This article was published in 2000, in *Gamelan*

Scientific Computing in Java (Part 2): Writing Scientific Programs in Java

By Ken Ritley

We live in a technological world, at the heart of which are scientists and engineers. They need programming tools to help make important discoveries and bring the next generation of technology to market.

Part 1 of this article discussed how scientists can benefit from Java. We've said that despite a few pitfalls which scientists should watch out for, the future of Java as a scientific programming language looks bright. Here in Part II we examine the structure of a scientific program more closely. We'll define a few scientific OOP design patterns in Java, and we'll give you a short style guide that can help scientists write good Java programs (please see the sidebar "A Style Guide for Scientific Programs in Java").

A Style Guide for Scientific Programs in Java

The Golden Rule of Programming states that computer programs should be understandable by the people who write and maintain them. But what's good programming practice for business is not necessarily what's good for science.

Here are some ideas scientists can use for making Java programs more readable and easier to maintain.

Translating Formulas to Source Code

1. Use short variable names like e, m and c, to mirror the variables which appear in the original equations.

Clearly `e = m*c*c;` is much easier to compare with the original equation and debug than `energy=mass*speedOfLight*speedOfLight;`.

2. Hungarian notation in Java can be helpful.

It's not necessary to precede every variable with a letter which indicates its type: d=double, i=integer, etc. But because Java convention is to start variable names with lowercase letters, sometimes Hungarian notation can be more aesthetically pleasing, especially for loop control variables; for example `iNumberOfElectrons vs. numberOfElectrons`. It also helps make clear when a seemingly integer-like variable has been converted to a different type, as in `double dNumberOfElectrons = (double) iNumberOfElectrons;`.

3. Use descriptive variable names for controlling loops and all other variables.

Remember when looking through source code, loops are what the eye sees first. It's important that the physical meaning of the loop be clear. For example,

```
// Eq. (5) Loop over all electrons
for (i = 0; i < iNumberOfElectrons; i++)
E[i] = m[i]*c*c;
```

4. When coding an equation which appears in a journal article, always cite both the article and the equation number.

For example, at the beginning of the method you may type `// Einstein, Ann. Phys. 17, 891-921 (1905)` Later on in the code, it only takes a few extra seconds to type `// Eq. (5).` Frequently overlooked, this simple rule will save enormous time and grief, both for you and for someone trying to understand your program.

5. Break up complicated equations to enhance their readability, but make your intermediate, temporary variable names clear.

$$v' = \sqrt{\frac{1 - v^2/c^2}{1 - v/c}}$$

The variable names numer and denom make it instantly clear that you've broken the formula up into a temporary numerator and denominator:

```
// Eq. (5) Loop over all electrons
numer = 1.0 - v*v/c/c;
denom = 1.0 - v/c;
vp = Math.sqr(numer/denom);
```

Java-Specific Advice

1. It's easier to share numerical methods than numerical classes or packages.

Especially if you are new to Java, it may be fun to implement a Java-intensive, object-oriented solution to a purely numerical problem - something involving a package with inherited classes and Java-specific features such as Hashtables. For truly massive projects or scientific libraries, this may be an ideal solution. Your source code may be elegant and efficient.

But for future scientists who wish to borrow only small pieces of your numerical ideas, it may be a nightmare to dissect object-oriented

numerical packages. And what's the sense of importing a massive library when only a single method may be needed? Try to write portable, static methods first, then classes if necessary, and finally packages if absolutely necessary after that.

2. If dedicated classes store global variables, preface their names with Global. If dedicated classes are used as libraries of easily-shared methods, preface their names with Utils.

It's a clever idea because it not only reminds you of what's in the class - it also ensures that the HTML files created by javadoc will all be grouped together. Some examples might include `GlobalEnergyVariables`, `UtilsFile`, `GlobalAntimatter` or `UtilsImageProcessing`.

3. When necessary, port Fortran source code rather than C source code to Java.

Java's syntax most nearly resembles C, but Java does not support pointers - and finding scientific subroutines in C without pointers is nearly impossible.

4. Interfaces are great for numerical constants.

With interfaces you'll have to type in each of your constants (like pi, pi/2, pi*pi), but since interfaces are easy to recycle, you'll only need to type them in once!

5. Don't forget `System.out.println` and the Console.

It's tempting to invoke the Java Virtual Machine with `javaw`, but don't forget that the standard console is a handy place to see error messages and liberally-sprinkled System.out.println comments. Since Java protects itself so well against errors which would be fatal in Fortran or C, it's important to see possible error messages when they are generated, or else you might not realize they are there! If you want to get fancy, there are even good ways to implement your own custom console.

