Language designer's notebook: Quantitative language design

Using real-world data to drive language evolution decisions

Skill Level: Advanced

Brian Goetz (brian.goetz@oracle.com)
Java Language Architect
Oracle

12 Apr 2010

For any given programming language, there is no shortage of new feature ideas. Language designers must not only perform the difficult task of deciding which of many possible (and often incompatible) language features should receive priority, but they also must consider that new language features can interact with existing ones in surprising, and sometimes incompatible, ways. Language evolution often requires making a trade-off between the benefits of enabling desirable new patterns of coding and the costs of potentially breaking some existing "weird" code. In this situation, being able to quantify — using real-world data — just how unusual that "weird" code is can provide valuable clues to which way a decision should go.

About this series
Every Java developer probably has a few ideas about how the Java language could be improved. In this series, Java Language Architect Brian Goetz explores some of the language design issues that have presented challenges for the evolution of the Java language in Java SE 7, Java SE 8, and beyond.

At my first JavaOne, I attended a talk by Java™ Architect Graham Hamilton, who observed, "Every developer arrives in the Java community with one or two feature ideas for the Java language in their back pocket." And sure enough, I found I had a few in my back pocket too. In this new series, *Language designer's notebook*, I will explore the process of evaluating which new ideas should make their way into the Java language, and the challenges of making room for new features in a widely used

© Copyright IBM Corporation 2010

language. In this first installment, I'll talk about how real-world data can be used to inform and influence language evolution decisions.

Evolving existing languages

At some point in our careers, we've probably all been tempted — probably out of frustration — to design a new programming language. The way we want to describe a solution to a particular problem in front of us, and the tools we have for expressing that solution, often do not align perfectly. The words we use to describe the resulting code are evocative: *clunky*, *bloated*, *smelly*. Sometimes the problem lies with our own shortcomings — that our solution is simply not elegant or clever enough — but sometimes, even the most elegant solution feels less elegant when expressed in a real programming language.

Language designers for new languages spend most of their time thinking about features; language stewards for existing languages spend most of their time thinking about compatibility.

The obvious solution — design your dream language — might be fun, but it's an uphill battle. Even if you are building on an existing rich platform such as the JVM — which provides features such as garbage collection, concurrency control, security, debugger support, and rich runtime libraries — a lot of work still remains: designing the language, implementing a compiler, implementing any runtime features not already supported by the underlying platform, designing and implementing core libraries, developing IDE integration, and so on. But all of that is still the easy part! The hard part comes later: getting users to adopt your new language, and dealing with the inevitable complaints and suggestions for improvement. And then comes the really hard part: facing the reality that your dream language is not perfect, that it needs to be evolved, and that you need to choose between compromises in your evolution plan and incompatible changes that will break your users' existing code. When you have 10 users, it's far easier to justify incompatible changes than when you have 10 million. Language designers for new languages spend most of their time thinking about features; language stewards for existing languages spend most of their time thinking about compatibility.

A simple example: More precise rethrow analysis

One of the new language features in Java SE 7, introduced as part of Project Coin (see Resources), is more precise rethrow analysis for exceptions. In Java SE 6, the language uses only the declared (static) type of an exception parameter to compute the types of exceptions that might be thrown from a block, meaning that the program in Listing 1 won't compile:

© Copyright IBM Corporation 2010

Listing 1. Code that won't compile under Java SE 6, because foo() is not declared to throw Throwable

```
void doIo() throws IOException { ... }

void foo() throws IOException {
    try {
        doIo();
    }
    catch (final Throwable t) {
        log("Exception in foo!", t);
        throw t;
    }
}
```

It's clear what the code in Listing 1 wants to do. The only things that can be caught by the `catch` clause are `IOException` and unchecked exceptions (`RuntimeException` and `Error`), all of which are throwable from `foo()`. But the exception parameter `t` is declared as a `Throwable`, and `Throwable` cannot be thrown from `foo()` unless the `throws` clause is modified. But this seems a little harsh. The compiler can perfectly well figure out which types of exception `t` could hold and prove that Listing 1 will not violate the contract of `foo()` — without making the programmer jump through hoops. The more precise rethrow feature enables exactly this sort of analysis: the compiler determines the set of exception types an exception parameter might hold from the set of exceptions that can be thrown by the block (and by the `catch` clauses that come before it.)

