Journal of Universal Computer Science, vol. 4, no. 1 (1998), 11-15
submitted: 25/9/97, accepted: 1/11/97, appeared: 28/1/98 © Springer Pub. Co.

Java for Scientific Computing, Pros and Cons

Jürgen Wolff v. Gudenberg
(Institut für Informatik, Universität Würzburg
wolff@informatik.uni-wuerzburg.de)

Abstract: In this article we briefly discuss the advantages and disadvantages of the language Java for scientific computing. We concentrate on Java's type system, investigate its support for hierarchical and generic programming and then discuss the features of its floating-point arithmetic. Having found the weak points of the language we propose workarounds using Java itself as long as possible.

1 Type System

Java distinguishes between primitive and reference types. Whereas this distinction seems to be very natural and helpful – so the primitives which comprise all standard numerical data types have the usual value semantics and expression concept, and the reference semantics of the others allows to avoid pointers at all– it also causes some problems. For the reference types, i.e. arrays, classes and interfaces no operators are available or may be defined and expressions can only be built by method calls. Several variables may simultaneously denote the same object. This is certainly strange in a numerical setting, but not to avoid, since classes have to be used to introduce higher data types.

On the other hand, the simple hierarchy of classes with the root `Object` clearly belongs to the advantages of the language. It is also used to introduce further concepts like exception handling, which thus is fully integrated in the object oriented flavor of the language and as extendeble as all other classes.

Multidimensional arrays in Java have to be handled as arrays of arrays. This means that there is no guarantee that a matrix is allocated in contiguous space. Certainly most numerical algorithms will suffer from that fact.

In a polymorphic environment it often is helpful to get information about the current specific type of an object at runtime. This can be achieved by the method `getClass` which returns the type as a constant object of the class `Class`. In that class methods to get more information about the interface or name of the type are provided.

1.1 Workaround: Operator Overloading

Especially the lack of operator overloading is annoying. It can be overcome by a precompiler which maps the operator to method calls. Since methods may return arbitrary classes this precompiler only has to change the notation of each expression locally. So the correspondence between the original source and the generated Java program is quite obvious. Using such a precompiler we can also allow for the definition of user defined operator names or symbols including the determination of its precedence.

2 Hierarchical Programming

The paradigm of hierarchical programming means the classification of the data using inheritance. Base class composition or programming abstract data types are alternate names. We recommend to separate between interfaces and implementations of classes as well as to collect common behavior in general abstract superclasses which then can be extended for the specific application. This design leads to reusable, easily extendable classes.

The separation of interfaces and classes and the use of packages in Java provide a good support for clearly structured programs.

Since there is only one hierarchy of classes, we always can cast or convert an object of a type to its supertype. This is the usual "is-a" relation in object oriented inheritance structures. A conversion in the other direction from supertype to subtype is also possible. Hence, if we know the specific type of an object, we can make use of its particular features. These downcasts are obtained by writing the name of the destination class in front of the object. In a similar way upward and downward casting of primitive types is possible, where we now have the following subtype relation.

`int` $\subset$ `long` $\subset$ `float` $\subset$ `double`

Note that this relation only indicates the upward direction for casts and does not mean that there is no information lost, when you convert an `int` or `long` to a `float`.

Methods are bound dynamically by default. The modifier `final`, however, which can be attached either to a method meaning that this method can not be redefined, or to a whole class which then prohibits the inheritance from this class, opens possibilities for optimization.

3 Generic Programming

The paradigm of generic programming in general deals with homogeneous containers, i.e. data structures composed of elements of the same type. These containers define one or several iterators to access all or some of their elements in a specified order. These iterators then link generally defined algorithms to the container. In the package `java.util` a simple iterator interface is provided which can easily be extended to the user's needs.

This style of programming yields a high degree of reuse, since the same structure is preserved. The data structures generally are parameterised by the element types. The algorithms are further generic with respect to the iterators. This kind of parametrization is usually provided by templates, a feature which unfortunatedly is lacking in Java.

3.1 Workaround: Templates

In the language manuals it is suggested to use containers of the type `Object` instead and perform appropriate type casts. This does not work well, since primitive types are no objects, so we can not fill a container with primitive types. To cope with this situation wrapper classes are included in the language, which simply wrap a constant value of the corresponding primitive type. Since they are objects, no operations are available for wrapper classes, not even an assignment of a new value is possible. Hence, if we want to write generic code for primitive data, we have to explicitly check the current type and cast. This kind of code – a large conditional satement depending on type of the current object– has not much to do with object orientation, even if it is encapsulated in a helper class, like `NumberHelper` [8]. This class provides methods for arithmetic operations for the wrapper classes which use the predefined primitive operations. So this approach only helps for the limited number of wrapper classes.

We can provide a similar helper class for higher arithmetic data types. The main disadvantage of this approach is that it is not extendable. If we add a new type with the same interface, we have to edit the helper class and to add a new branch in the conditional statement.

We might try to overcome the difficulty by using the runtime information to get the name of the class of the elements and then cast to it. This, however, can not be compiled, since the syntax requires an identifier and not a string as a typename. Therefore, we write a factory class which produces a helper class for each element type. The name of this helper class contains the name of the element class. All helper classes implement one generic interface which now replaces the template. The application program calls the `makeHelper` method of the factory class with an object of element type as parameter. This looks for the corresponding helper class and loads it, if available. If not, it writes the source to a file inserting the object's type name where appropriate, compiles this file on the fly and then loads the corresponding class. Since all helpers have identical structure this process of writing and compiling can be made automaticly.