— K. R.

Don't OOP? Don't Worry!

Since the real world is composed of objects related in complex ways, object-oriented strategies are often the best solution to programming problems. One reason for Java's success is that it simplifies OOP development.

However, many scientific problems are not amenable to full-blown object-oriented treatments. The problem may be too simple, such as evaluating an equation. Or sometimes the relationship between objects is just too complex. For example, a very simple equation describes how the moon orbits the earth, but add a third object to the problem (such as the sun or a comet or an asteroid) and the resulting equations can be so intertwined and complex that entirely different calculational techniques become necessary.

Nevertheless, OOP principles can be valuable for scientific programming, and increasing numbers of scientific programmers are now learning to think in OOP terms. For the scientist new to Java, the first lesson to learn is this: the classes are where it's at.

Classes: The Heart of Scientific Programs

Coming from a traditional Fortran or C background, it's easy to look at methods in Java and assume they are like functions or subroutines. They're not. For scientific programming, Java methods are weaker than their Fortran or C counterparts. A scientist expects to invoke a subroutine with a long argument list containing lots of variables, then to have those variables updated and changed upon return.

In Java, variables can be passed into methods (passing by value), but at most only one of them can be changed, such as via a function: `x=myMethod(a,b,c,x);`. Of course, groups of variables can be stored in arrays, and methods can change the contents of arrays (passing by reference). But cheating with variable-arrays is rarely an elegant approach.

This is not a shortcoming of Java — in fact, it's an improvement over Fortran and C! The Java solution is elegant: construct a class and feed it the necessary variables a, b, and c, then provide either public variables or else public methods which return x, like this:

```
MyCalc mc = new MyCalc(a,b,c);
x = mc.x; // or, alternatively . . .
x = mc.getX();
```

Java ends endless parameter lists, in which it is never clear which variables stay fixed and which are changed; and Java prohibits pointers — no comment necessary! And as this example shows, the Java code for such classes is clean and concise, and above all, the programmer's intent is crystal clear (Please see the sidebar "Converting Scientific Subroutines to Scientific Classes").

Converting Scientific Subroutines to Scientific Classes

Subroutines are the building blocks of scientific programs in Fortran and C. They're portable and easy to re-use. But they're not without problems, because it's frequently unclear which of the variables stay fixed and which are changed. And as the needs of the programmer change, and as more features become necessary, subroutines quickly become unreadable spaghetti-style code.

A typical Fortran subroutine might look like this:

```
      SUBROUTINE root (a,b,c,root1,root2)
c Which input parameters are changed?
      real*8 a, b, c, root1, root2, disc
      disc = b*b - 4.*a*c
      root1 = (-b + DSQR(disc))/2./a
      root2 = (-b - DSQR(disc))/2./a
      return
```

```
      end
```

Java classes provide a better way. The input and output parameters are easy to see. Variables can be protected, to prevent inadvertly changing their values. And best of all, a finished class is a file in its own right, complete with useful HTML comments (courtesy of javadoc) - ready to be shared with colleagues and recycled in many different programs!

It's easy to convert this Fortran subroutine to a Java class:

```
/** Solve a quadratic equation, a*x^2+b*c+c=0
public class QuadraticEquationSolver {
  private double root1, root2;
  /** Initialize the parameters
  public void setup (double a, double b, double c) {
    double disc = b*b - 4.*a*c;
    root1 = (-b + Math.sqrt(disc))/2./a;
    root2 = (-b - Math.sqrt(disc))/2./a;
  }
  /** Returns the "+" root
  public double getPosRoot() { return root1; }
  /** Returns the "-" root
  public double getNegRoot() { return root2; }
}
```

A few extra lines of code may be necessary, but the programmer's intent is crystal clear, and useful HTML documentation is automatic. The use of the setup method ensures the class can be reused many times (for example, within a loop) without multiple instantiation. And new methods can be added (such as for mimicking Fortran functions) without changing the code which performs the calculation. Further, the code can be easily modified, perhaps adding an `isThereASolution` method - or possibly, by following the Java convention of returning a -1 when the desired operations could not be performed.

— *K. R.*

Handling Global Variables in Java

A scientific program is about numbers, not (necessarily) about interrelationships between inherited classes or objects. So numerical variables — sometimes many dozens of them — must be easy to group and share between sections of the program. Fortran originally provided a pre- object-oriented tool for this task, the named common statement: global variables were easy to share, but it required significant programmer overhead to ensure appropriate typing and dimensioning.

