Dr. Dobb's Journal

October 2012

A Long Look at JVM Languages

Next

ALSO INSIDE

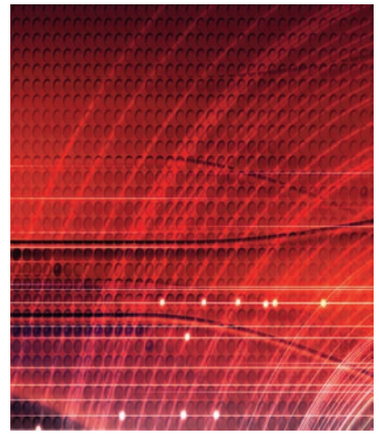
UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

October 2012



COVER STORY

More on DrDobbs.com

Developer Staffing Survey: Needs Outstrip Supply

An *InformationWeek/Dr. Dobb's* survey shows IT managers expect that developers with the needed skills will be hard to find for a while yet.

http://www.drdobbs.com/architecture-and-design/240006708

Microsoft Visual Studio 2012 Reviewed

The new IDE from Microsoft is a must-have tool for Windows 8 development. For developers working on current platforms, it has many attractive new features — but you'll have to abide its peculiar UI.

http://www.drdobbs.com/tools/240007128

When Systematic Testing Isn't Enough

A testing strategy that says, "When all the tests pass, you're done" is not enough for anything beyond trivial programs.

http://www.drdobbs.com/cpp/240007331

Qt 5 Beta: A Developer's Tour

The just-released Qt5 beta shows the library and toolkit thriving and continuing to add capabilities, while simplifying development even more.

http://www.drdobbs.com/mobile/240006976

Unit Testing: Is There Really Any Debate Any Longer?

Continued resistance to unit tests says more about the organization than it does about the practice.

http://www.drdobbs.com/testing/240007176

Mailbag

Processors, unit tests, "Wintendo syndrome," code correctness, and language productivity…readers had a lot to say this month.

Special-Purpose Processors at Intel

In response to our editorial questioning whether fine-grained parallelism will ever gain adoption (http://www.drdobbs.com/parallel/240003926), a former Intel engineer sent this recollection of a related discussion inside the company.

Your editorial brought back memories of my product planning days in the PCI Components Division (PCD) at Intel. We did have a product concept in PCD called the I/O processor. Within the chipset silicon of the time (mid-1990s) there was enough unused die space to slip a 386 core in to handle user-visible I/O. It was very attractive since manufacturing cost would have been free due to Intel hand packing the 386 core for years. The I/O processor proposal ran into 2 objections that killed it, however: (a) It would require a software driver/API effort to allow segmentation of OS/BIOS code/IO drivers between the main x86 CPU (Pentium/P6) and a 386 (eventually 486) I/O processor. Given my computer engineering and software background, I didn't see this as a major hurdle as minicomputer I/O processors are common in mainframe architectures. However, Intel has never succeeded in including software in a business or product cost flow. (b) The internal Intel political push to drive all CPU MIPS and value back into the CPU groups. Chipsets were not allowed to have x86 CPUs or cores. From a technical and business view, the I/O Processor concept was sound. It lost in the software and political areas.

Assuming Microsoft goes ahead with an ARM version of Windows 8, it would be possible to segment OS code, downloading slower/real-time I/O to an ARM processor to create the hybrid system of your dreams. I could also see Apple using an ARM as an I/O processor for improved user experience and leav[ing] the x86 core for crunching.

—Patrick Kearns

Unit Testing

In response to our editorial on unit testing (http://www.drdobbs.com/testing/240007176), we received numerous replies, including the guest editorial in this issue. Also:

I've just read your editorial on unit tests. Despite the fact that I believe that tests are useful, you must bear in mind that there are situations when you cannot use them all:

- Using an event driven environment, sadly the TDD evangelists haven't been able to show us a way to add unit testing to frameworks like Web Forms, their answer is: move to MVC/MVP/MVVM…not an answer that helps to bring Unit Tests to RAD and prototype based frameworks.
- Databases. When unit testing a CRUD application and you have dependencies on the state of the database, the cost of setting database environment can be expensive.

Despite my objections, I defend the use of Tests when dealing with deterministic problems, such as mathematical or financial situations. When you know the input and the expected output, not using tests is suicide, and that that's what I told to my students last year. So when dealing with tests, these considerations are needed.

—Ernesto Cardenas Cangahuala

Andrew Binstock responds: "This is not entirely correct. Unit tests, in the sense most people use the term, do not interact with actual databases. Databases are simulated using mocks, which entirely remove the problems you mentioned."

DBAs Doing the Developers' Dirty Work

In response to the editorial "Longing for Code Correctness" (http://www.drdobbs.com/architecture-and-design/240005803), one reader wrote:

I just finished politely yelling at someone the other day about this issue. I call it the "Wintendo syndrome." It started with Windows 95 (I believe) and Nintendo. At that point in time, we taught a generation that it was OK to have a commercial program FAIL and "all we had to do was reboot" to fix it. Now, we're reaping the rewards of that generation as they code their way into middle age. As an aside, that phrase "all you have to do…" translates into 40-60 person/hrs of your time--not theirs.