It turns out that this more precise rethrow analysis, while clearly a useful feature, interacts with the existing reachability analysis for `catch` blocks (wherein the compiler will reject `catch` blocks that catch exceptions that are not thrown from their corresponding `try` block or are already caught by a previous `catch` block). This could break some existing code, such as the code in Listing 2:

Listing 2. Code that compiles under Java SE 6 but would be rejected under the proposed more precise rethrow analysis rules

```
class Foo extends Exception {}
class SonOfFoo extends Foo {}
class DaughterOfFoo extends Foo {}

class Test {
    void test() {
        try {
            throw new DaughterOfFoo();
        } catch (Foo exception) {
            try {
                throw exception; // ***
            } catch (SonOfFoo anotherException) {
                // Question: is this block reachable?
            }
        }
    }
}
```


© Copyright IBM Corporation 2010

Under the old analysis, the program in Listing 2 is legal because the `throw` statement at *** is presumed to throw `Foo`, and so it is conceivable for the nested `catch` clause to catch a `SonOfFoo`. But the more precise analysis can show that the only thing being caught at *** is a `DaughterOfFoo` (or an unchecked exception), so the nested catch clause is unreachable because it's known that `SonOfFoo` cannot be caught at this point, and therefore should be rejected as an error. If we were to take the strict position of "never break existing code," we would be forced to reject this useful new feature. But the code in Listing 2 seems pretty contrived. It would be a shame to lose this nice new feature just so we don't break this contrived-seeming code. So what should we do?

This is a classic trade-off in language evolution: weighing the benefits of improving the language in an obvious way against the unknown risk of breaking some (possibly weird) programs. It is easy to believe wishfully that no one would ever code like this, but a casual perusal of a few coding-support websites is enough to demonstrate the folly of making such generalizations.

The initial design of the more precise rethrow analysis also required the `final` modifier on the exception parameter `t`. The intent of this restriction was twofold. First, it was meant to prevent someone assigning, say, an `SQLException` to `t` (which would be a legal assignment, but then would no longer be an acceptable candidate for rethrow). Second, it was intended to reduce the chance of breaking existing code, because relatively few exception parameters are already `final`; therefore even if examples like Listing 2 exist in the wild, turning on the more precise rethrow analysis only if the exception parameter is `final` would greatly reduce the chance of breaking existing code. However, the `final` restriction is irritating — developers will grumble that they must declare something `final` when the compiler can already figure out everything it needs to know. Historically, decisions like this were made on gut feel. But there is a better way: Gather some data.

Corpus analysis

If one of the challenges of evolving a widely used language is that there's a lot of code out there that we don't want to break, one of the benefits is that there's a lot of code out there that we can use to analyze how often certain idioms come up. By grabbing a big body of code (a *corpus*) and running an analysis tool (often an instrumented version of the compiler) over it, you can learn a lot about how often certain idioms come up in the real world, and whether they are prevalent enough to influence how we evolve the language.

As a starting point for evaluating the effectiveness of using a corpus for language evolution decisions, we took several large and actively maintained open source codebases: the JDK libraries, Apache Tomcat, and NetBeans. We then ran our analysis tool to detect whether any instances of `catch` blocks would compile under the old analysis rules but fail to compile under the new rules. We found none.

© Copyright IBM Corporation 2010

This was extremely encouraging, because we could then go from the subjective "I don't think a lot of people would code like that" to the more concrete "Analysis of millions of lines of code shows that this pattern does not commonly occur." Although this doesn't mean that this pattern of coding would never occur, it does provide real evidence for its rarity, rather than relying on gut feeling.

On the JDK code base, our analysis showed:

- Number of `catch` blocks: 19,254
- Number of effectively `final` exception parameters (those that are not `final` but behave as if they were): 19,197 (99.7 percent)
- Number of explicitly final exception parameters: 16 (0.08 percent)
- Number of non-effectively-`final`, non-`final` exception parameters: 41 (0.21 percent)
- Number of cases where compilation would fail under precise analysis: 0

Armed with this data, we were willing to drop the requirement that the `catch` formal be `final` in order to get the more precise analysis — which made a lot of developers happy.

