WIKIPEDIA

Comparison of Java and C++

This is a **comparison of the programming languages Java and C++**.

Contents

Design aims

The differences between the programming languages C++ and Java 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 procedural programming language C, which was designed for efficient execution. To C, C++ added support for object-oriented programming, exception handling, lifetime-based resource management (RAII), generic programming, template metaprogramming, and the C++ Standard Library which includes generic containers and algorithms (STL), and many other general purpose facilities.

Java is a general-purpose, concurrent, class-based, object-oriented programming language that is designed to minimize implementation dependencies. It relies on a Java virtual machine to be secure and highly portable. It is bundled with an extensive library designed to provide a full abstraction of the underlying platform. Java is a statically typed object-oriented language that uses a syntax similar (but incompatible) 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 trade-offs between the languages. The differences are as follows:

C++	Java
Extends C with object-oriented programming and generic programming. C code can most properly be used.	Strongly influenced by C++/C syntax.
Compatible with C source code, except for a few corner cases.	Provides the Java Native Interface and recently Java Native Access as a way to directly call C/C++ code.
Write once, compile anywhere (WOCA).	Write once, run anywhere/everywhere (WORA/WORE).
Allows procedural programming, functional programming, object-oriented programming, generic programming, and template metaprogramming. Favors a mix of paradigms.	Allows procedural programming, functional programming (since Java 8) and generic programming (since Java 5), but strongly encourages the object-oriented programming paradigm. Includes support for creating scripting languages.
Runs as native executable machine code for the target instruction set(s).	Runs on a virtual machine.
Provides object types and type names. Allows reflection via run-time type information (RTTI).	Is reflective, allowing metaprogramming and dynamic code generation at runtime.
Has multiple binary compatibility standards (commonly Microsoft (for MSVC compiler) and Itanium/GNU (for almost all other compilers)).	Has one binary compatibility standard, cross-platform for OS and compiler.
Optional automated bounds checking (e.g., the `at()` method in `vector` and `string` containers).	All operations are required to be bound-checked by all compliant distributions of Java. HotSpot can remove bounds checking.
Native unsigned arithmetic support.	Native unsigned arithmetic unsupported. Java 8 changes some of this, but aspects are unclear.[1]
Standardized minimum limits for all numerical types, but the actual sizes are implementation-defined. Standardized types are available via the standard library `<cstdint>`.	Standardized limits and sizes of all primitive types on all platforms.
Pointers, references, and pass-by-value are supported for all types (primitive or user-defined).	All types (primitive types and reference types) are always passed by value.[2]
Memory management can be done manually via `new / delete`, automatically by scope, or by smart pointers. Supports deterministic destruction of objects. Garbage collection ABI standardized in C++11, though compilers are not required to implement garbage collection.	Automatic garbage collection. Supports a non-deterministic finalize() method use of which is not recommended.[3]
Resource management can be done manually or by automatic lifetime-based resource management (RAII).	Resource management must generally be done manually, or automatically via finalizers, though this is generally discouraged. Has try-with-resources for automatic scope-based resource management (version 7 onwards). It can also be done using the internal API `sun.misc.Unsafe` but that usage is highly discouraged and will be replaced by a public API in an upcoming Java version.
Supports classes, structs (passive data structure (PDS) types), and unions, and can allocate them on the heap or the stack.	Classes are allocated on the heap. Java SE 6 optimizes with escape analysis to allocate some objects on the stack.
Allows explicitly overriding types, and some implicit narrowing conversions (for compatibility with C).	Rigid type safety except for widening conversions.
The C++ Standard Library was designed to have a limited scope and functions, but includes language support, diagnostics, general utilities, strings, locales, containers, algorithms, iterators, numerics, input/output, random number generators, regular expression parsing, threading facilities, type traits (for static type introspection) and Standard C Library. The Boost library offers more functions including network I/O.	The standard library has grown with each release. By version 1.6, the library included support for locales, logging, containers and iterators, algorithms, GUI programming (but not using the system GUI), graphics, multi-threading, networking, platform security, introspection, dynamic class loading, blocking and non-blocking I/O. It provided interfaces or support classes for XML, XSLT, MIDI, database connectivity, naming services (e.g. LDAP), cryptography, security services (e.g. Kerberos), print services, and web services. SWT offered an abstraction for platform-specific