4 Arithmetic

Java is one of very few languages which completely specify the semantics of their floating-point types. The IEEE 754 formats *single* and *double* have to be used for the `float` and `double` data type, respectively. All 4 basic arithmetic operations are to be performed with full precision using the rounding to the nearest neighbor. Expression evaluation has to use these basic operations to compute each intermediate result, i.e. round after each operation. This rule ignores the facilities which some hardware units supply, like fused multiply and add or extended temporaries, and therefore has lead to some discussion [4]. In our opinion this feature of strict expression evaluation is mandatory for a language which claims to get identical results on all computers. And, indeed, this goal is achieved by that rule. So even programs using floating point operations are completely portable.

An interesting Java extension where expression evaluation may be switched between this strict mode and a so-called natural mode which makes use of extended hardware features and hence produces results which are at least as good, is discussed in [3]. It will be interesting to investigate, if the switching of arbitrary evaluation modes can be made an integral part of a language. Then other modes like the accurate expression evaluation of the C++ toolbox [2] can be incorporated smoothly. But this certainly leads to another language.

Although Java supports some of the IEEE 754 properties there are a lot of flaws in the adaption of the full standard. Two minor ones are the lack of any extended floating point type and the definition of the square root function without accuracy requirements. In our opinion all standard functions should be optimally accurate or at least specify their errors exactly.

The IEEE standard provides 4 rounding modes, but Java only knows one of them. So it is impossible to call the directed roundings and hence define an interval arithmetic in the language.

One strong point of Java is its extendable exception handling. Surprisingly enough the 5 IEEE floating point exceptions are ignored. There is no way to enable traps for these exceptions and specify handlers, or at least to check for the occurrence of the exception. Only if an infinity or NaN is the result, the user has the indication that something has gone wrong.

4.1 Workaround: Arithmetic

We derive all IEEE exceptions from the Java exception `ArithmeticError`.

The native interface of Java allows to write methods in C which return java classes and raise Java exceptions and bind them to Java programs. We exploit these features to define a class `FPU` which has access to the underlying hardware status. It provides native methods, i.e. written in C, for the arithmetic operations for all rounding modes. These methods execute the corresponding operation, read and copy the fpu state register, raise a new Java exception, if the IEEE exception occurred and the corresponding trap is enabled and then clear the fpu state. Class `FPU` further provides native methods to set, clear and test traps or exception flags and native methods to set and get the rounding modes. The utility functions like nextAfter etc. can then be written in Java.

Since this class has to use native methods, the program can only be distributed as an applet, if the external part is locally available, or it has to be ported as a stand-alone program.

The native arithmetic operations are declared as methods of the `FPU` class, i.e. they have 2 parameters. So it is advisable to define higher numeric data types like `IEEE_float` or `interval` on top of these class. We have tried such an implementation and have measured a considerable decrease of efficiency mostly due to the fact that native methods run in a separate thread, for details see [9]. As stated in the beginning operators for these classes can not be declared.

5 Conclusion

We have pointed out some strong and weak points in the definition of the Java language. In the following table we give a concise overview.

Several criteria may be checked to judge whether a language is favorable for scientific computing.

- portability
- efficiency
- clean and reliable floating point arithmetic
- definition of intervals and other arithmetic types
- mathematical notation

If only the first three items are considered to be important, Java is a good choice. The efficiency is surprisingly high and will certainly be improved by new optimizing compilers.

We, however, consider the last two items to be important and hence proposed different workarounds: a precompiler for operator overloading, dynamic generation of templates and native methods to adapt the floating point arithmetic. Since these are relatively comprehensive tools or expensive and inefficient workarounds, we cannot really recommend Java as a language for scientific computation.

topic	pro	con
type system	hierarchy	
	primitive vs reference	objects are no values
		no operators
		no expressions
		no contiguous matrices
	wrapper classes	without semantics
	runtime info (`Class`)	
	exceptions, threads	
hierarchical prog.	packages	
	classes	
	interfaces	
	final	
generic prog.		no templates
	type casts	need explicit type id
	iterator `Enumeration`	
exception handling	available	not floating-point
arithmetic	IEEE formats	no *extended*
	IEEE operations	no srqt
	strict evaluation rules	
		no directed roundings
		exceptions ignored

Java pros and cons (summary)

References

1. Barton,J. und Nackman,L.: *Scientific and Engineering C++*, Addison-Wesley, 1994
2. R. Hammer, M. Hocks, U. Kulisch, D. Ratz: *C++ Toolbox for Verified Computing*, Springer, 1995
3. Coonen, J.:*RealJava*,`www.validgh.com/java/realjava`
4. newsgroup `www.validgh.com/java`
5. *The Java(tm) Language: An Overview*, `java.sun.com/docs/overviews/java/java-overview-1.html`
6. Arnold,K., Gosling,J.:*The Java(tm) Programming Language*, Addison-Wesley, 1996.
7. Gosling,J., Joy,B., Steele,G.:*The Java(tm) Language Specification*, Addison-Wesley, 1996.
8. The Java Generic Library, `www.objectspace.com`
9. Lerch, M. and Wolff von Gudenberg, J.: *Java for Scientific Computing*, Technical Report, Institut für Informatik, Universität Würzburg, to appear