Object-orientation

I need more references

nice chapter on overloading encapsulation: http://www.cs.rpi.edu/ szymansk/OOF90/Forum.html

important point on Compositional Software Construction (locate and reuse code): http://g.oswego.edu/dl/html/javaInCS.html

EEE? expressiveness first, efficiency later?

ABSTRACT

I advocate that scientific programming should be done in an object-oriented language, that Java offers researchers an easy-to-use, and very expressive language. Java's World Wide Web capabilities and its portability improve a researcher's ability to maintain his compute environment, collaborate with other researchers, and publish his implementations along with his scientific description. Subroutines in more efficient languages, such as Fortran and C, help to overcome local performance bottlenecks in Java programs.

INTRODUCTION

Computers perform complex tasks by executing many simple instructions. Humans specify these instruction by writing programs in a programming language. Programming languages combine individual machine instructions into more powerful compound instructions. They also allow new compound instructions to be created, named, and used as a unit. Computations expressed in these units communicate the program in a more abstract way, ignoring the detail of machine instructions and highlighteing the essential concepts of the computer solution.

I suggest to distinguish computer languages by three criteria: Expressiveness, ease-of-use, and efficiency. **Expressiveness.** Computer languages do not differentiate as much in what solutions can be implemented in a computer language, but in the ease with which a programmer can express and implement these solutions. Floyd (**?**) suggests that a programmer selects the computer language that expresses the needed programming paradigms [1]

Ease-of-use. I like simple languages, because it is hard enough to program well. Some languages are extremely expressive, but very hard to learn and to use. Computer scientists often mention Algol68, Pl/I, and ADA as languages that are overly complex (Wirth, Hoare). Especially, programmers in application fields have to avoid the Turing tar pit: a language that can express almost anything but cannot express anything conveniently.

Efficiency. Finally, computer languages execute with variable efficiency. I believe,

[1] what are our paradigms? Structured programming, object-oriented programming, arrays and indexing, addressing of subvectors, looping, recursion, physical vector spaces, IO to disk, World-Wide-Web.

that fundamentally, the languages do not differ in execution speed, but the abstraction a programer chooses come at a cost in performance. For example, Fortran90 can contain traditional Fortran77 code that will execute with the usual speed. However, Fortran90's more object-oriented features cost execution speed. The same is true for for C++ and C. Additionally, compilers for new languages will require some time to evolve to the efficiency of its mature competitors. So does application code written in such a new language. For example, A recent comparison of Fortran90 and C++ implementations of computational kernels at Lawrence Livermore Laboratory showed little difference in performance cite(http://jove.colorado.edu/ cary/CompCPP_F90SciOOP.html). Mature compiler and application optimization resulted in comparable performance. In summary, choosing a programming language requires a trade-off between expressiveness, ease-of-use, and efficiency. In general, a high-level, abstract language is more expressive but less efficient. Ease-of-use seems to be an art and requires most of all consistency.

I want Particle example, and deconvolution example.

I believe none of the current programming languages will prevail forever, but I believe the principle of object-orientation will be with us for a long time. Object-oriented languages allow a programmer to combine data *and* functions to an abstract compound unit, a class. For example, a particular program may require that a two dimensional array `x(n0,n1)` is scaled by a constant scalar `a`. Fortran77, C, C++, and Java allow the programmer to loop over the two dimensional array `x` using the array parameters `n0,n1`. However, the languages vary in the way, the programmer is able to package the data and operation into compound units. The Fortran77 programmer can write a subroutine `arrayscale(x,n1,n2,a)` that changes the array values. The C programmer may combine the two-dimensional array into a single structure `xx`. The C invocation may read then `arrayscale(xx, a)`. Java (or C++) can combine the two-dimensional structure and its scaling operations into a single object `xxx`. The invocation of the array scaling method `xxx.scale(a)` scales the object's internal array values. Another method may compute the norm of the array elements or write the elements to a file. Such an object-oriented compounding of variables and functions leads to programming concepts that do not exist in traditional, procedual languages.

