Conventions for Arithmetic Operations in Java

Conrad Weisert
Information Disciplines, Inc., 1205 Madison Park, Chicago, Illinois 60615
10 August, 1997
(Minor revisions Dec., 2001, based on practices recommended by Joshua Bloch in *Effective Java*, Addison-Wesley, 2001, ISBN 0-201-31005-8)

Background

Experienced programmers learning Java are dismayed to discover that they can't apply arithmetic and relational operators to operands that are instances of the classes they define. Java won't allow a programmer to use both the object paradigm and ordinary expression syntax on the same data items.

Some of us regard that limitation as a serious impediment, while many others view it as a mere notational inconvenience. A few even consider it a positive protection against *misuse* of operations! Whatever our opinion, application programmers still need to do arithmetic on application-domain[1] data. In C++ we might code:

```
Money  amtDue, unitCost, shipChg;
Date   startDate, dueDate;
  .          .
  .          .
amtDue  = unitCost * qty + shipChg;
dueDate = startDate + 14;
```

What's the Java equivalent of that? How can we best circumvent Java's restrictions and minimize their impact on software development and maintenance? We shall propose a set of conventions, but first let's dispose of two ill-advised techniques that Java zealots sometimes propose:

A false solution: saying "no" to numeric objects

Why not use only Java's built-in *primitive* data types for elementary numeric data, i.e. forgo the benefits of object-orientation for application-domain data[2] like amounts of money, dates, distances, and weights? If we took that advice we'd rewrite the above example as:

```
double  amtDue, unitCost, shipChg;
int     startDate, dueDate;
 .              .
 .              .
amtDue  = unitCost * qty + shipChg;
dueDate = startDate + 14;
```

This approach is naïve and very damaging to program maintainability. The benefits we'd forgo include not only type safety, but also localizing our choice of data representation, a major advantage of OOP. Knowledgeable OOP practitioners scorn that technique.

Another false solution: expose a data representation

Well, then, suppose the program just converts back and forth between numeric objects and some publicly known representation[3]. When the problem calls for an arithmetic or comparison expression, the program extracts a primitive internal representation from the objects, does the operations, and, if necessary, converts the results back to objects.

Whenever we find that a Java class has an accessor method called **`value`** or **`getValue`** we suspect that it was designed with this technique in mind. The programmer might code our original example like this:

```
Money  amtDue  = new Money();
Money  unitCost, shipChg;
Date   startDate;
Date   dueDate = new Date();
  .                .
  .                .
amtDue.setValue(unitCost.value() * qty
              + shipChg.value());
dueDate.setValue(startDate.value()+14);[4]
```

Aside from the readability obstacles, this technique repudiates what object-oriented programming is about. In true OOP we don't consider the *value* of an object as something distinct from the object itself.

It makes little difference whether the publicly known representation is the same as the hidden internal representation or different. If it's known throughout an application or in multiple applications, then it's going to be costly to change. The year-2000 crisis some organizations are now suffering exemplifies the consequences of not localizing knowledge of data representation.

---

1 As contrasted with program *housekeeping* data, such as loop counters and subscripts.

2 There's nothing wrong, of course, with using built-in types for program housekeeping data.

3 This technique used to be common among beginning Smalltalk programmers, and still appears in occasional articles and textbooks.

4 This expression wouldn't work with the *standard* **`Date`** class that comes with the JDK, which demands a more awkward construction.

This technique is even messier where we need *multiple* accessors to retrieve the value or where the arithmetic operations are more complicated. For example, how would we convert this straightforward C++ code to Java?

```
Complex  x, y, z;
  .
  .
z = x * y;
```

We'd either (a) have to write out the complex multiplication algorithm inline:

```
z.setValue(x.realPart() * y.realPart()
         - x.imagPart() * y.imagPart(),
           x.realPart() * y.imagPart()
         + x.imagPart() * y.realPart());
```

which is unacceptably repetitive and error-prone, or (b) invoke a non-method (static) function:

```
z = Complex.multiply(x,y));
```

which is all too reminiscent of vintage 1959 Fortran. OOP is supposed to hide that sort of detail.

An ugly but conceptually sound solution: operator functions

Suppose we give up the use of traditional arithmetic operators altogether in favor of *named methods*. Then the Java programmer would code something like:

```
Money  amtDue  = new Money();
Money  unitCost, shipChg;
Date   startDate;
Date   dueDate = new Date();
   .               .
   .               .
amtDue.set(
 ((unitCost.mpy(qty)).add(shipChg)));
dueDate.set(startDate.add(14));
```

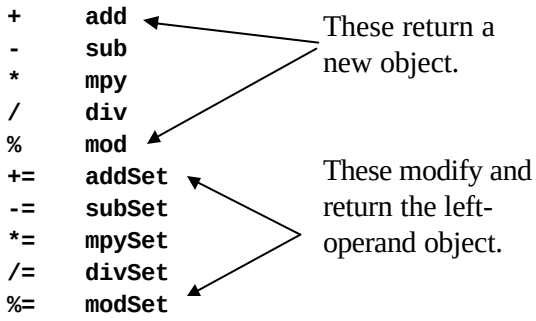
Yes, that's awfully tough to read compared with an arithmetic expression. It exposes the absurdity of this sort of Java defense:

> " . . . the language designers decided (after much debate) that overloaded operators were a neat idea, but that code that relied on them became hard to read and understand." [5]

> "One of the major problems with operator overloading is that it gives the programmer the power to easily write code that is difficult to read."[6]

But despite its ugliness, this solution is in perfect harmony with the letter and the spirit of OOP. We can localize knowledge of internal data representations. We can preserve dimensional integrity[7]. We can prevent illegal operations, such as multiplying two amounts of money or adding two dates. We can provide just about all the functionality we'd provide in C++[8], but of course in Java's clumsy notation.

