

Re: Operator overloading [was Re: 7.0 wishlist?]

Source: <http://coding.derkeiler.com/Archive/Java/comp.lang.java.programmer/2008-11/msg01579.html>

- *From:* Jerry Gerrone <scuzwalla@xxxxxxxx>
 - *Date:* Mon, 17 Nov 2008 04:32:07 -0800 (PST)
-

On Nov 16, 11:27 pm, Joshua Cranmer <Pidgeo...@xxxxxxxxxxxxxxxx> wrote:

A reason why immutability is a good thing. In any case, the intent I attempted to convey was the fallacy of naïve translation of mathematical concepts to programming.

Somehow, I doubt Harry was proposing that the translation be naïve. :)

A side effect is a side effect. True, some may be only visible to the internal code in the API, but, as far as the JLS is concerned, it's a side effect if some Java code could detect the difference.

As I understood the proposal, it would have to break encapsulation to do so, and the semantics of well-behaved code wouldn't be dependent on the exact order of evaluation, although the performance might be.

Poorly-behaved code would behave in a manner that would require language-lawyering, just like it already does with multiple "i++" in one line of code. An example I think Harry'd mentioned explicitly.

(The above assumes correct code).

While users ideally need only deal with correct code, the specification must deal with any valid code.

I'm sure it would.

To paraphrase a CSS spec developer: "You may not worry about what a web page looks like on a 3x3-pixel screen, but a browser has to worry about that."

Re: Operator overloading [was Re: 7.0 wishlist?]

A browser might as well give up on that and assume some sort of minimum that's a darn sight bigger than 3x3; no matter what it does, the results will be unusable below some threshold resolution, so it might as well be designed to give the best possible results above that threshold regardless of the effects doing so has below that threshold.

Worrying about how the presentation might degrade at 3x3 is like worrying about how you'll get to work the next day if your car goes over that cliff at 40mph, when you happen to be in that car.

I doubt a change that would break either of these conditions would fly:

- * User-overloaded operators would evaluate in the same order as the operators they are based on, i.e. operands are evaluated as they appear and associate as the operators (left-to-right except for the few right-to-left ones).
- * An expression involving user-overloaded operators can be rewritten as an expression (not a series of statements) using the base methods.

I don't see why the second one is necessary. The first one is strongly desirable, but for the case of OlderClass op NewerClass with NewerClass defining op it will generally be necessary to give up the second one to get the first.

2. The spirit of operator overloading should be agnostic to the name of the operator being overloaded, modulo something like interface names or the arity of the operator.

Why?

A proposal I saw on the possibility of "contracts" (or static implementation of interfaces) gave me an idea to work around the LHS-issue with a bit more workaround:

```
public interface Addable<Left, Right, Value> {
    Value add(Left left, Right right);
}

public class Matrix<T extends ...> implements
    static Addable<Matrix<T>, Matrix<T>, Matrix<T>>,
    static Multipliable<T, Matrix<T>, Matrix<T>> {

    static Matrix<T> multiply(T scalar, Matrix<T> matrix) {
        Matrix<T> result = new Matrix<T>(matrix);
        for (T[] row : result.rows) {
```

Re: Operator overloading [was Re: 7.0 wishlist?]

```
    for (int i = 0; i < row.length; i++) {  
        row[i] = T * row[i];  
    }  
}  
}
```

with suitable `super` and `extends` thrown in the type definitions.
Writing generics libraries is still a pain, though.

Thus the compiler could then translate the expression `|scalar * matrix|`
to `|multiply(scalar, matrix)|...`

Seems pointless to introduce a whole new kind of "interface" for this purpose. In that case, you might as well just allow static methods `Foo.operator+` (or whatever, I used the C++-like name) and operator overloading enabled using `import static`.

Given such a fix, `Addable` and similar interfaces enable extending `java.util` (or `java.math`?) with useful new things, for example an `accumulate` method that can take a `Collection<Addable<Foo>>` and return a `Foo` that is their sum, or an `Accumulator<Foo>` that can perform accumulations.

Implementation via interfaces opens up wide possibilities that could not be offered by other proposals for operator overloading. Indeed, it is probably safe to say that these interfaces alone—along with retrofitting the appropriate primitive wrappers, `BigInteger`, and `BigDecimal`—would be powerful enough, even if operator overloading didn't follow.

They would also make operator overloading that much "closer", in some sense, such that it'd have a decent shot at getting in in the `*next*` Java version after the one that added Harry's proposed interfaces.

One step at a time, I'd say. Pin down the implementation of regular operator overloading before trying to extend it.

I thought this was now about implementing these interfaces and some compiler tweaks, then adding operator overloading later as a layer on top?

Harry?

Re: Operator overloading [was Re: 7.0 wishlist?]

Re: Operator overloading [was Re: 7.0 wishlist?]

Hmm...

```
public interface CumulativeAddable<Param, Return> extends
    Addable<Param, Param, Return> {
    public Return add(Param... params);
}
```

Erk. I think I prefer Accumulator.

fully generalize this -- left-summand type, right-summand type, and result type; `StringBuilder` would implement `Accumulator<String, Object, String>` since it can add any `Object` to a `String` and return a `String`, for instance.

But it can also add any `String` to an `Object` and get a `String`. Unfortunately.

As I understood Harry's latest proposal, this would basically be `StringBuilder`, not `String`.

Eh, then the third type parameter's not needed. Or the first isn't. It'd just be `Accumulator<Object, String>`, left hand argument is what you can add, right hand argument is the type of the running sum and of the result. `Addable<Object, String>` on `String`, also, since it can be added to any `Object` to get a `String`.

Eh. You also need a way to get the "zero" for the accumulation. `Addable` needs a `getIdentity()` method, and ugly-enough this needs to be non-static for an interface to specify it. For strings it would return "", for numbers zero.

Then again, this could be generalized further. Accumulation of products, with identity 1; of logical-ors, with identity false; of logical-and, with identity true; and so forth. Maybe some notion of generalized commutative, associative operations? Mathematicians call that an abelian group. We'd need something less exotic and easier to type. And yes, we'd need to be able to implement the same interface with multiple sets of generic parameters in a single class. That would need reified generics, since otherwise method foo taking an `Addable<Foo, Bar>` that's also an `Addable<Baz, Quux>` and doing its `add` won't know whether to call `Bar add(foo)` or `Quux add(bar)` at runtime due to erasure.

.