WHY OBJECT ORIENTATION?

Object orientation hinges on three fundamental concepts: encapsulation, inheritance, and abstraction.

Encapsulation (data-hiding). A single program unit, e.g. the array object above, unifies the data that represents the object's state (member variables) and the functions (member methods) that access or modify that state. To build such a program unit the programmer creates a class, a type description of the object not unlike a structure definition in C. An object is instantiated from a class by specifying its state. All objects of the same class have the same methods but different states. If classes compare to C structures, than a class's objects compare to variables of that structure type.

Inheritance (sharing common code). Classes are organized in hierarchies to share common member variables or methods. The class at the bottom inherits all members of the classes above and usually adds its own enhancements and specializations. Inheritance avoids unnecessary source code duplication, a vice any programmer avoids. C++ and Java allow for classes that do not implement any of its members but only declare their name, arguments, and return types. Java calls such classes interfaces. Interfaces are important to set in stone the method names and arguments, so that programs can be written for any implementation of a given interface.

better a remark?

should this be polymorphism?

Abstraction (or polymorphism) is the idea to address objects of different classes as a common class type. For example, an array of floats and an array of complex numbers are two distinct classes. However, if both classes share a common array class, than an application can use the common array methods to treat either array independent of its specific type. Such a single application can then process float arrays, complex-number arrays, or any other object of the array type. Usually, the common class is a class higher up in the class hierarchy. **Expressiveness. Ease-of-use. Efficiency.** run time method resolution (dynamic method resolution). strongly typed (requires casts). primitive types versus object types (simple things simple).

better example: scientific. want to use array of common stuff.

Object orientation. Encapsulating the data and its corresponding methods in a single programming unit, the class and its objects, allows us to isolate these programming units within objects of our application. For example, in the numerical optimization library Jest (**?**) we decided to isolate objects by expertise: numerical analysts are likely to write the solver classes while researchers in the natural science can concentrate on their particular operator. Vector classes are probably shared among researchers in the same community. Inherited method invocations for each class enforce that all objects collaborate with each other. Such an organisation follows the classic criteria of Parnas (**?**) of Carnegie-Mellon University to decompose a software system into modules so that any later change is likely to occur within a single module. Claerbout (**?**) shows a library of Fortran routines that fails to separate the physical science (operators) and the numerical analysis (solvers).

Malleability. Programs evolve and evolution is easiest if the changes are local. Good design will enclose all pertinent information within a single class. Access to the objects data is restricted to a few object methods. A change to the internal data does not effect any code outside classes as long as the class method's remain the same in functionality and invocation. The continuity of the member functions is often enforced by the object's inherited (possibly abstract) member functions. For example, an array class may change its internal storage of its elements from a fixed size array to a dynamic array with variable size. Applications that use the array class do not have to change if the array class methods remain the same.

Extensibility. Inheritance allows a programmer to incrementally add functionality by deriving new classes from the existing class hierarchy. These new classes are compatible with the existing class hierarchy since they inherit predefined member methods and implement more abstract parent classes.

later when discussing other languages?

These benefits usually rely on two or all three concepts of object orientation: encapsulation, inheritance, and abstraction. A pseudo-object-oriented language, such as Fortran90, that misses one of these concepts severly limits its usefulness.

other criteria of comparison: no natural language has the consistent preciseness of programming languages.

In summary, On the scale set by Fortran and C, object-oriented languages such as C++ and Java stand far ahead in the level of discourse they allow.

```
say something about:
run time method resolution (dynamic method resolution).
strongly typed (requires casts).
primitive types versus object types (simple things simple).
```

Notes and Comments

Axelrod organized a fascinating programmer competition as the basis of his book (**?**). Every participant had to submit a program that would face off withanother program in a prisoner-dilemma situation. Axelroth carried out a tournament between player strategies submitted in form of a program by fellow game theoreticians. Such a tournament is easily organized in a object oriented language: As an exercise, design an abstract player class. Make sure every player shares the same accounting methods. Implement two players for that abstract player class. Axelroth limited his competition to duels. Assume you interact within a community of up-to 100 individuals. You have to add a `getName()` to the abstract player class.