Java provides better ways. The easiest way is simply to create a separate class for each collection of global variables to be grouped, and to declare these variables as static members of the class. The static keyword ensures that no matter how many instances of the class exist, the variables will all share the same address space — which means there is effectively only one instance of each variable in the class.

A more clever approach is known as the Singleton pattern, a technique which ensures that one — and only one — instance of a class can be created. Each desired collection of variables can be declared in its own Singleton class. Here is an example of these strategies (please see "Interfaces: Where Scientific Bakers Bake their Pi!").

One advantage of global variables stored in classes is that, to use these variables, their names must contain the class instance in which they appear (`gt.x` vs. `x`) - perhaps awkward at first but ultimately a much-needed bookkeeping mechanism.

But for scientific constants which never change, such as `pi` (3.141...) or `e` (2.718...), it is useful to define them in an interface which can be implemented by classes which need them. In fact for scientists usually this is not enough: they want to store all variations (such as pi/2, 2*pi, pi*pi, etc.) which might show up in formulas. The advantage of interfaces is that such constants need to be typed in one time only - thereafter, it's easy to recycle the interface in many different programs!

Scientists Like to Think Global

Java offers advantages over traditional Fortran 77 for storing and managing global variables in scientific programs. By defining, dimensioning and storing variables in a class, the need for repetitive declarations in each subroutine is eliminated!

Here's an example program, `GlobalVariableTest.java`, which demonstrates both static and Singleton-style global variables:

```
public class GlobalVariableTest {
  public static void main (String args[]) {
      GlobalVariableTest.init();
      GlobalVariableTest.calc();
  }
  public static void init() {
      GlobalSingletonTest gt = GlobalSingletonTest.getInstance();
          gt.x = 1.0;
      GlobalStaticTest gst = new GlobalStaticTest();
          gst.y = 2.0;
      System.out.println("X,Y: "+gt.x+", "+gst.y);
  }
  public static void calc() {
      GlobalSingletonTest gt = GlobalSingletonTest.getInstance();
      GlobalStaticTest gst = new GlobalStaticTest();
      System.out.println("X,Y: "+gt.x+", "+gst.y);
  }
}
```

Here's the accompanying file, `GlobalSingletonTest.java`, defining global variables using the Singleton class:

```
/** Demonstrates Singleton-style  global variables */
public class GlobalSingletonTest {
  /** Manages SingletonClass */
  private GlobalSingletonTest() {}
  static private GlobalSingletonTest _instance;
  static public GlobalSingletonTest getInstance() {
    if ( _instance == null )
      _instance = new GlobalSingletonTest();
    return _instance;
  }
  // List of global variables
  public double x;
}
```

And here's the accompanying file, `GlobalStaticTest.java`, defining the global variables using static members:

```
/** Global variables via static members */
public class GlobalStaticTest {
  // List of global variables
  static public double y;
}
```

Better for Scientific Programs: Public Access not Accessor Methods

In these examples, the programmer handles global variables directly (e.g., `gt.x = 1.0;`) rather than by using methods (e.g., `gt.setX(1.0)`). This latter approach is not always optimal in a scientific program which contains dozens of variables which appear in complicated equations! But scientists take note: methods are a useful programming tool to make it obvious when setting or changing important control variables!

— *K. R.*

A useful convention is to preface the name of such global variables classes with the word Global, as in `GlobalEnergy` or `GlobalMaterial`. This not only describes the class, but also ensures that all these classes will be grouped together in any javadoc HTML files which are produced.

Design Patterns for Scientific Programs

Modern programming languages are like children's Lego-type block toys, comprising myriads of small components which can build structures of great complexity. There are a few basic substructures used frequently, such as boxes to build houses or chassis to build cars. By clearly defining these structures, known as design patterns, and by recycling them in programs, the programmer can quickly assemble elegant programs of great complexity. The Singleton pattern for global variables is one such design pattern.

Here we present three design patterns which are useful for scientific programming in Java.

The DataTransceiver Pattern

As the sidebar on antimatter illustrates, some number-crunching programs take hours or even days to run. What happens when, ten minutes before such a program is finished executing, the system crashes or the computer is accidentally switched off? Such problems are not just frustrating, they are also expensive. The scientist's time costs money, and on supercomputers each CPU second may recorded and billed.

When Matter and Antimatter Collide

$$v' = \sqrt{\frac{1 - v^2/c^2}{1 - v/c}}$$

Any Star Trek fan knows what happens when matter and antimatter collide: annihilation! But fortunately for us, the annihilation is not total -- the only thing to annihilate are the miniscule bits of matter and antimatter which have collided. The equally-miniscule light waves emitted during annihilation can be measured easily, and they make a sensitive fingerprint of the matter that's been annihilated. Medical diagnostic techniques called PET scans are based on this process, and they help save countless lives every year.