To backtrack a second, I started out as a simple COBOL programmer back at the dawn of the online terminal (1980). I quickly realized that Assembler got you where you wanted to go quickly on your own terms. The added benefit of knowing the underlying roots to the Operating System (MVS/VM) helped as well. From there, it was full speed ahead as a Database Administrator. Testing was drilled into my head by every developer who came out of the pre-1980s. Everything was grand until we decided to go "distributed" in the mid 90s. Hiring programmers out of college who knew C and C++ was the fad. That's

where the trouble began and it's only gotten worse over the last 15+ years.

Testing required a "full systems." The cry of we can't test with a small database was met with "OK, give them what they want." So, we started giving programmers full or as-full-as-we-could test databases that pretty much mirrored production. It didn't matter. It only got worse. For the last 15 years, how many DBAs were hired as code control gates for programmers? Talk about wasting money! At one site I consulted, our team of 15 DBAs were in our weekly staff meeting and we literally had 4 programmers waiting for us to move code for them. I later found out one person had requested his module be moved 5 times that morning alone. Can you imagine JPL putting a press release out saying that Curiosity crashed because of improper change control or worse, bad code?

—Roy Niemann
Sr. Oracle DBA Consultant

The Illusions of Bad Programmers

In response to the editorial "What Makes Bad Programmers Different?" (http://www.drdobbs.com/architecture-and-design/240001941), we received this reply, which identifies a trait discussed in the editorial but lacking the precision of the specific term and description:

You're describing in part the Dunning-Kruger effect. Here's how it's described in Wikipedia: "The Dunning-Kruger effect is a cognitive bias in which unskilled individuals suffer from illusory superiority, mistakenly rating their ability much higher than average. This bias is attributed to a metacognitive inability of the unskilled to recognize their mistakes.

Actual competence may weaken self-confidence, as competent in-

dividuals may falsely assume that others have an equivalent understanding. As Kruger and Dunning conclude, 'the miscalibration of the incompetent stems from an error about the self, whereas the miscalibration of the highly competent stems from an error about others.'" (http://is.gd/liQUPr)

This applies to just about any profession — not just IT.

—Ron Klimaszewski

The Productivity of C# vs. Java

In response to the editorial comparing the productivity of programming languages and discussing why C# came in far below Java (http://www.drdobbs.com/jvm/240005881), we received this thoughtful speculation:

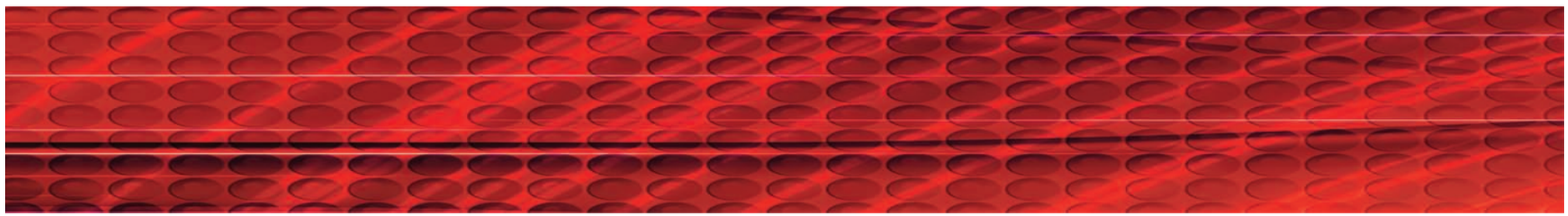
Like you, I too would expect Java and C# to be somewhat similar, but if you excuse me, I will speculate a bit on why C# did not fare as well.

During the last few years, the C# community has been very focused on various patterns on how to produce software, for instance MVVM. Whilst it's interesting, I personally feel MVVM and other patters often taught as being good frequently detract from productivity. Developers apply these patterns without true understanding of the pattern or what you are supposed gain from them, thus not reaping the full rewards. In a project I am working on right now, the previous developers aimed to follow best practice patterns. The end result is lot of models, views, viewmodels, controllers and similar constructs with lot of duplication. Whilst this give a good structure, they didn't seem to resolve the loads of duplicate code (which kills maintainability) or that they set things up so that tasks which are common are also among the most time-consuming (adding a new column in a grid requires changes all over the codebase).

My point is this: I sometimes feel that the C# community is too consumed on implementing software using specific patterns instead of thinking of what the software needs to do. It's a fixation on how the software is made instead of what it should do. This detracts from productivity. Perhaps the Java community, while not uninterested in patterns, is more pragmatic in its approach?

—Mårten Rånge

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.



IN THIS ISSUE

TDD: Is There Really Any Debate Any Longer?