Procedual immitation. Object-oriented languages can mimic procedual languages by offering classes with a single member function (subroutine) or a single member variable (data structure). It is difficult to immitate object-oriented languages in a procedual language even if you can combine data variables and functions into single structures. The structure would still miss encapsulation, abstraction, and inheritance.

Inheritance. Programmers of object-oriented languages distinguish between an “is a” and a “has a” relationship: only if an object is a specialization of the other object, should a programmer use inheritance. For a “has a” relationship a programmer should include a variable member of the related “has a” class. If in doubt, a “has a” relationship is preferable in non-abstract classes.

WHY JAVA?

Among all object-oriented programming languages, I advocate Java as a replacement for Fortran in engineering and scientific research. Only simple, compute-intensive parts of a software should be implemented in Fortran 77.

Ease-of-Use

Java is easy to learn and fun to use. In contrast to C++, Java does not use pointers and includes automatic garbage collection. Java does not permit multiple inheritance and operator overloading. Currently, Java does not support a template mechanism; however, a a keyword *generic* is reserved for a possible future implementation.

Java wraps a general programming language, a graphics-, and networking toolkit into a single consistent framework. Java comes with a large well-designed and pragmatic object library (e.g. hashtables, IO streams, and GUI buttons). Java has no C-like header files but its own package inclusion scheme that makes Java programs available on the world wide web. Furthermore, Java's naming scheme is intuitive and consistent. Java's decision to derive every class from a single root class, *Java.Object*, gives the package a unified concept. However, Java includes wisely a few primitive data types, such as floats or integers, for which an object implementation is often inefficient and superfluous.

Determenism. Java was built to be reliable. The creators of Java specified a virtual machine and ensured that Java programms change the state of that machine entirely deterministically. Consequently, Java includes strong typing, array bound checking, and other language controls. The thorough language specifications and the determinism of its programs, enables the compiler to detect more errors at compilation time, and the run time system to deliver a precise error stack of any runtime error.

Tools. Java's documentation tool Javadoc produces standardized HTML pages from comments in any Java source code: an important step towards Knuth's literate programming paradigm. Java has its own up-to-date mechanism (checksource flag of the Java compiler) for source files. needs citation for Knuth's web

Availability We program on personal computers, workstations, and supercomputers. Quickly after Java's introduction, Java compiler were available on all platforms that we are using. For many platforms we can now choose between vendor compilers and third party compilers. Furthermore, most popular Internet browsers (Netscape, Internet Explorer) include a Java compiler. Finally, Java's integration into all popular Web browsers helps a programmer to publish and exchange his software.

Overall, the software industry and especially the personal computer vendors seem to invest into Java technology. Unfortunate for the scientific community, Java's development is driven mostly by non-scientific applications. It seems unavoidable that many future computer technology advances will be driven by the growing consumer market of personal computing, rather than the proportionally shrinking scientific computing. Consequently, the scientific and engineering community should take advantage of mass-market technologies, avoid expensive specialized tools, and attempt to influence the available tools to avoid the most flagrant short comings. As Professor Kahan pointed out with a $1000 price tag per Fortran line only the federal government can afford it. (Kahan: ACM invited talk). Java critics within the scientific community (CIP article) have pointed out that the sales of Java developer environments make only a small fraction of the total market. I argue that among the mass-market programming languages, Java is the most attractive one for scientists. In other words, Java probably outsells Fortran90 developer environments. It sure outranks Fortran shelf space in technical bookstores.

Portability. We move programs from machine to machine. As time passes computers are replaced. As we move around from job to job our hardware changes. Over time new compilers and developer environments become available. Nothing is more frustrating than *fixing* software that *ought* to work.