A rich amount of third-party libraries exist for GUI and other functions like: Adaptive Communication Environment (ACE), Crypto++, various XMPP Instant Messaging (IM) libraries,[4] OpenLDAP, Qt, gtkmm.	GUIs, but was superseded by JavaFX in the latest releases ; allowing for graphics acceleration and CSS-themable UIs. It although doesn't support any kind of "native platform look" support.
Operator overloading for most operators. Preserving meaning (semantics) is highly recommended.	Operators are not overridable. The language overrides + and += for the String class.
Single and Multiple inheritance of classes, including virtual inheritance.	Single inheritance of classes. Supports multiple inheritance via the Interfaces construct, which is equivalent to a C++ class composed of abstract methods.
Compile-time templates. Allows for Turing complete meta-programming.	Generics are used to achieve basic type-parametrization, but they do not translate from source code to byte code due to the use of type erasure by the compiler.
Function pointers, function objects, lambdas (in C++11), and interfaces.	Functions references, function objects and lambdas were added in Java 8. Classes (and interfaces, which are classes) can be passed as references as well through `SomeClass.class`
No standard inline documentation mechanism. Third-party software (e.g. Doxygen) exists.	Extensive Javadoc documentation standard on all system classes and methods.
`const` keyword for defining immutable variables and member functions that do not change the object. Const-ness is propagated as a means to enforce, at compile-time, correctness of the code with respect to mutability of objects (see const-correctness).	`final` provides a version of `const`, equivalent to `type* const` pointers for objects and `const` for primitive types. Immutability of object members achieved via read-only interfaces and object encapsulation.
Supports the `goto` statement.	Supports labels with loops and statement blocks. `goto` is a reserved keyword but is marked as "unused" in the Java specification. (https://docs.oracle.com/javase/specs/)
Source code can be written to be cross-platform (can be compiled for Windows, BSD, Linux, macOS, Solaris, etc., without modification) and written to use platform-specific features. Typically compiled into native machine code, must be recompiled for each target platform.	Compiled into byte code for the JVM. Byte code is dependent on the Java platform, but is typically independent of operating system specific features.

Language features

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.

C++	Java

`class Foo {  // Declares class Foo`
`  int x = 0;  //  Private Member variable. It will`
`  // be initialized to 0, if the`
`  // constructor would not set it.`
`  // (from C++11)`
`public:`
`  Foo() : x(0)  //  Constructor for Foo; initializes`
`  {}  //  x to 0. If the initializer were`
`  //  omitted, the variable would`
`  //  be initialized to the value that`
`  // has been given at declaration of x.`
`  int bar(int i) { // Member function bar()`
`  return 3*i + x;`
`  }`
`};`
	`class Foo {  // Defines class Foo`
	`  private int x;  // Member variable, normally declared`
	`  // as private to enforce encapsulation`
	`  // initialized to 0 by default`
	`  public Foo() {  // Constructor for Foo`
	`  }  // no-arg constructor supplied by default`
	`  public int bar(int i) {  // Member method bar()`
	`  return 3*i + x;`
	`  }`
	`}`

`Foo a;`
`// declares a to be a Foo object value,`
`// initialized using the default constructor.`
`// Another constructor can be used as`
`Foo a(args);`
`// or (C++11):`
`Foo a{args};`
	`Foo a = new Foo();`
	`// declares a to be a reference to a new Foo object`
	`// initialized using the default constructor`
	`// Another constructor can be used as`
	`Foo a = new Foo(args);`

`Foo b = a;`
`// copies the contents of a to a new Foo object b;`
`// alternative syntax is "Foo b(a)"`
	`// Foo b = a;`
	`// would declare b to be reference to the object pointed to by a`
	`Foo b = a.clone();`
	`// copies the contents of the object pointed to by a`
	`//  to a new Foo object;`
	`// sets the reference b to point to this new object;`
	`// the Foo class must implement the Cloneable interface`
	`//  for this code to compile`