A reader's response to the *Dr. Dobb's* editorial on the universal acceptance of unit testing wonders: Why not TDD, too?

By Joe Eames

This article is in response to *Dr. Dobb's* editor Andrew Binstock's recent editorial about the universality of unit testing (http://is.gd/mn7R97). I think we all can agree that it's universally accepted that unit tests are good. But what about Test-Driven Development (TDD)?

Promoting the practice of TDD in the JavaScript world has occupied much of my recent professional activity. I have created a website (http://testdrivenjs.com) to do that, and I'm currently authoring a course for Pluralsight.com on that very subject. The reason I have done all this work is my very strong belief that TDD as a practice has a considerable value well beyond what most people understand. In fact, I believe that of all the practices made popular by the Agile movement, TDD is the most beneficial overall. Here are eight reasons why I believe that TDD should be beyond debate.

Better Code

Both unit testing and TDD produce better code than not using tests. To unit test a piece of code, you must decouple it from its dependencies so that they can be easily mocked. That process not only helps you write more loosely coupled code, but also encourages you to reduce the number of dependencies a class may have. It will address several code smells such as: too many parameters in methods and constructors, feature envy (a class that uses another class too much), and too many dependencies on other classes.

Beyond that though, the process of TDD causes an incremental approach to building code. You will follow a red (failed unit test)-green (passing unit test)-refactor cycle many times while building a class. That opportunity to frequently refactor your code will encourage you to spend more time refactoring it. This means more time spent renaming identifiers to more intuitive names, more time spent thinking about the readability of your code, more time spent deciding if a given

method is too large or too small, more time thinking about whether a class should be split up into more than one class.

Easier Coding

Because you build a class incrementally in TDD, not in one shot, you have to worry about only one responsibility at a time. You don't have to keep in your head all the things you want the class to do, all the dependencies it needs to work with, all the inputs and outputs that might break the class, all the conditions you need to check for to make sure that arguments are valid. With TDD, you can take that one step at a time. And as you build each case, it often leads you to the next case, and the next one.

Although you can follow the same process without writing your unit tests first, testing after the fact doesn't inherently encourage you to do this.

Solution Triangulation

Because in TDD you incrementally build an algorithm, you can essentially triangulate to the correct solution. The bowling game kata by Robert Martin (http://is.gd/OZW98h) is a fantastic example of this. You can start with an algorithm that solves just a simple case of the problem, then slowly add more cases and refine the solution until it's 100% correct. This makes it much easier to solve a problem than just trying to get it right the first time in one big shot.

Again, test-afterwards doesn't prevent you from following this process, but it doesn't encourage it either. TDD practically forces you to follow this process.

Coding from the Client's Viewpoint

With TDD, you write the test first. A test is a client of an object. It calls the class, consumes its return value, and must know if the process was successful. That process makes you think about the object from the client's point of view, so you will be encouraged to make the object a good abstraction of the responsibilities that the class has. You will make its interface more clean and concise, and the method names will be more explanatory than otherwise.

More Time Thinking Up Front

We all know that too much up-front design is a bad thing, but so is too little. On a micro scale, TDD makes you think more about the design of a particular class. You have to think about what dependencies it needs, what methods it will have, how it will interact with its collaborators, what kinds of values its methods will return. Because you can't write a test until you have some concept of how you will call a method, and what that method will return, and how you will check its correctness, you will be prone to building better classes that are highly cohesive and follow the Single Responsibility Principle (http://is.gd/BVUQGd). This point is closely related to the previous one.

Testing afterwards doesn't encourage this practice. Again, you're free to do it, but as a practice, you aren't encouraged to think about a class because you can just start coding and worry about that stuff as the class gets built.

Better Unit Test Coverage

Although unit testing means writing unit tests to cover your code, there are many variations that code can take. I'm not talking about just the number of branches through code, but all the variations in code. If you take in a string parameter, that string can be null/undefined, empty, or contain a value; it can be overly long; it can contain special

characters. Does your code truly handle all these cases? If you test after, the quantity of tests you write will trend lower because you will simply be verifying current behavior. (See the next point.)

With test-after, your tests don't help you decide what code to write, so therefore, variations on tests become less interesting and less a part of the core process.

With TDD, you can't write code without testing it first, so you can't skip out on the tests regardless of the pressure

Sloth and the Temptation to Skip Unit Tests When Under Pressure

As humans, we are lazy. It's wired into us. If you don't believe this, read *Thinking Fast and Slow* by Daniel Kahneman (http://is.gd/4Czj39). When doing a task, the part of the task where we are most likely to cut corners is the end. We're almost finished; we want to be finished; we want to check it off. So if testing is a distinct activity at the end of our current mini-task, we are far more likely to cut corners and write fewer and less comprehensive tests.