Prof. Kelvin Lynn and Dr. Marc Weber are two physicists at Washington State University who work at the forefront of antimatter annihilation technology. They build complicated machines, such as the one shown above, which use precisely-controlled beams of antimatter (called positrons) to study new materials and new electronic devices. But to interpret their results, they often need to compare their data with the results of computer simulations.

These so-called Monte Carlo computer programs use probability theory to simulate the complicated scattering processes which occur just before annihilation, as the antimatter beam enters the material and is randomly

scattered by the atoms. These calculations predict what happens by simulating and averaging thousands and thousands of discrete scattering events. Even on the fastest computers, the calculations may take hours or even days, depending on the level of accuracy that the researchers need.

Some of Dr. Lynn's programs use the DataTransceiver design pattern (see separate sidebar below). As the program runs, the results of the simulation are periodically written to ASCII files. This strategy allows the programs to be stopped at any time to look at the intermediate results, and to be restarted again when more accuracy is needed. Computer time is not free, especially on high-speed supercomputers, so this strategy also helps save money and protect the results in case of a computer crash.

— *K. R.*

The DataTransceiver pattern is a solution to this problem. Like a radio transceiver, which both transmits and receives, the DataTransceiver pattern regularly reads and writes all the essential information for a calculation to data files. It allows the calculation to be interrupted at any time, either accidentally (such as a system crash), or intentionally (such as to check the intermediate results) — and then to be restarted again.

And by giving a little thought to the data files which are produced, interspersing them liberally with variable descriptions and user-comments, they make a useful archival record which stores the calculation results together with the initial parameters used to generate them.

The DataTransceiver Design Pattern

The Problem

A numerical calculation without user interaction requires lengthy execution time, perhaps hours or even days. If the computer system crashes considerable time and resources will be lost. The user may periodically wish to interrupt the calculation, to verify the intermediate results.

The Solution

At periodic times the calculation is halted and such intermediate results as are required to restart the calculation are written to datafiles.

Implementation Details

A single class can be used to perform both the data input and the data output operations.

Advantages

- By use of a single class for input and output, the programmer is automatically reminded that any new variables introduced into the calculation during program development must be both initialized and as well as saved.
- By appropriately labelling the output variables in the data file, and by providing space in the data file for user-written commentary, the data file makes a useful archival record which can store the results of the calculation together with the parameters used to generate them.
- By creating a sequence of temporary data files, rather than a single data file which is continually overwritten, the programmer obtains a record of results during intermediate stages in the calculation.

Disadvantages

Depending on the calculation (e.g., one involving dozens of large, multidimensional arrays), intermediate data files may be

large and unwieldly.

Responsibilities for the Programmer

1. Ensure the library methods are functionally independent.
2. Try to use *static* methods whenever possible.

Example

The DataTransceiver design pattern is implemented in many scientific calculations, although scientists may not call it by this name! A good reference to some publications about programs which use this pattern is the 1993 journal article in the *Journal of Applied Physics*, volume 74, pages 3479-3496 (1993).

The scientific programmer will often not know ahead of time how many iterations are required to obtain an accurate answer. The DataTransceiver pattern lets the scientist start and stop the calculation as often as necessary to obtain the desired accuracy.

— *K. R.*

The ScientificLibrary Pattern

We've said in Part 2 that numerical methods are the heart and soul of scientific programs, and that many scientific programs are nothing but well-tested legacy methods patched together in new ways. So there must be good techniques for archiving and maintaining these methods.

The ScientificLibrary pattern provides a good technique. A scientific library is a class which contains numerous independent, static public methods and internal classes, with no global variables. Every scientific programmer has a favorite collection of tools which he/she reuses often, such as for solving an equation, calculating a histogram, writing data to an ASCII file, etc. A scientific library is the ideal repository for these methods. Declaring the methods as static is the ultimate in programmer-friendly strategy: the methods can be used without needing to instantiate the class!

The ScientificLibrary Design Pattern

The Problem

A scientist needs to have easy access to a collection of many small, functionally independent numerical methods. These may be favorite methods the scientist uses often, such as for solving an equation, making histograms or writing columns of data to ASCII files.

The Solution

A class is used as a repository for a "library" of recyclable, interdependent methods.

Implementation Details

The methods are declared as static whenever possible, so they can be used without the need for class instantiation.

