



artima developer *Best practices in enterprise software development*

[Articles](#) | [News](#) | [Weblogs](#) | [Buzz](#) | [Chapters](#) | [Forums](#)

[Leading-Edge Java](#) | [Discuss](#) | [Screen Friendly Version](#) | [Email](#)

Sponsored Link • [Effective C++ in an Embedded Environment](#) - get the PDF eBook, only \$24.9

Leading-Edge Java

# From JavaOne 2009: On Trust and Types

A Conversation with Klocwork's Gwyn Fisher

by Frank Sommers

June 11, 2009

## Summary

The JVM's promise of write once, run anywhere works most of the time. But it can also make programmers complacent about cases where the WORA promise fails, says Gwyn Fisher, CTO of [Klocwork](#), maker of the eponymous code analysis tool. One such case is resource management, where the JVM's latent garbage collector necessitates that developers think about the machine behind the VM.

In this interview with Artima, Fisher describes cases where well-known Java APIs work differently based on deployment platform, and how well-defined type systems can make resource management more reliable:

---

**Gwyn Fisher:** The Java community is a bit like the iPhone store: No matter what you want to do, we've got an app for that. In the Java community, it's more like, we've got an API for that, or we've got a framework for that.

Having so many APIs and frameworks is great most of the time. The downside is that the API or framework might not work the way you want it to work. That's especially the case when you realize that some of the Java APIs work some of the time on some of the platforms, and on some of the architectures, and the rest of the time, you're on your own.

That there are differences in how a framework or API works on various platforms is not new, of course. But in the Java community, we've been conditioned to trust that the JVM isolates us from platform-related divergences in how an API functions.

We've run into one of those issues very recently. It's a trivial issue, and has to do with

finding the user's home directory. Yes, we've got an API for that: `System.getProperty()`. You ask it for the user's home directory, and you get ... the wrong answer, on some platforms, some of the time.

On Linux or Solaris, you get the right answer. On Windows you may sometimes get the wrong answer, if you are logging in interactively, through terminal services, through an ssh session, or through some third-party rsh. Each of those is going to have a subtly different take on how your environment is considered. The overly simplistic way Sun decided to figure out where a user's home directory was, doesn't work. If you look at the JDK bug reports, that bug's been outstanding for six years.

The bug itself is irrelevant, but the notion that "we've got an API for that," and that we just trust that API in all cases, breaks down. The question is, what do you do then?

## The Machine Beneath the VM

Java gives you an open environment, and you can, if you have to, go down to the JNI level and call the native API if you need to go around a Java API issue. But then all the nice things about writing once and deploying it anywhere are gone.

Java has grown to become an immeasurable quantity of code. By comparison, the C++ community is just now getting its act together behind C++0x, with standardized generics, standardized lambdas, all the things people have been whining about for the last twenty-five years. It's a slow-moving, deliberate community, for very good reasons: People write C++ applications for mission-critical applications, bomb guiding systems, telecoms switches, contexts where we as consumers don't have patience. In those environments, it's all your problems as a developer—there's almost nothing to help you.

As you move into an environment where you're culturally predicated to expect that everything is going to be done for you, you lose track of the fact that, underneath, there is a real machine. It might be a Windows machine, a Linux machine, it might be a smart device.

The emphasis is back on you to do that digging, that this bit of what I'm expected to trust doesn't actually work, and I have to replace this code with something that works the way I want it to work. That kind of thinking is something people in the native community have been doing all along.

## Resources as Types

This issue comes up especially in the context of managing device-based resources. Whether you're writing an application for an Android phone or for a data center server environment, you're responsible for the physical devices that are being exercised.

When you create something as simple as a file input stream on a server, or a camera object on your Android device, you'd think of those as being memory-managed by the

JVM. You're right, the object is managed. But beneath the covers, it's not: it's still a physical device. As a Java developer, you still have to go through the code pattern of releasing that device so the JVM doesn't run out of resources. To put it bluntly, if you're dumb enough to trust the platform, then you get what you deserve.

On the JVM, finalization isn't serialized: it can't be, because you have a latent GC. That's just the way the environment works. Until you have serialized and synchronous GC, you can't possibly know that what you had allocated is going away at a particular time.

The language patterns and language design don't give you a lot of help to manage those resources. Good coding practices are encouraged, but are not enforced.

In Java, we have the use-and-consume design pattern: You create something, you use it, you intended to consume it, but you also intended to close it.

In a language like Java that has no synchronous wind-down capability, the consumption of the object is obvious, but the release of the object is still at the behest of the consumer. That's kind of backwards, because everything else in the language says, the consumer doesn't have to worry about it. If you think of it from an OO perspective, it also violates the OO principles, because the resource is not completely encapsulated.

In languages with more formalized [types], or in languages that are perhaps less "objecty," you have a design pattern where the provider of the resource is responsible for releasing the resource when the consumer is done with it. That's built into the language, and the library can explicitly manage the life-cycle of the thing that it's handing out.

[In that way], types can definitely help. You might think of a type vertically, and consider a type all the way down to something that's implementing it. If you then use units of measurement around that thing, your compiler can [help you] understand if that thing is really going away. You can also apply units of usage, or prevalent designer patterns. Whatever you want to apply, something has to be applied at some point, if you want to go beyond simply encouraging that sort of resource management.

What do you think of the notion that types can help you improve trust in APIs?

[Post your opinion in the discussion forum.](#)

[Leading-Edge Java](#) | [Discuss](#) | [Screen Friendly Version](#) | [Email](#)

#### Sponsored Links

[Download the OpenSource Flex 4 SDK](#)  
[Get "the feel" of Scala in this one-hour free video introduction to Scala at Parleys.com](#)  
[New Adobe® Flash® Builder™. Download yours today!](#)  
[Free Online Flex Training](#)  
[FREE book – Learn best practices for peer code review](#)