Also, when we test afterwards, because it's at the end of the process, if we are under pressure from our bosses, product managers, or clients, or just our timeline itself, then one of the most tempting corners to cut is the testing. If you've already seen that your code is working correctly, then skipping tests for that code is extremely tempting when the pressure to ship is applied. With TDD, that is impossible. You can't write code without testing it first, so you can't skip out on the tests regardless of the pressure.

No Extra Code

With TDD, the process is to just do the minimum amount of coding to make the test pass. This becomes an art form because you must learn to write appropriate tests that cause you to write sufficient code. When your tests are all passing, and you've cleaned up the code to be as well-factored as you want it, then you're done. The temptation to write more code that handles some possible future requirement (YAGNI; http://is.gd/PXiTkq) is diminished because you're only writing tests to satisfy known requirements of the code.

Testing afterwards doesn't guide how you write your code, with the exception of making it testable. So it gives you no encouragement to write only the code that you need and no more.

TL;DR

In summary, the difference between TDD and simply writing unit tests is big. Don't let the commonalities make you think that you're getting pretty much all the benefits because you write unit tests. If you're not doing TDD, you're missing out on so much more.

— Joe Eames is a front-end Web developer at Domo Inc., an author for Pluralsight.com, and an evangelist of Agile practices. He is a frequent blogger and speaker, the curator of testdrivenjs.com, and a panelist on the JavaScript Jabber podcast (http://javascriptjabber.com/).

Comment

A Long Look at JVM Languages

Exploring the remarkably fertile Java Virtual Machine development platform

By Eric Bruno

For the last decade, the Java Virtual Machine (JVM) has been a popular platform to host languages other than Java. Language developers have continually relied upon the performance and "write-once-run-anywhere" portability of the JVM, as well as other advanced features, to provide the runtime environment for their languages. In fact, porting language interpreters to the JVM became so popular that Sun had hired some high-profile language developers, such as Charles Nutter (of the JRuby language), Ted Leung (Jython), and others to accelerate the porting efforts.

Why have so many languages, including Ruby, Groovy, and Python, been ported to Java? Mainly because it's much easier to target one platform (Java code, in this case) and rely on the multiplatform JVM to host it than it is to write interpreters for each operating system. Additionally, with the JVM's advanced just-in-time (JIT) compilation, the resulting compiled and optimized Java bytecode will typically run with equal or better performance than native interpreters. Further, the Java JIT compiler continues to optimize code well after it's first compiled, depending upon changes in code execution and branching. The cost and effort associated with building this into each independent interpreter implementation makes this kind of performance prohibitive; so, it makes sense to leverage the JVM's existing multiplatform implementation of this performance optimization feature.

Additionally, the JVM brings along other useful features, such as a huge set of well-tested libraries, garbage collection, and excellent built-in tools and debugging interfaces that a language developer can easily plug into.

`invokedynamic`: Bridging the Gap

Although writing language interpreters in Java offers huge benefits in terms of performance and multiplatform support, the biggest win is the interoperability it provides. For instance, Ruby code executing on

the JVM is readily callable from a Java application, and vice versa. As a result, entire libraries of Java code, in source or JAR form, are instantly available to developers building applications in these other languages. Conversely, Java developers can experiment with code written in these other languages as well.

However, prior to Java SE 7 in 2011, there was a small penalty for doing so. These languages weren't first-class citizens, and there was a small performance hit when crossing the barrier to and from Java code, such as when calling Ruby code from a Java application, or Ruby code calling into Java classes in a JAR file. This was mainly because the JVM was not initially built to support dynamically typed languages. To fix this and eliminate the performance penalty, Java Specification Request (JSR) 292 was developed with the goal of creating a new bytecode instruction to resolve the issue.

The result is the `invokedynamic` bytecode, which is an enhancement to the JVM that allows dynamic — and direct — linkage between the calling code and the receiving code at runtime. With Java 7, invoking code in other languages is now equivalent to straight Java method calls. This applies to Java code and any of the languages executing on top of the JVM.

Many of the existing dynamic languages on the JVM are now upgrading their implementations to use `invokedynamic`. Perhaps the most aggressive in this regard is JRuby, which was one of the earliest adopters of the technology and has reaped the rewards. According to the Alioth benchmarks (http://is.gd/PHKBS2), it outperforms the native Ruby 1.9 implementation on many tests.

Oracle, too, is getting in on the `invokedynamic` action — somewhat of a change for a company that has mostly avoided active development of JVM alternatives to Java. At JavaOne 2011, Oracle announced Project Nashorn (German for Rhino), which is an advanced JavaScript engine that's built using `invokedynamic`. Planned for JDK 8, the Nashorn JavaScript engine will work closely with the Oracle Hotspot JVM to provide good performance, and allow seamless Java-to-JavaScript (and reverse) method calls. Plans for JDK 9 include further optimization regarding Java to native calls (JNI), a unified type system, and the meta-object protocol. The goal is to provide a more generic set of rules and descriptors for the code being written, regardless of language, and achieve symbolic freedom (non-Java-specific types) throughout the JVM.