`a.x = 5; // modifies the object a`	`a.x = 5; // modifies the object referenced by a`

`std::cout << b.x << std::endl;`
`// outputs 0, because b is`
`// some object other than a`
	`System.out.println(b.x);`
	`// outputs 0, because b points to`
	`// some object other than a`

`Foo *c;`
`// declares c to be a pointer to a`
`// Foo object (initially`
`// undefined; could point anywhere)`
	`Foo c;`
	`// declares c to be a reference to a Foo`
	`// object (initially null if c is a class member;`
	`// it is necessary to initialize c before use`
	`// if it is a local variable)`

`c = new Foo;` `// binds c to point to a new Foo object`	`c = new Foo();` `// binds c to reference a new Foo object`
`Foo &d = *c;` `// binds d to reference the same object to which c points`	`Foo d = c;` `// binds d to reference the same object as c`

`c->x = 5;` `// modifies the object referenced by c`	`c.x = 5;` `// modifies the object referenced by c`

`a.bar(5);  // invokes Foo::bar() for a`
`c->bar(5); // invokes Foo::bar() for *c`
	`a.bar(5); // invokes Foo.bar() for a`
	`c.bar(5); // invokes Foo.bar() for c`

`std::cout << d.x << std::endl;`
`// outputs 5, because d references the`
`// same object to which c points`
	`System.out.println(d.x);`
	`// outputs 5, because d references the`
	`// same object as c`

- 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 is not strongly enforced, except for public data members that are labeled `final`.

C++	Java

`const Foo *a; // it is not possible to modify the object`
`// pointed to by a through a`
	`final Foo a; // a declaration of a "final" reference:`
	`// it is possible to modify the object,`
	`// but the reference will constantly point`
	`// to the first object assigned to it`

`a = new Foo();`	`a = new Foo(); // Only in constructor`

`a->x = 5;`
`// ILLEGAL`
	`a.x = 5;`
	`// LEGAL, the object's members can still be modified`
	`// unless explicitly declared final in the declaring class`

`Foo *const b = new Foo();` `// a declaration of a "const" pointer`	`final Foo b = new Foo();` `// a declaration of a "final" reference`
`b = new Foo();` `// ILLEGAL, it is not allowed to re-bind it`	`b = new Foo();` `// ILLEGAL, it is not allowed to re-bind it`
`b->x = 5;` `// LEGAL, the object can still be modified`	`b.x = 5;` `// LEGAL, the object can still be modified`

- C++ supports goto statements, which may lead to spaghetti code 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 functions. Some commenters point out that these labelled flow control statements break the single point-of-exit property of structured programming.[5]

- 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 via the Java Native Interface, with a significant overhead for each call.

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 needed to compile for C++ is a function, for Java is a class.
- C++ allows a range of implicit conversions between native types (including some narrowing conversions), and also allows defining implicit conversions involving user-defined types. In Java, only widening conversions between native types are implicit; other conversions require explicit cast syntax.
  - A result 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.[6][7][8]
- 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., 64-bit integers for a 64-bit 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* (strictfp) that guarantees more consistent results across platforms, though at the cost of possibly slower run-time performance. However, Java does not comply strictly with the IEEE 754 standard. Most C++ compilers will, by default, comply partly with IEEE 754 (usually excluding strict rounding rules and raise exceptions on NaN results), but provide compliance options of varied strictness, to allow for some optimizing.[9][10] 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.[11] 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.
- Via 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 implementations for these operators. It is generally recommended to preserve the semantics of the operators. Java supports no form of operator overloading (although its library uses the addition operator for string concatenation).
- Java features standard application programming interface (API) support for reflection and dynamic loading of arbitrary new code.
- C++ supports static and dynamic linking of binaries.
- Java has generics, which 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 (also termed *fundamental* or *built-in* types) and user-defined types (also termed *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 synchronizing. Only as of C++11 is there a defined memory model for multi-threading in C++, and library support for creating threads and for many synchronizing primitives. There are also many third-party libraries for this.
- 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...",[12] 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.",[13] 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.