Advantages

- Speeds program development by keeping often-used methods at fingertip distance.
- Results in cleaner source code, especially for programs which may use dozens of "library style" numerical methods.
- Easy to modify and maintain — new methods are simply appended to the class.
- HTML documentation (javadoc) is clean and adequately describes method usage.
- Because Java allows defining multiple methods with different parameter lists, the library can easily contain several often-used versions of numerical methods.

Disadvantages

Not well-suited for interdependent methods.

Responsibilities for the Programmer

- The programmer must analyze a calculation for appropriate breakpoints for writing data files.
- The programmer must implement a calculation in such a way that it can be restarted using parameters read from a data file.
- The programmer must ensure the DataTransceiver class has access to the appropriate variables. This may require the use of global variable techniques such as Singleton classes.

Example

There are several useful ScientificLibrary classes in DataScan, a Java-based software application for data analysis in x-ray diffraction and microscopy experiments.

— *K. R.*

As with global variable classes described above, a useful convention is to name the scientific library classes with the word Utils, as in `UtilsMath` or `UtilsFile`. This not only describes the class, but also ensures that all these classes will be grouped together in any javadoc HTML files which are produced.

The NoOOP Pattern

For simple scientific programs, such as a calculation requiring little user interaction, the NoOOP pattern (pronounced nope) can be useful. It's a way to implement a Fortran-like procedural program. It may contain methods to initialize the starting data, methods to perform the calculation, and methods to output the results.

This pattern is not the optimal use of Java's powerful OOP resources, as the name cleverly suggests. But traditional (non-scientific) software developers should take note: a scientific programmer may have years of successful software design experience, though none of it object-oriented. The NoOOP pattern provides a perfect place for a scientist to begin with Java.

The NoOOP Design Pattern

The Problem

A scientist — possibly a scientist new to Java and without experience with classes and inheritance and other OOP techniques — needs to quickly write a short program, perhaps to evaluate an equation.

The Solution

A Fortran- or C-style procedural program can be easily developed using Java.

Implementation Details

A single class can contain all variables and methods to perform all required tasks.

Advantages

- Extremely quick program development for simple, procedural programs.
- Only superficial, not detailed (esp. OOP) knowledge of Java is required.

Disadvantages

- Not well-suited for longer programs with many methods or global variables.
- Difficult to modify and maintain.

Responsibilities for the Programmer

- Must be prepared to completely redesign the program when the calculation outgrows the procedural framework

Example

This example shows a simple procedural program to find when a user-specified function is equal to zero. For this example, the function is the sine of x (`sin(x)`), and useful "test" parameters are 0.1, 5, and 1e-7. If the program runs correctly, it will report that the function is zero for the value 3.1415 ...

Some Java features this program demonstrates:

- Since the `main` method is static, it is especially useful to invoke or encapsulate the procedural program within a different method, to simplify how the methods are invoked.
- How to perform basic input and output from the console.

```
public class Procedural {

    // This function just starts the program
    public static void main(String[] args) {
        Procedural myproc = new Procedural();
    }

    // The main procedural program goes here
    public Procedural() {
        double d, dStart, dStop, dStepSize;
        dStart = getDoubleInput("Enter startpoint: ");
        dStop = getDoubleInput("Enter endpoint: ");
        dStepSize = getDoubleInput("Enter stepsize: ");
        for (d=dStart; d<dStop; d+=dStepSize) {
            if ( myFunction(d)*myFunction(d+dStepSize) < 0 ) break;
        }
        if (d>dStop) System.out.println("No zero found. ");
            else System.out.println("The function is zero when x = " + d);
    }

    // Example of how to read double numbers from the keyboard
    double getDoubleInput(String sMessage) {
        double d = 0.0;
        System.out.print(sMessage);
        try {
```

```
            d = Double.valueOf(new java.io.DataInputStream(System.in).readLine()).doubleValue();
        } catch (java.io.IOException e) {}
        return d;
    }

    // Example of a user-defined function
    double myFunction (double x) {
        return Math.sin(x);
    }

}
```

— K. R.

Scientific Programming for the 21st Century

There's good reason for Java's explosive popularity: it gives business programmers the tools they need to write effective, easy-to-maintain programs. But as we've discussed, Java offers excellent tools for scientific programmers as well. Scientists switching to Java will be rewarded with fast-running, platform-independent programs — which are far easier to modify and maintain than their Fortran or C counterparts!

About the Author

Dr. Kenneth A. Ritley is a consultant with HP Consulting in Sindelfingen, Germany. Until recently, he was a physicist in Department Dosch at the Max-Planck-Institut für Metallforschung (Metals Research) in Stuttgart, Germany. He's made scientific computing contributions in the wide-ranging fields of astronomy, antimatter, high-temperature superconductivity, and magnetism.