And of course, as improvements are continually made to the JVM runtime performance, JIT optimization, and platform support (including embedded), as they have been with each Java release, all of the dynamic languages built on top of the JVM will benefit as well.

JVM Language Profiles

The following section takes a quick look at the popular dynamic JVM languages, and the languages' implementation details. These languages fall into three broad categories: those that were created to be an improved alternative to Java, those that are ports of other languages, and a miscellaneous category of languages that have other aims. Where relevant, we include links to *Dr. Dobb's* features on these languages.

Let's begin by looking at the JVM languages that were created to extend Java, or to make up for some of the perceived shortcomings in the Java language or platform. These include Groovy, Scala, Gosu, and Kotlin, among others.

Language: Groovy (http://groovy.codehaus.org/)
Project inception: Created by James Strachan, around 2003
Licensing: Apache License, v2.0
Commercial support: Community-based
Binding: Late, reflective
Language type system: Strong, supports static and dynamic typing
Compilation: Bytecode, and JIT compiled
Description: Groovy is an object-oriented programming language much like Java, but meant to be used as a scripting language to the Java platform with a dynamic-language feature set. Features include a more compact, less verbose programming syntax, dynamic typing, native support for DOM-based structures (that is, XML), closures, operator overloading, and many other features that Java does not or will not support. Today, Groovy includes a language branch called Groovy++ (http://www.drdobbs.com/open-source/229500788) that was designed to simplify static typing so as to improve Groovy's performance.

Language: Scala (http://www.scala-lang.org/)
Project inception: Designed by Martin Odersky around 2001; released 2003
Licensing: BSD
Commercial support: Typesafe Inc. / Scala Solutions
Binding: Late, reflective
Language type system: Strong
Compilation: Bytecode, and JIT compiled
Description: Designed to be a better Java, yet built on top of Java, and meant to interoperate with Java code at every level. Odersky built Scala to clean up many of what he perceived as Java's shortcomings. These included a non-unified type system (primitives vs. objects), type erasure (polymorphism), and checked exceptions. Scala brings functional programming concepts to the Java language, such as type inference, anonymous functions (closures), lazy initialization, and many others. The name itself implies a more "scalable" version of Java (even though the language name is pronounced "scah-lah") that is meant to continually improve upon its parent language's shortcomings. Regarding the design of the language and its future, *Dr. Dobb's* interviewed Odersky (http://www.drdobbs.com/architecture-and-design/231001802) recently.

Language: Kotlin (http://confluence.jetbrains.net/display/Kotlin/)
Project inception: Created by JetBrains in 2011
Licensing: Apache, v2.0.
Commercial support: JetBrains Inc.
Binding: Early
Language type system: Strong, static
Compilation: Compiles to JavaScript or Java bytecode, JIT compiled
Description: Kotlin (http://www.drdobbs.com/jvm/232600836) is a codename for a statically typed programming language that compiles to JVM bytecode and JavaScript. The main design goals for Kotlin are: to compile as quickly as Java; be safer than Java (that is, statically check for common pitfalls such as null pointer dereference); to be more concise than Java via local type-inference, closures, extension functions, mixins, and first-class delegation; and to make it way simpler than other, similar, JVM languages.

Language: Gosu (http://gosu-lang.org/)
Project inception: Began as the GScript scripting language in 2002

Licensing: Apache License, v2.0
Commercial support: Community-based; Guidewire Software Inc.
Binding: Early
Language type system: Strong, static
Compilation: Compiles to Java bytecode, JIT compiled
Description: Gosu (http://www.drdobbs.com/open-source/231001429) borrows concepts from other languages, such as Ruby, Java, and C#, and is mainly used as a scripting language within other JVM software systems. However, Gosu has some innovative, very interesting features such as the Open Type System, which allows it to be easily extended for compile-time type checking, and its use of XML and XSL as native types. Its syntax is compact and concise, lending to its simplicity.

Other languages in this group include Fantom (see page 16), which compiles to Java, JavaScript, and .NET, and Ceylon (http://is.gd/fLecZB), which is under development at RedHat.

Ports of Other Languages

The following languages are ports of existing language to the JVM.

Language: JRuby (www.jruby.org)
Project inception: Created by Jan Arne Petersen in 2001
Licensing: Free software via a triple CPL/GPL/LGPL license
Commercial support: Engine Yard Inc.
Binding: Late, reflective
Language type system: Strong, object-oriented
Compilation: Mixed mode — code can be interpreted, JIT compiled, or AOT compiled
Description: JRuby is a Java implementation of the Ruby interpreter, yet tightly integrated with the JVM to allow for efficient and fast calls to and from Java application code. JRuby was a big reason the `invokedynamic` bytecode was added to Java with the Java SE 7 release. This allows all Java-to-Ruby and reverse calls to be treated the same as Java-to-Java calls. Additionally, JRuby inspired Sun's Multi-VM JVM research effort, where one JVM can act as a separate sandbox to each of multiple applications, allowing different versions of the JRuby runtime to be loaded and active at the same time.