Java was designed to be portable. Since Java is defined for a virtual machine, only this virtual machine specifications (its instruction set) has to be ported. Once the virtual machine is available, Java will execute. To make Java safe and verifiable, Java is specified so that its possible states are entirely deterministic and therefore the code can be **verified** to be harmless to the client computer. SUN attempts to protect Java's purity by testing any Java's compiler's ability to execute standard Java before. We have ported the Jest library to two platforms and two browsers without a single portability problem. Furthermore, we changed between minor and major – 1.0 to 1.1 – compiler versions without any complications. We know of three Jest installations and nobody has reported Java portability problems. However, Jest relies mostly on parts of Java that are traditional language features, such as arithmetic operations, string handling, stream IO, etc. We heard about portability problems with more innovative parts of the language, such as the window toolkit AWT.

make this clearer contrast: standardization or help to evolve. Ultimate goal: making yesterday's software available today.

A friend, Bill Harlan, pointed out how a language is finally standardized just as it becomes obsolete: C, C++, Java. In the mythical man month, Brooks (**?**) suggests that a project the moment it is finished it is aging. In contrast, Java attempts a more evolutionary approach. SUN reserves the right to add new classes and to remove obsolete ones. During a transition period, Java compilers support new as well as deprecated language features. However, when encountering a deprecated feature, the compiler warns the programmer. Such an evolutionary approach to software maintenance encourages incremental changes to existing programs and may proof superior to the traditional ANSI standardization process that attempts to fix a language forever but usually requires catastrophical changes after a decade of stagnation.

Correctness. We enjoy writing a program, but we hate to debug it. Java is a language that catches many errors during compilation time. Additionally, subtle language features such as object libraries package support, its naming conventions, its documentation tool Javadoc, prevent many errors from being programmed in the first place. Dijkstra (**?**), the man who slayed the goto statement, points out that *those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper.*

Libraries. We want to build upon the time-tested programs of others when we can: Java comes with a large set of packages that include a graphics toolkit, networking programming, threads, and a database interface. Rapidly, additional expert classes are being developed and become available. Because of Java's portability these classes are quickly downloaded and tested. Additionally, Java's simplicity, naming conventions, and documentation gives all packages a consistent programmer interface.

Furthermore, Java provides a standard mechanism to invoke C subroutines. Consequenly, we may not need not abandon our time tested C- and Fortran routines after all.

A potential reason to shun Java is its current rapid growth and change. Java is not standardized and past releases changed sufficiently to require considerable reprogramming of already existing applications. In particular, Java's window toolkit, AWT, has been undergoing drastic changes. However, Java's core libraries (list here ???) have been stable.

closer to portability?

Expressiveness

We work on many different types of problems. Especially interdisciplinary work involves combinations of several data sets with very different representations and formats. We need a computer language that can express a wide range of different concepts and solutions. Java includes a general purpose programming language, a rich set of flexible libraries, and, for example, a window toolkit. Java offers solutions to problems that would have otherwise required at least two or more computer languages (C++ and X-Windows for example). A data object, such as an array of experimental values might include a method to plot the data on a given media, such as a graphics window or on printer.

Publishing. Java code is very readable and can be included in publications. In general, Java more readable than C and C++ since it lacks pointers. At the arithmetic level, Java is as readable as Fortran, except when dealing with complex numbers. Because of its ability to encapsulate details in its objects, Java is, however, far more readable than any procedural languages at the more abstract level of solvers and main routines. Java's Javadoc helps documenting software: Javadoc generates online documentation directly from comments in a class' source code. None of the other languages includes such a tool and consequently their documentation varies from software to software. We plan to integrate Java applications directly into the web versions of our research documents, making our research easily available to more readers. Java is built for the web, and our laboratories unique reproducible electronic documents – a packaging of research description, figures, and software – will soon become available on any internet browser.