If we adopt this solution, we should agree on *standard* names for the operator functions in order to avoid needless confusing variation. For the binary arithmetic operators I've used these in a number of such classes:



Let's also pick standard names for the relationals[9]:

```
==   equals
<    lessThan
>    greaterThan
```

We can implement **`equals`** in either of two ways:

---

5 David Flanagan: Java in a Nutshell, O'Reilly & Associates, 1996, ISBN 1-56592-183-6, p. 35.

6 Paul Tyma, Gebriel Torok, & Troy Downing: *Java Primer Plus*, Waite Group Press, 1996, ISBN 1-57169-062-X, p. 254.

7 i.e. the units in which the result is expressed. See my paper "The Point-Extent Pattern for Dimensioned Numeric Data", ACM *SIGPLAN Notices*, November, 1997.

8 Except for having a primitive data type as the *left* operand. That is **`x-1`** is rendered as **`x.sub(1)`**, but there's no direct way to render **`1-x.`** Given a unary negation function (**`minus`**), however, the latter can be rendered as **`(x.minus()).add(1)`**.

9 Java has gotten off to a regrettably inconsistent start here. The standard **`String`** class uses a catch-all **`compareTo`**, while the standard **`Date`** class[9] uses **`before`** and **`after`**, and the so-called "wrapper" classes don't support ordering relations at all.

- by *overriding* that method in the **Object** base class, or
- by *overloading* (i.e. ignoring) the **Object** method.

The first alternative will meet with the approval of many in the Java insider community, but at the cost of both:

- extra complexity and repetition in the method (you have to test whether the argument is really of the expected type and then cast it), and
- being obliged to implement a *hashing* function, which Bloch reminds us[10] is essential to honor the **equals** "contract".

Some Java purists will also prefer the ugly **compareTo** function to our relational operators. For a robust class, we can provide both.

Whichever conventions we follow, they make all our numeric classes consistent. The client-programmer never has to look up the supported methods. If a particular operation is meaningful, then it should work in the expected way. And although functional notation will never be as easy to read as ordinary expressions, experience indicates that programmers who use it regularly get used to it.

A comparative example

Readability is hard to judge in isolated lines of code. Here's an example[11] comparing C++ and Java for a reasonable implementation of **Date**[12] and **Money**:

Java version

```
Date dt, startDt = new Date(),
         endDt   = new Date();
Money cumSales   = new Money(0);
Money sales      = new Money();

System.out.println("Enter start date ");
startDate.get(System.in);
System.out.println("Enter end date: ");
endDate.get(System.in);
for (dt = new Date(startDt);
      dt.lessThan(endDt); dt.addSet(7));
   {System.out.print(
      "Enter sales for week of " + dt);
    sales.get(System.in);
    cumSales.addSet(sales);
   }
```

C++ equivalent

```
Date  dt, startDt, endDt;
Money cumSales = 0, sales;

cout << "Enter start date: ";
cin  >> startDt;
cout << "Enter end date:  ";
cin  >> endDt;

for (dt = startDt; dt < endDt; dt += 7);
  {cout << "Enter sales for week of "
        << dt;
   cin >> sales;
   cumSales += sales;
  }
```

Although the C++ version is much easier to understand on first reading, the Java version will become clear enough to a programmer familiar with our conventions.

A red herring: operator misuse

We keep hearing presentations and reading articles that defend the omission of overloaded operators from Java on the grounds that it prevents a misguided programmer from defining operators with bizarre or misleading semantics. Presumably that's what Flanagan[5] and others in the "Java community" are worrying about.

But that's no concern for serious software developers or their managers. Any programmer so undisciplined as to define **+** to do subtraction is likely also to define **add** in the same misleading way or to commit all sorts of other programming sins. Competent professionals are unlikely to do either.

Any rich programming language presents opportunities for misusing its facilities, and we can't police them all. A *good* programming language makes it easy to follow principles of good programming practice. Java is getting there, but it has quite a way to go.

Using named operator functions yields code that isn't particularly readable, but it's by far the least among evils. Let's hope Java will one day support numeric and relational operators so we can lay this issue to rest.

---

10 Joshua Bloch: Effective Java, Addison-Wesley, 2001, ISBN 0-201-31005-8, item #8..

11 We omit commentary in order to focus on code readability.

12 Definitely not Java's atrocious *standard* **Date** class. This is one place where we feel justified in substituting a well designed class for a standard library class without changing its name.