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 via 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 via 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 Resource Acquisition Is Initialization (RAII) 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 before an object's deallocation, but they differ significantly. A C++ object's destructor must be invoked implicitly (in the case of stack-bound variables) or explicitly to deallocate an object. The destructor executes synchronously just before the point in a program at which an object is deallocated. Synchronous, coordinated uninitializing and deallocating 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 deallocated. Very few objects need finalizers. A finalizer is needed by only objects that must guarantee some cleanup of the object state before deallocating, typically releasing resources external to the JVM.
  - With RAII in C++, one 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). (*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.[14][15][16]

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.

Runtime

C++	Java
C++ is compiled directly to machine code which is then executed directly by the central processing unit.	Java is compiled to byte-code which the Java virtual machine (JVM) then interprets at runtime. Actual Java implementations do just-in-time compilation to native machine code. Alternatively, the GNU Compiler for Java can compile directly to machine code.

- 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 Java virtual machine (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 execution. In some cases, especially since Java 7, compiler analysis can prove a bounds check unneeded 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."

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 quite different.

C++ Templates	Java Generics
Classes, functions, aliases[17] and variables[18] can be templated.	Classes and methods can be genericized.
Parameters can be variadic, of any type, integral value, character literal, or a class template.	Parameters can be any reference type, including boxed primitive types (i.e. Integer, Boolean...).
Separate instantiations of the class or function will be generated for each parameter-set when compiled. For class templates, only the member functions that are used will be instantiated.	One version of the class or function is compiled, works for all type parameters (via type-erasure).
Objects of a class template instantiated with different parameters will have different types at run time (i.e., distinct template instantiations are distinct classes).	Type parameters are erased when compiled; objects of a class with different type parameters are the same type at run time. It causes a different constructor. Because of this type erasure, it is not possible to overload methods using different instantiations of the generic class.
Implementation of the class or function template must be visible within a translation unit in order to use it. This usually implies having the definitions in the header files or included in the header file. As of C++11, it is possible to use extern templates to separate compiling of some instantiations.	Signature of the class or function from a compiled class file is sufficient to use it.
Templates can be specialized—a separate implementation could be provided for a particular template parameter.	Generics cannot be specialized.
Template parameters can have default arguments. Pre-C++11, this was allowed only for template classes, not functions.	Generic type parameters cannot have default arguments.
Wildcards unsupported. Instead, return types are often available as nested typedefs. (Also, C++11 added keyword `auto`, which acts as a wildcard for any type that can be determined at compile time.)	Wildcards supported as type parameter.
No direct support for bounding of type parameters, but metaprogramming provides this[19]	Supports bounding of type parameters with "extends" and "super" for upper and lower bounds, respectively; allows enforcement of relationships between type parameters.
Allows instantiation of an object with the type of the parameter type.	Precludes instantiation of an object with the type of the parameter type (except via reflection).
Type parameter of class template can be used for static methods and variables.	Type parameter of generic class cannot be used for static methods and variables.
Static variables unshared between classes and functions of different type parameters.	Static variables shared between instances of classes of different type parameters.
Class and function templates do not enforce type relations for type parameters in their declaration. Use of an incorrect type parameter results in compiling failure, often generating an error message within the template code rather than in the user's code that invokes it. Proper use of templated classes and functions is dependent on proper documentation. Metaprogramming provides these features at the cost added effort. There was a proposition to solve this problem in C++11, so-called Concepts, it is planned for the next standard.	Generic classes and functions can enforce type relationships for type parameters in their declaration. Use of an incorrect type parameter results in a type error within the code that uses it. Operations on parametrized types in generic code are only allowed in ways that can be guaranteed to be safe by the declaration. This results in greater type safety at the cost of flexibility.
Templates are Turing-complete (see template metaprogramming).	Generics are probably not Turing-complete.

Miscellaneous

- Java and C++ use different means to divide code into multiple source files. Java uses a package system that dictates the file name and path for all program definitions. Its compiler imports the executable class files. C++ uses a header file source code inclusion system to share 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++ compiling features an added textual preprocessing phase, while Java does not. Thus some users add a preprocessing phase to their build process for better support of conditional compiling.
- Java's division and modulus operators are well defined to truncate to zero. C++ (pre-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 application binary interface (ABI) dependent within given constraints. Thus a Java program will have consistent behavior across platforms, whereas a C++ program may require adapting for some platforms, but may run faster with more natural integer sizes for the local platform.

