, Millermk, Cntras, AgniKalpa, Widr, Ben morphett, Antiqueight, Helpful Pixie Bot, BG19bot, Bmusician, Otyugh~enwiki, Adityarajbhatt, Jhagelgans, Ricky6565, Alexvreidy, BattyBot, ChrisGualtieri, TheJJJunk, Muffins94, Heist rulz, Lugia2453, Frosty, SFK2, Sriharsh1234, Telfordbuck, Dhs India, Alex Yursha, Mikael.s.persson, JustBerry, My name is not dave, Cc11rocks, Monkbot, Cvedauwoo, ChristianGaertner, Nelsonkam, Trevor the java programmer, H662, FindMeLost and Anonymous: 332

7.2 Images

- **File:Ambox_important.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/b/b4/Ambox_important.svg *License:* Public domain *Contributors:* Own work, based off of Image:Ambox scales.svg *Original artist:* Dsmurat (talk · contribs)
- **File:Edit-clear.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/f/f2/Edit-clear.svg *License:* Public domain *Contributors:* The *Tango! Desktop Project*. *Original artist:*

  The people from the Tango! project. And according to the meta-data in the file, specifically: "Andreas Nilsson, and Jakub Steiner (although minimally)."
- **File:Question_book-new.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg *License:* Cc-by-sa-3.0 *Contributors:*

  Created from scratch in Adobe Illustrator. Based on Image:Question book.png created by User:Equazcion *Original artist:*

  Tkgd2007
- **File:Text_document_with_red_question_mark.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/a/a4/Text_document_with_red_question_mark.svg *License:* Public domain *Contributors:* Created by bdesham with Inkscape; based upon Text-x-generic.svg from the Tango project. *Original artist:* Benjamin D. Esham (bdesham)
- **File:Unbalanced_scales.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/f/fe/Unbalanced_scales.svg *License:* Public domain *Contributors:* ? *Original artist:* ?
- **File:Wikibooks-logo-en-noslogan.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/d/df/Wikibooks-logo-en-noslogan.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Bastique, User:Ramac et al.

7.3 Content license

- Creative Commons Attribution-Share Alike 3.0