Language: Jython (www.jython.org)
Project inception: Created by Jim Hugunin in 1997
Licensing: Permissive free, GPL-like, Python Software Foundation license
Commercial support: Community-based
Binding: Late, reflective
Language type system: Fully dynamic
Compilation: Mixed mode — code can be interpreted, JIT compiled, or AOT compiled
Description: Jython, one of the very first languages ported to the JVM, is a Java implementation of the Python interpreter to enable high-performance, compiled, Python application execution. Written on the JVM, Jython fully supports calls into Java code and libraries, and all code is JIT compiled at runtime. Jython's use of `invokedynamic` is described in a video at http://www.drdobbs.com/jvm/231600435.

Language: Clojure (www.clojure.org)
Project inception: Created by Rich Hickey in 2007
Licensing: Eclipse Public License
Commercial support: Community-based

Binding: Late, reflective
Language type system: Strong, dynamic
Compilation: Bytecode, and JIT compiled
Description: Clojure is a modern-day Lisp for functional programming, interoperable with object-oriented code, and built for concurrency. It's built on and tightly integrated with the JVM, treats all code as data, views functions as objects, and emphasizes recursion over looping. The main feature of Clojure is its immutable state system that provides for much easier, less error-prone, parallel code execution.

Other languages in this group include Fortress (http://is.gd/OctxVb), a now-abandoned port of Fortran; Mirah (http://www.drdobbs.com/jvm/229400307), another Ruby knock-off by the principal developer of JRuby; and NetRexx (http://www.drdobbs.com/jvm/184410534), an IBM-sponsored port of Rexx, which was the first scripting language for Java, but is now mostly inactive.

Miscellaneous Languages

Finally, let's take a quick look at the miscellaneous group, whose most prominent member is Rhino. While technically it is a port of JavaScript, it began life as a standalone interpreter blessed by Sun for scripting within Java apps and is now bundled with the JVM for server-side execution.

Language: Rhino (https://developer.mozilla.org/en-US/docs/Rhino)
Project inception: Created by the Mozilla Foundation in 1997; as part of the Java SE in 2006
Licensing: Mozilla Public License v1.1 / GPL v2.0
Commercial support: Oracle; Mozilla; Community-based
Binding: Late, reflective
Language type system: Dynamic, weak
Compilation: Bytecode, and JIT compiled
Description: Rhino is a JavaScript engine, bundled as part of Java SE, which executes JavaScript code and allows for interoperability with Java code. The engine works in either interpreted or compiled mode, and is meant to execute server-side JavaScript code as part of an enterprise application solution.

Other languages in this group include Adobe's ColdFusion (http://en.wikipedia.org/wiki/ColdFusion), which is the scripting language for the Web application of the same name. It can be compiled to Java bytecodes and run on the JVM.

What is clear from how many of these languages are very actively under development is that the JVM continues to be a remarkably fertile development platform for new languages.

— Eric Bruno is a blogger for **Dr. Dobb's.** *Andrew Binstock also contributed to this article.*

From the Vault

Fantom

The Fantom language, highlighted on drdobbs.com early last year, generates JavaScript, Java bytecodes, and .NET binaries.

— DDJ

By Brian Frank

In 2009, my brother Andy and I started SkyFoundry, a software company focused on analytics for the Internet of Things. One of the many exciting aspects of bootstrapping a software startup is that you begin with a clean slate. So we began to think about what programming language we might use to construct our product, and we found the options wanting. We spent our nights and weekends creating a new programming language, and thus Fantom was born.

I had been building software systems with Java since the 1.0 days, so I was familiar with all Java's strengths and weaknesses. Perhaps Java's strongest asset was the JVM as a mature, bullet-proof runtime. So from the get go, the JVM was our primary target. But we had some guiding principles on where we thought Fantom should break new ground:

- Stick to Java's statically typed object-oriented core, but integrate functional programming and dynamic typing
- Ability to target alternate runtimes such as .NET and JavaScript
- Tackle big problems that plagued Java programs: concurrency bugs and null pointer bugs
- Built-in modularity to aid the construction of large software systems and avoid issues like classpath hell
- Fantom should be licensed and developed as an open source project

What we wanted to keep from Java was the syntax, which was approachable and readable to a wide audience of everyday programmers. Fantom is at its heart a statically typed object-oriented language much like Java or C#. In addition, functional programming is blended

into Fantom with support for first class functions and closures. The whole library is designed to use closures for easy extensibility. For example,

```
// find all the employees who have a salary over $100,000
highPaid = employees.findAll |e| { e.salary > 100_000 }

// sort a list of files by modified time
files.sort |a, b| { a.modified <=> b.modified }
```

In the first example, we pass a closure function to the `findAll` method. This closure is called for each item in the list and returns true or false to match the item. All the matched items are returned as a new list. In the second example, we see how to use a closure to provide a customized sorting comparator. Judicial use of functional programming is a key ingredient in the design of Fantom's easy-to-use APIs.