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

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:

Java/C++ statement	C++ generated code (x86)	Java generated byte code

`vector[i]++;`	`mov edx,[ebp+4h]`
	`mov eax,[ebp+1Ch]`
	`inc dword ptr [edx+eax*4]`
		`aload_1`
		`iload_2`
		`dup2`
		`iaload`
		`iconst_1`
		`iadd`
		`iastore`

Since performance optimizing 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. Given the very different natures of the languages, definitive qualitative differences are also difficult to draw. In a nutshell, there are inherent inefficiencies and hard limits on optimizing 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. In any case, if the inefficiencies of Java are too great, compiled C or C++ code can be called from Java via the JNI.

Some inefficiencies that are inherent to the Java language include, mainly:

- 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.
- Performance-critical projects like efficient database systems and messaging libraries have had to use internal unofficial APIs like `sun.misc.Unsafe` to gain access to manual resource management and be able to do stack allocation ; effectively manipulating pseudo-pointers.
- 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 use by adding one pointer per object to a virtual table. It also induces a start-up performance penalty, since a JIT compiler must perform added optimizing passes to devirtualize 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.[20]

- 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,[21] as this form of automatic memory management introduces memory overhead.[22]

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 use 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.[23][24]
- Run-time compiling 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.[25] Also, the inverse argument can be made that native compilers can better exploit architecture-specific optimizing and instruction sets than multi-platform JVM distributions.
- Run-time compiling 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.[26] Inlining can allow for further optimisations like loop vectorisation or loop unrolling, resulting in a huge overall performance increase.
- In Java, thread synchronizing is built into the language, so the JIT compiler can potentially, via escape analysis, elide locks,[27] significantly improve the performance of naive multi-threaded code. This method was introduced in Sun JDK 6 update 10 and is named biased locking.[28]

Also, some performance problems occur in C++:

- Allowing pointers to point to any address can make optimizing difficult due to possible interference between pointers that alias each other. However, the introduction of *strict-aliasing* rules largely solves this problem.[29]
- 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 (code bloat). However, because function templates are aggressively inlined, they can sometimes reduce code size, but more importantly allow for more aggressive static analysis and code optimizing by the compiler, more often making them more efficient than non-templated code. In contrast, Java generics are necessarily less efficient than non-genericized code.
- Because in a traditional C++ compiler, dynamic linking is performed after code generating and optimizing in C++, function calls spanning different dynamic modules cannot be inlined. However modern C++ compilers like MSVC and Clang+LLVM offer link-time-code-generation options that allow modules to be compiled to intermediate formats which allows inlining at the final link stage.
- Because thread support is generally provided by libraries in C++, C++ compilers cannot perform thread-related optimizations. However, since multi-threading memory models were introduced in C++11, modern compilers have the needed 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.

Official standard and reference of the language

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.[30]

The C++ language evolves via 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*,[31] a book which is published by Oracle.

The Java language continuously evolves via 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[32]—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 constant source of criticism by stake-holders wanting more stability and conservatism in the addition of new language and library features. In contrast, the 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.

Trademarks

"C++" is not a trademark of any company or organization and is not owned by any individual.[33] "Java" is a trademark of Oracle Corporation.[34]