Efficiency

Java is less efficient than Fortran, C, C++ and probably half-a-dozen other languages. When interpreted, Standard Java is about 30-60 times slower than C. When Java is compiled by a just-in-time compiler (jit), it seems to be about 3-6 times slower than C. Java compiler technology is improving rapidly, but how efficient it can get relative to other languages is unclear. Hank Schiffman (**?**) believes that Java's native code will be about as fast as C++ in most cases, but hardly ever faster. For the time being, we find Java's just-in-time performance acceptable for prototyping small- to medium-sized problems. However, for large field data examples, we may have to re-implement our application in a better performing language, after we developed it in Java.

Java's performance is improving. New compiler technologies, such as just-in-time compilers, are reducing the performance gap rapidly. System programmers, people who write compilers, improve their Java implementations. Hardware vendors are designing computer chips with Java optimization in mind. Additionally, a group of enthusiastic Java programmers develop tools for high-performance network computing (cite conference 98).

Java offers a standard linking mechanism for native C routines. The mechanism allows a programmer to take advantage of proven, existing C code when ever performance or convenience justifies it. Of course, native subroutines do not have Java's portability, however, a simple mechanism can replace a native subroutine by a Java version if the native subroutine

	Ease of Use	Expressiveness	Efficiency
Java	+	+	
C++	-	+	
Fortran	+	-	+
Matlab	+	-	-

Table 1: Comparison of common scientific programming languages. Ease-of-use includes the learning curve, the library support, and the portability and availability of a language. Expressiveness is a language's capability to express complex entities such as Jest's composite operators. Efficiency is a language's comparative execution speed.

is not available in a certain runtime environment.

But efficiency is not the only criteria to choose a language, or we would all write assembler code. Instead we need to balance the various criteria of a language and ask ourselves: is Java sufficiently efficient?

What about ...

In the past years we encountered other programming languages, such as C, C++, Fortran, but none offered the combination of dramatic new capabilities and practical usability available in Java. Some of these languages we used for years or months (Basic, Fortran, C, C++, Pascal, Mathematica, Matlab) others we read about and may have considered learning (Eiffel, Ada). How do these languages compare to Java and why do we prefer Java?

Fortran, C, and Pascale are all practical but their approach to programming is to concrete. The structural improvements of C and the type checking improvements of Pascal are not sufficient to dislodge Fortran as the most useful scientific programming language. Fortran90 has a limited object concept but lacks inheritance and the abstraction of polymorphism (**?**). Ada provides limited form of objects but again no inheritance nor polymorphism. Ada does support concurrency and synchronization. Smalltalk is inefficient. We read that Eiffel is a fine alternative to Java or C++. Unfortunately, Eifel has not achieved Java's or C++'s availability and popularity. In comparison to Java, C++ is difficult to learn and to use. Mathematica, Matlab do not enable a the user to define problem specific data structures (object orientation) which limits their usefulness for large, complex software projects. None of these languages except Java integrates the World-Wide-Web as a computing environment. Is that correct?

Fortran

> Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. *Brain W. Kernighan (Author of C)*

> I don't know what the language of the year 2000 will look like, but it will be called Fortran. *C.A.R. Hoare*

Fortran77 is a serious alternative to Java (or C++). Not because it challenges Java's abilities directly, but by emphasizing a different set of goals. The two languages can hardly be compared. Fortran77 is easy to learn and to use. Classic Fortran77 is limited to a small set of language features. Fortran77 was once universaly available. I believe that Fortran77 limits a programmers thinking to the few paradigms Fortran supports well: arrays and do loops.