We love static typing and find it a useful tool in our toolbox. But there are definitely times when a static type system hinders an elegant solution. In Fantom, we wanted the best of both worlds, so we designed two different "call" operators. The "`.`" operator calls a method with static typing and the "`->`" operator calls a method with dynamic typing. Using the "`.`" operator lets you reap all the benefits of static typing: compile-time checks that the method exists and that your parameters are all correct. But if the compiler is getting in your way, just switch to "`->`". This duality between static and dynamic method dispatch lets us have our cake and eat it, too.

Portable Runtime

Java deserves much credit for making "portable programs" a mainstream concept. However, licensing restrictions and the huge surface area of Java's standard library make porting the Java runtime a sticky situation. Witness the current Oracle lawsuit against Google regarding Android to see how Java isn't necessarily the best vehicle for portability. And despite more than a decade of trying, Java never made any traction as a suitable runtime inside web browsers. In Fantom, we deliberately designed the language and the standard library to be portable to alternate runtimes.

Fantom supports three target runtimes: Java VM, .NET CLR, and JavaScript. The JVM runtime is the most mature; Fantom code on the JVM runs with the same performance as most Java code. The .NET runtime is completely functional, although it tends to lag in maturity. The most interesting alternative is the JavaScript runtime. Fantom code can be compiled directly to JavaScript for execution in web browsers. Most of the core standard library is available, too, as JavaScript including Fantom APIs for working with the DOM.

Consider the emerging model of web applications: increasingly, a large percentage of a Web application's codebase is dedicated to front-end JavaScript. It seems likely that in this new decade we will revert to a traditional client/server model of development, with a web application's codebase roughly split evenly between server and client. The only difference from the 90's is that the client will be JavaScript code running in a browser. Writing your client and server code in two different languages is bound to lead to duplication and maintenance issues as projects grow in sophistication. With Fantom you can write your entire codebase in one language. On the server, run your Fantom code using the JVM to maximize performance. On the client, you can leverage Fantom's JavaScript runtime to create powerful HTML5 user interfaces. Data structures, validation, and HTTP messaging code can all be written once using a single codebase. We believe this is the future of web applications, and one of the killer features of Fantom.

Concurrency

Perhaps the biggest quality issue that plagues Java programs relates to concurrency. Java's architecture of shared memory with manual locking is prone to race conditions and deadlocks that befuddle even expert programmers. Concurrency bugs are especially insidious because they often slip through testing, only to be discovered in production systems.

Fantom was designed to address concurrency by making it impossible to share mutable state between threads. Fantom achieves this using a variety of features. The most important concurrency feature comes from Fantom's immutable types, which have compile-time guarantees that once an instance is constructed, it will be deeply immutable for its lifetime. Fantom also comes bundled with its own actor framework. All these features are seamlessly integrated to make Fantom ideal for designing highly concurrent systems. At SkyFoundry, we have built databases, web servers, message queues, and protocol stacks — all completely in Fantom. The result has been high performance, high quality software with none of the headaches and hours spent debugging low-level race conditions and deadlocks. Let's look a simple example of a `const` class:

```
const class Library
{
  const Str name
  const Book[] books
}
```

In the example class, the `const` keyword informs the compiler that the `Library` class should be immutable. Notice that `Library` contains a field called `books`, which references a list of `Book` instances. In Fantom, the compiler and runtime will guarantee that the `books` field is deeply immutable — that the list and what the list contains are also immutable.

Nullability

Beyond concurrency, null pointers are another major source of quality issues — which Fantom directly addresses. Fantom's type system has explicit support for "nullability." If a method signature accepts or returns a type that might be null, then it must be declared directly in code. For example:

```
Int? read()  // might return null
Int readU1() // guaranteed by compiler to never return null
```

Annotating fields and methods with nullability is as easy as sticking a "`?`" on the end of the type. We've found that the process of forcing developers to think about whether a type can be `null` has all sorts of implicit advantages for the design and documentation of developer intent. When you combine this with Fantom's compile-time and runtime enforcement of nullability, most null-pointer bugs are caught much sooner in the development lifecycle than they would be in a language like Java.