References

1. https://blogs.oracle.com/darcy/entry/unsigned_api
2. "The Java Tutorials: Passing Information to a Method or a Constructor" (http://download.oracle.com/javase/tutorial/java/javaOO/arguments.html). Oracle. Retrieved 17 February 2013.
3. "The Java Tutorials: Object as a Superclass" (http://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html). Oracle. Retrieved 17 February 2013..
4. "XMPP Software » Libraries" (http://xmpp.org/xmpp-software/libraries/). xmpp.org. Retrieved 13 June 2013.
5. Robert C. Martin (January 1997). "Java vs. C++: A Critical Comparison" (http://www.objectmentor.com/resources/articles/javacpp.pdf) (PDF).
6. "Reference Types and Values" (http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.3). *The Java Language Specification, Third Edition*. Retrieved 9 December 2010.
7. 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*."
8. 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."
9. "Semantics of Floating Point Math in GCC" (https://gcc.gnu.org/wiki/FloatingPointMath). GNU Foundation. Retrieved 20 April 2013.
10. "Microsoft c++ compiler, /fp (Specify Floating-Point Behavior)" (http://msdn.microsoft.com/en-us/library/e7s85ffb.aspx). Microsoft Corporation. Retrieved 19 March 2013.
11. "Java Language Specification 4.3.1: Objects" (http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.3.1). Sun Microsystems. Retrieved 9 December 2010.
12. Standard for Programming Language C++ '11, 5.3.2 Increment and decrement [expr.pre.incr].
13. 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/
14. Satish Chandra Gupta, Rajeev Palanki (16 August 2005). "Java memory leaks – Catch me if you can" (https://web.archive.org/web/20120722095536/http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/). IBM DeveloperWorks. Archived from the original (http://www-128.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/) on 2012-07-22. Retrieved 2015-04-02.

15. How to Fix Memory Leaks in Java (https://web.archive.org/web/20140205030750/http://www.openlogic.com/wazi/bid/188158) by Veljko Krunic (Mar 10, 2009)
16. Creating a memory leak with Java (https://stackoverflow.com/questions/6470651/creating-a-memory-leak-with-java) on stackoverflow.com
17. http://en.cppreference.com/w/cpp/language/type_alias
18. http://en.cppreference.com/w/cpp/language/variable_template
19. Boost type traits library (http://www.boost.org/libs/type_traits/doc/html/boost_typetraits/reference.html)
20. Clark, Nathan; Amir Hormati; Sami Yehia; Scott Mahlke (2007). "Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping". *HPCA'07*: 216–227.
21. Hundt, Robert (2011-04-27). "Loop Recognition in C++/Java/Go/Scala" (https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf) (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*"
22. Matthew Hertz, Emery D. Berger (2005). "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management" (http://people.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf) (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.*"
23. 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.
24. "Boost Pool library" (http://www.boost.org/doc/libs/release/libs/pool/). Boost. Retrieved 19 April 2013.
25. Targeting IA-32 Architecture Processors for Run-time Performance Checking (http://www.slac.stanford.edu/comp/unix/.../icc/.../optaps_dsp_qax.htm)
26. Fixing The Inlining "Problem" by Dr. Cliff Click |Azul Systems: Blogs (http://www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem)
27. Oracle Technology Network for Java Developers (http://java.sun.com/performance/reference/whitepapers/6_performance.html#2.1.2)
28. Oracle Technology Network for Java Developers (http://java.sun.com/performance/reference/whitepapers/6_performance.html#2.1.1)
29. Understanding Strict Aliasing - CellPerformance (http://cellperformance.beyond3d.com/articles/2006/06/understanding-strict-aliasing.html)
30. "Working Draft, Standard for Programming Language C++" (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf) (PDF).
31. The Java Language Specification (http://java.sun.com/docs/books/jls/)
32. The Java Community Process(SM) Program - Participation - JCP Members (http://www.jcp.org/en/participation/members)
33. Bjarne Stroustrup's FAQ: Do you own C++? (http://www.stroustrup.com/bs_faq.html#revenues)
34. ZDNet: Oracle buys Sun; Now owns Java (http://blogs.zdnet.com/BTL/?p=16598).

External links

- Object Oriented Memory Management: Java vs. C++ (http://ineed.coffee/98/object-oriented-memory-management)
- Chapter 2:How Java Differs from C (https://web.archive.org/web/20020402093228/http://www.ora.com/catalog/javanut/excerpt/index.html), chapter from Java in a Nutshell by David Flanagan
- Java vs. C++ resource management comparison (http://www.fatalmind.com/papers/java_vs_cplusplus/resource.html) - Comprehensive paper with examples
- Java vs C performance... again... (http://www.azulsystems.com/blog/cliff-click/2009-09-06-java-vs-c-performanceagain) - In-depth discussion of performance differences between Java and C/C++
- Hyperpoly - Java and C++ Comparison (http://christianrubiales.com/hyperpoly/?q=java,cpp)