Our experience with this limited corpus was so encouraging that we now make corpus analysis a key part of our language evolution process. In addition to the code bases I named earlier, we now incorporate the Qualitas Corpus, a curated collection of more than 100 popular open source packages (see Resources). (Analysis over this corpus also turned up no cases where the more precise analysis breaks existing code.) In the future, we plan to incorporate source bases for closed-source Oracle products as well, to further increase the predictive effectiveness of our corpus analysis.

Another example: Generic type inference

Another language feature introduced in Java SE 7, also as part of Project Coin, is *diamond*, or type inference for generic type arguments. The motivation for this feature is the needless redundancy — a frequent source of irritation for developers — in code like this example:

```
public List<String> list = new ArrayList<String>();
```

The type arguments need to be repeated on both sides of the assignment, but most of the time that repetition is redundant. Instead, the compiler could figure it out for itself using type inference. With type inference, the compiler figures out the types of

© Copyright IBM Corporation 2010

variables based on constraint solving, rather than making the programmer declare them explicitly. Some languages, such as ML, make heavy use of type inference, but type inference only arrived in the Java language in Java SE 5, to support generic methods. Type inference is not perfect and can sometimes fail in confusing ways (this is why we sometimes must explicitly declare witnesses to type parameters to generic methods using the `foo.<T>bar()` syntax), but limited type inference can often enable us to have the benefits of static typing with less, well, typing.

The diamond feature uses type inference to figure out the type parameters on the right side of the assignment, allowing the preceding example to be rewritten as:

```
public List<String> list = new ArrayList<>();
```

This specific example gets only six characters shorter when we apply the diamond feature, but other examples get significantly shorter, because type arguments can have much longer names, and some types have multiple type arguments (for example, `Map<UserName, List<UserAttribute>>`).

Type inference is a powerful technique, but it has limitations, especially within the context of languages like Java that feature both parametric polymorphism (generics) and inclusive polymorphism (inheritance). Fundamentally, type inference is a constraint-solving technique, and there are often multiple reasonable choices for which constraints to start with.

As a concrete example, consider Listing 3, which declares a generic type `Box<T>` and invokes the `Box` constructor:

Listing 3. Typical use of the diamond feature from Java SE 7

```
public class Box<T> {
   private T value;

   public Box(T value) {
      this.value = value;
   }

   T getValue() {
      return value;
   }
}
... = new Box<>(42);
```

What type should the compiler infer for `T` in the assignment at the bottom? Certainly a lot of types could be inferred for `T`, given what we know: `Integer`, `Number`, `Object`, `Comparable<?>`, `Serializable`, as well as some pretty weird-looking types (such as `Object & Comparable<? extends Number> & ...`).

While this feature was being designed, the community proposed two competing

© Copyright IBM Corporation 2010

inference schemes:

- Use the type of the variable to which it is being assigned (call this *simple*).
- Use assignment context, as in the simple scheme, and also use the type of constructor arguments (call this *complex*).

Proponents of both schemes claimed their algorithm was "better," and each cited as evidence cases where the other scheme failed. Of course, both worked well in cases where the left-hand side has a simple type like `Box<Integer>`, but things started to get tricky when the left-hand side is something like `Box<? extends Number>`. In this case, the simple algorithm produces `Box<Number>` (because it uses only the types from the left-hand side), whereas the complex algorithm produces the more precise inference `Box<Integer>` (because it can take into account the type of the argument 42). You might think, "That's great, I like the complex algorithm because it gives a more precise answer." But that's just one case. In the case where the left-hand side is `Box<Number>`, the complex algorithm yields an inference that would cause a compile error: The complex algorithm infers `Box<Integer>`, which is not assignable to `Box<Number>` (recall that generics, unlike arrays, are not covariant, so assigning a `Box<Integer>` to a `Box<Number>` produces a type error), whereas the simple algorithm yields a `Box<Number>`, which in this case is a better choice. So the more complex algorithm doesn't always yield the perfect answer.