Modularity and Packaging

Fantom's concurrency and nullability features are just two examples where we've tried to improve the quality of software at the source code level. But what about the macro level? Much of the cost of a large software system comes after the initial code is written during the deployment and maintenance phases. The ability to assemble large systems from modular software components is fundamental to empowering large teams to build and maintain these systems. Modularity is

baked directly into Fantom's architecture by packaging all code into modules called "pods." Pods are independently versioned and explicitly declare their dependencies on other pods. Fantom's type system and reflection APIs are all based around pods. The ease with which you can reflect pods, types, and slots is a pivotal feature and really allows for some creative designs. For example:

```
// iterate all the installed pods
Pod.list.each |pod|
{
  // iterate all the types packaged by each pod
  pod.types.each |type|
  {
    // if the type is a concrete subclass of AutoStartService
    // then create an instance and start it up
    if (type.fits(AutoStartService#) && !type.isAbstract)
      type.make->start
  }
}
```

Other Goodies

We've covered some of the key features of Fantom, but there are lots of other design decisions and conveniences designed to make programming more fun:

- Easy integration with existing Java or JavaScript code
- Standardized build script and unit testing frameworks
- Support for mixins, which are like Java/C# interfaces, but methods can have implementations
- Built-in human readable serialization syntax, which can be used directly in Fantom source code
- All non-`const` fields automatically have a getter/setter accessor method
- Multi-line strings and string interpolation
- Support for `List`, `Map`, `Type`, `Slot`, `Uri`, and `Duration` literals
- Type inference for local variables
- Method parameters can declare default values

We've also eliminated the need to think about 32-bit versus 64-bit numbers and associated overflow problems — in Fantom all numbers are just 64-bit.

Tooling, Community, etc.

To learn more about Fantom, the first thing to do is head over to http://fantom.org/. This site is the home base for the Fantom community where you can find: online documentation, examples, a forum, links to source and other downloads, plus an IRC channel.

You will find the Fantom community active and helpful. Fantom has two active projects for tooling: Xored, a company based out of Russia, has developed F4, which is an Eclipse IDE for Fantom. And Thibaut Colar is actively developing FantomIDE, a NetBeans-based IDE.

Fantom is a language developed for software engineers by software engineers. We didn't create Fantom as an academic experiment to publish research papers. We take pride in the fact that Fantom is a "boring language" — a workhorse, everyday language to build large software systems. Fantom is easy to learn and bundled with all the tools a typical project needs: module system, build system, testing framework, web server, desktop UI framework, and web UI framework all packaged up into simple, easy-to-use APIs. Enjoy!

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

VISUAL STUDIO 2012 FOR ASP.NET DEVELOPERS

The new version of Visual Studio includes many small touches that facilitate ASP development, particularly in simplifying REST programming and in dealing with multiple form factors on mobile devices that access Web apps.

http://www.drdobbs.com/windows/240007155

LEAVE BREADCRUMBS FOR YOURSELF

By noting where you were when you stopped and what you were planning to do next, you greatly facilitate resuming coding work. Knowing what to do with your notes enhances the benefits even more.

http://www.drdobbs.com/240006675

YOU'VE GOT PARALLEL CODE IN MY CHOCOLATE

Clay Breshears' take on a parallel Depth-First Search (DFS) algorithm visits all nodes in a graph and the order of visitation is likely to change from one run to the next. He's recently been told that there are algorithms that require the visitation order of nodes be the same as it is for a serial execution.

http://www.drdobbs.com/parallel/240007344

RESTFUL WEB SERVICE IN GO POWERED BY THE GOOGLE APP ENGINE

In this installment of our series of Go tutorials, Go's support for REST-based Web services and cloud computing make quick work of useful document analysis project.

http://www.drdobbs.com/cloud/240006401

ARE YOU SURE THAT YOUR PROGRAM WORKS?

Everyone who has ever written a nontrivial program knows that humans make mistakes, and no program can ever simply be assumed to work as specified. Where people's opinions differ is in what to do about this state of affairs.

http://www.drdobbs.com/cpp/240006889

IT'S A JAVA EMBEDDED WORLD

While you still may not be able to run Java on iOS devices, or pure Java on Android devices, you can run Java on billions of embedded devices. This includes feature phones (flip phones and others not considered smart phones, of which there are billions in the world), embedded controllers (i.e., RFID readers, kiosks, and so on), and small footprint computers (called "plug" computers).

http://www.drdobbs.com/jvm/240005983

IN THIS ISSUE

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

Jon Erickson Editor in Chief Emeritus , Dr. Dobb's

CONTRIBUTING EDITORS
Scott Ambler
Mike Riley
Herb Sutter

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

Dr.Dobb's Business Contacts

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

EVP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Salvatore Silletti
(212) 600-3327, ssilletti@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, aleemoll@techweb.com

Marketing Manager, Monique Kakegawa
(949) 223-3609, mluttrell@techweb.com

Director, Client Marketing, Michelle Somers
(516) 562-7928, msomers@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
(212) 600-3092, ssorhaindo@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Account Manager, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000

UBM TECHWEB

Paul Miller CEO

John Dennehy CFO

David Michael CIO

Joseph Braue Sr. VP, Light Reading Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, InformationWeek Business Technology Network

John Ecke VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

Martha Schwartz EVP, Group Sales, InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

Fritz Nelson VP, Guest Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

Eric Lundquist VP and Guest Editorial Analyst, InformationWeek Business Technology Network

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic Development and Business Admin.

Marie Myers Sr. VP, Manufacturing