Fortran90 adds object-like structures, infix and array section syntax, and dynamic memory allocation for variables. But Fortran90's lack of inheritance and polymorphism limit its expressiveness. Furthermore, Fortran90's object-oriented features slows down a Fortran code's execution speed, the major merit of the language. However, because of Fortran90's ability to execute efficient Fortran 77 code, it offers the programmer a direct trade-off between execution speed and expressiveness. Non-scientific applications, such as IO streams, data formatting, or graphics are difficult to program in Fortran. Consequently, few programmers can live on Fortran alone. Fortran90 compiler, however, were slow to arrive on all our platforms. Linux and GNU still do not offer a Fortran90 compiler. I believe the number of Fortran programmer's is declining. Only few places outside of sciences and applied engineering use Fortran. Most computer science departments do not teach Fortran to their students. Students in the sciences should consider a computer language that offers more career opportunities after graduation than Fortran.

`Fortran90: Michael Metcalf and John Reid: Fortran90 Explained, Oxford University Press, New York, 1990.`

Nevertheless, Fortran code can be fast. Consequently, I advocate to use Fortran the way Fortran programmers used assembler language many years ago: only where absolutely necessary.

The specifications of Fortran2000 would make Fortran into a true object-oriented language. However, these abstract language features will come at the same performance decrease that other object-oriented languages experience.

C++

C.A.R. Hoare said that he did not know what the language of the year 2000 would look like but that it would be called Fortran. We know today that Hoare was wrong: Today, C replaces Fortran in both universality and uniformity. More young engineers and scientists learn C than any other language. UNIX, the operating system of most scientific computers, is implemented in C. C overcame the memory-allocation limitation of Fortran77, and offered new features such as data structures and scope.

C was standardized, and new developments have centered on the C++ language. C++ extends C but adds object oriented features. C++ has grown rapidly. It already has many more users than Fortran90 (**?**). We find C++ compilers on all manner of computers (also the free GNU compiler). However, C and C++ lack the performance of Fortran77 even they are often comparable to Fortran90 (**?**).

However, the C++ language is difficult to learn and error-prone because of the lack of automatic garbage collection and the addressing of objects through pointers. Multiple inheritance is possible in theory but is best avoided according to experts. I feel about C++ as Dijkstra (**?**) feels about Algol60: *it allows the programmer so much combinatorial freedom that its confident use requires a strong discipline from the programmer. Besides being expensive to implement it seems dangerous to use.*

Garbage collection: C++ burdens the programmer with explicite memory management. If a programmer does not free an object's memory after it has outlived its usefulness, the memory will not be available to the program until the program exits. Since in an object-oriented language the allocation of resources and the dependencies between objects can be hidden, garbage collection by the programmer is prone to error and potentially wasteful. In contrast, Java automatically frees memory allocations that the program does not address anymore. However, if desired, a Java programmer can explicitely disallocate memory (and other resources).

Notes and comments

Gould. Interesting biological analogy: Gould catastrophic, standard smooth.

Gossling wrote a preliminary discussion about an extension of Java for numerical applications (**?**).

Complex number array. To avoid the overhead incurred by a complex number objects, I implemented a complex number array class with a rich set of methods. Internally, the class efficiently represents a complex number by two float arrays. The trick is to replace operations on individual complex elements by array operations.

Only after a call to SGI's compiler experts, Martin Karrenbach (**?**) managed to link Fortran90 with C++ framework on our SGI Power Challenge.

Infix. I do not advocate to add array sections (that `v[3:9:2]` stands for `v[3], v[5], v[7], v[9]`) or infix operators (that if `x` and `y` are arrays that `x*y` computes the vector product). I feel that these operations are difficult enough to warrant the more explicit syntax. See `http://www.vni.com/products/wpd/jnl/JNL/docs/intro.html` for an supportive view of array sections and infix operators.

In Java static class methods are similar to traditional subroutines.

C++ compilers accept C programs and consequently offer an incremental learning of C++ for C programmers. Java does not.

can be avoided in new framework

Strongly typed. Java is strongly typed. Unfortunately, Jest's abstract base class concept requires a considerable amount of explicit type casts which inevitably clutter the Jest code. The casts change the apparent type of an object. For example, it may be necessary to *downcast* an object to a specific derived object so that the programmer can take advantage of any additional methods or members that the specific class carries.