The story is slightly different when the diamond expression appears in a method-invocation context. Consider a `Box<>` that appears as an argument to the `m()` method:

```
void m(Box<Integer> box){...}
...
m(new Box<>(42));
```

Because the Java language types expressions bottom-up — so that the types of method arguments are available before overload resolution is performed — the simple algorithm has nothing to work with here. It therefore must conclude that the argument is of type `Box<Object>`, which will cause a compilation error after method resolution because `Box<Object>` is not assignable to `Box<Integer>`. On the other hand, the complex algorithm yields `Box<Integer>`, because that is the most specific type it can infer using the constructor arguments.

So, it's easy to find examples where the simple algorithm produces a better answer, and some where the complex algorithm does. How, then, should we choose? Historically, these decisions were made on a gut-feel basis about which cases were more prevalent or important, or which errors were more surprising. But given that diamond is a feature intended to simplify an existing idiom, we can use the many instances of generic method invocations to guide us. So we created a specialized

© Copyright IBM Corporation 2010

version of the compiler — the "diamond finder" — that identifies when explicit type parameters can be removed without causing compilation failures, and we tabulated the results for both algorithms.

The results were interesting. On approximately 200,000 instances of generic constructor invocations, both algorithms predicted the correct result approximately 90 percent of the time. This meant that both schemes were effective, and neither was more effective than the other. So at the very least, we learned that the feature is useful enough to carry its weight — that it predicted correctly the vast majority of the time, making things easier for developers. We also learned that each predicted a slightly different 90 percent — neither was a subset of the other. Because neither was strictly more predictive than the other, we couldn't pick one algorithm now and upgrade to the other later. But given that both algorithms were approximately equally effective, this meant that they were both reasonable choices, and therefore we could make the decision on the basis of secondary considerations. Eventually we chose the complex algorithm because it is more similar to the behavior of type inference elsewhere in the language. That decision makes the language more consistent, makes the implementation more maintainable, and offers more opportunities to take advantage of future improvements in type inference.

Wrap-up

When designing new language features, it's hard to predict what developers will do with them, how they will change coding patterns, or in what ways developers will be frustrated by their shortcomings. But, when refining existing language features, we can use statistical analysis on existing code bases to answer questions about what developers actually do with those features — and prioritize features that will have a more significant impact on how people actually code. This information is invaluable for making language evolution decisions. In Java SE 7, we used corpus analysis to refine several key language design decisions, and this helped guide us to better decisions. Corpus analysis will be an essential part of our toolbox from now on.

Acknowledgments: The work described here was done by Joe Darcy and Maurizio Cimadamore of Oracle.

© Copyright IBM Corporation 2010

Resources

Learn

- Project Coin home page: The Coin features were standardized by the JCP as part of JSR 334.
- Diamond operator & implementation strategies (v3) An analysis from the coin-dev mailing list on the two alternative type-inference algorithms for more precise rethrow analysis.
- Proposal: Improved Exception Handling for Java: The Coin proposal for multi-catch.
- Qualitas Corpus: The Qualitas Corpus is an academically curated collection of source code from more than 100 open source Java projects.
- Project Coin: Small Language Changes in JDK 7: A video presentation on Project Coin by Joe Darcy and Maurizio Cimadamore.
- *Java Concurrency in Practice* (Brian Goetz, Addison-Wesley, 2006): Brian's book is the definitive work on concurrent programming in Java.
- Articles and tutorials by Brian Goetz: Check out Brian's other writings on developerWorks.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Open beta program for IBM SDK for Java 7: The open beta program provides licensed access to the latest IBM SDK for Java 7 beta.

Discuss

- Get involved in the developerWorks community.

About the author

Brian Goetz

Brian Goetz is the Java Language Architect at Oracle and a veteran contributor to developerWorks. Brian's writings includes the *Java theory and practice* column series published here from 2002 to 2008, and the definitive work on Java concurrency, *Java Concurrency in Practice* (Addison-Wesley, 2006).

© Copyright IBM Corporation 2010