Can Eiffel overcome this?

APPENDIX A: EXPERT WITNESSES ON OPTIMIZATION

Rules of Optimization:
Rule 1: Don't do it.
Rule 2 (for experts only): Don't do it yet.
M.A. Jackson

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity.
W.A. Wulf

We should forget about small efficiencies, say about 97premature optimization is the root of all evil. *Donald Knuth*

The best is the enemy of the good. *Voltaire*

APPENDIX A: JAVA ENHANCEMENTS

Good design makes simple features simple to use. As collatory difficult, infrequent, or unimportant capabilities can tolerate a certain complexity in their use. For numerical applications, Java violates this fundamental design rule in the two cases: Java does not facilitate the simplicity the indexing operations and complex numbers warrant. Java was not designed for numerical analysis; neither were C nor C++.

Indexing operator

I propose that Java enables programmers to overload of the indexing operator `[]`. The indexing operator is currently restricted to Java arrays. For example, elements of a Java array of floats are addressed straighforwardly:

```
float x[] = new float[10];  // using native Java arrays
float y[] = new float[10];
...
y[i] =  (-x[i+1] + 2*x[i] - x[i-1]) / 4.0;
```

The code is very readable and concise. Consequently, it is easy to maintain. In contrast, any array-like object that does not use Java arrays must provide functions to address its array elements. Such verbose function calls are not warranted by the index operator's simplicity:

```
Myarray x = new Myarray(10);     // using Java's Vector class
Myarray y = new Myarray(10);
...
y.setElement(i,(- x.ElementAt(i+1)
+ x.ElementAt(i  )
               - x.ElementAt(i-1)) / 4.0);
```

Why not always using the built-in Java array? Obviously, there is a need for additional array-like classes. JGL and the Java built-in vector class offer arrays that can be dynamically allocated. In my own research, I would like to write an out-of-core array class.

Arrays are fundamental to many applications, not only in science and engineering. By limiting the index operator to the Java built-in array, the language designer leaves the programmer with a no-win choice: either the programmer accepts verbose functional calls for simple element addressing or the programmer gives up the freedom to represent his array-like data structures in any way but as a Java built-in array.

Primitive complex number type:

I advocate to include a complex number type in the Java API. Complex numbers are important in many applications in sciences, engineering, finance, and graphics. Their representation is not complex enough to warrant the overhead a class implementation incurs. Without a standard implementation, additional wrapper classes are needed to make varying implementations compatible. Additionally, since Java does not support operator overloading, such a complex number class cannot offer the concise mathematical expressions that we expect. For example, the following code would compute the root of a quadratic equation, if Java had a complex primitive type[2]:

Replace example with your non-static complex class

```
static complex root(complex a, complex b, complex c) {
  return (-b + complex.sqrt(b*b-4*a*c))/(2*a);
}
```

Instead Java currently requires:

```
static Complex root(Complex a, Complex b, Complex c) {
  Complex b2      = Complex.multiply(b,b);
  Complex ac4     = Complex.multiply(4,Complex.multiply(a,c));
  Complex discrim = Complex.sqrt(Complex.subtract(b2,ac4));
  return Complex.divide(Complex.subtract(discrim,b)),Complex.multiply(2,a)));
}
```

A complex number type would have to be accompanied by the standard unary and binary operators. The operators should allow for mixed operands such as float and complex. Operators without a complex extension, such as increment and modulus, would, of course, not exist. Furthermore, the class should be accompanied by a set of static class methods corresponding to the java.lang.Math package for the double type.

[2]A non-static Complex class may be able to avoid some of the clutter listed here. I want to thank the authors of http://math.nist.gov/jnt/problems.html for the example.

APPENDIX C: DECONVOLUTION IN RATFOR, FORTRAN90, C++, AND JAVA

APPENDIX D: PARTICLES IN RATFOR, FORTRAN90, C++, AND JAVA

REFERENCES