Programming Languages

Dr. Dobb's Journal

April 2013

Ada With Contracts

Support for contract-based programming improves a language already famous for its focus on reliability.

Next

ALSO INSIDE

UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

April 2013

COVER STORY

FEATURES

More on DrDobbs.com

Why Code in C Anymore?

The traditional reasons for preferring C to C++ have been steadily whittled away. Are there any good reasons to still use C?

http://www.drdobbs.com/240149452

Xamarin 2.0 Review

Write Android and iOS Apps in C# from Microsoft Visual Studio.

http://www.drdobbs.com/240150634

Concurrent Programming with Chain Locking

Concurrent access to trees and lists requires carefully managed fine-grained locking. Here's a generic solution in C# that removes many of the typical problems.

http://www.drdobbs.com/240149442

Learning New Languages

If you want to expand your programming horizons, learn a new language. But don't take the path of least resistance — aim for mastery of the advanced features.

http://www.drdobbs.com/240150070

Arms in the Clouds

Would you use a cloud-based IDE?

http://www.drdobbs.com/240150440

Creating and Modifying XML in Java

How to use the Apache Xerces DOM parser to create and modify XML.

http://www.drdobbs.com/240150782

Mailbag

Coding in C, logging, job hunting, and bad programmers

Why Code in C Anymore?

Our editorial, "Why Code in C Anymore?" (www.drdobbs.com/240149452), which discussed the possibly diminishing role for C (versus C++), drew thoughtful letters:

"You didn't mention the most important reason that C is still required is that C++ doesn't define a standard for name mangling, so you have to provide a C interface to your C++ binary library so it can be linked."

— James Sutton

"There are two other scenarios where C is preferable.

Constrained-memory environments: I believe a program in C will generally have a smaller memory footprint than a similar program in C++. In an embedded environment with limited memory, this has two implications. One is simply whether the object code and the runtime data will fit in memory. The other (for systems that do paging) is that you will have fewer page swaps, and thus better performance if more of your program and data fit in real memory. For a conventional computer, these are not so much of an issue.

Security: With a C program, you pretty much know exactly what the computer is doing. With C++, the language runtime is doing a lot of things in the background, and you really have no idea what they are. (Certainly not to the extent of something like Java, of course, but still a lot more than C.) If you are trying to develop code that has no security vulnerabilities, it is much harder when parts of your program are doing things outside of your control and which you can't even see or analyze."

— Todd Arnold,
Senior Technical Staff Member
IBM Cryptographic Coprocessor Development

"I think the simplicity of C is such a benefit, I wouldn't think of using any other tool for 'real' programming. To write code, one needs to understand deeply what you're writing down. It is possible to see right through the C code to the machine code underneath. What's happening to the stack is clear; what unions mean is clear; casting to manage datatype flexibility is clear; `longjmps` are clear; etc.

Of course, C also has more than its share of issues, some of which are eased in the newer and higher level languages. You're a lot more likely to have a bad pointer in C than in C++, for example."

— Gary Knott

Logging Assistance To Debugging

"While I agree with most points in your recent article "Printf != Debugging" (www.drdobbs.com/240148855) , I think you are a little too harsh on the old school `printf` debugging; both approaches are useful.

You set up a little bit of a straw man with the issue of adding and then removing `printf`s. The best practice wraps the executable debugging code, whether it's `printf`s or other debugging logic, with compile time conditionals that can turn it on and off. I personally like a multi-level approach: `DEBUG=0` eliminates debugging code, `DEBUG=1` compiles the minimal subset, and higher values bring more and more detailed debugging.

Here's why inline debugging code is useful: As you said, debugging is hard. Someone said that since debugging is actually harder than writing the code, it follows that it's impossible to successfully debug high-performance programs that already push one's limits as a programmer when writing them.

Debugging implies a hard process of figuring out the relevant invariants, data and control flow, and so on. Placing conditional debug statements in the code results in a reusable infrastructure that is tied in to the application code, and that will pay off in the future. Relying on debuggers is more ad hoc solution — when it's sufficient, it's great and quick and satisfying. But quickly gets tedious if you have to drill down through the same process every time you work on a new bug. Of course, you can remediate this somehow by creating and storing debugging macros, but since they are decoupled from the source, they are not reliable."

— Przemek Klosowski

Andrew Binstock responds: Thanks for your note. I understand the point you're making, but I don't agree with the approach that you suggest, namely "`DEBUG=0` eliminates debugging code, `DEBUG=1` compiles the minimal subset, and higher values bring more and more detailed debugging."

This practice adds lots of clutter and you have to be very good at figuring out ahead of time what data should be shown at what level. Often, I find that I don't use, say, level 3, but almost always use level 4. Now I have level 3 statements throughout my code communicating something that's never used. On large projects, you could be using level 4 for a long time and suddenly discover because of changes in the program, you need to include an item from level 5 in level 4. Unfortunately, you can't just use level 5 because it dumps out lots of other data that just clogs the debugging . So you create level 4.5. And on it goes. Increasingly more convoluted, overlapping debug layers. It becomes a maze.

How To Gain Experience For Job Hunting

"I'm currently a student attending college. I'm about to graduate and, in terms of projects, I don't have anything to put on my résumé. I only just found out about the advantage of having side projects last year, so I've been brainstorming ever since. Do companies (who are hiring) care about whether these side projects are console or GUI applications? Whether these programs are long or short? What sort of advice what you give to students in terms of side projects?

Thanks in advance, I really appreciate it."

— Alle from New Jersey

Andrew Binstock responds: Let me answer the question from a different perspective than you've asked it. Unless you write a project that is a perfect complement to the work you're trying to find, I don't believe doing your own project will help you much. I'm not raining on your parade, but I am trying to be helpful.

If you're doing the side work for the purpose of building up credibility, then your best path forward is to start contributing to an existing open-source project that interests you. For it to be uselful to your employment prospects, it helps for it to be a high-visibility project (such as the ones hosted at Apache). You'll read a lot of code; you'll get feedback from other coders; and the whole experience will enrich you in many, many ways that writing your own app will not do. In addition, you'll have a useful résumé entry. All projects are looking for contributors. Apache is particularly good at explaining how to get on board their projects. Good luck!

Bad Programmers

In response to our editorial about bad programmers, "What Makes Bad Programmers Different?" (www.drdobbs.com/240001941), we received this additional observation:

"Your article mentioned in the subject line was pretty much spot on. One more bullet item that belongs in the list of traits, however, is: lack of knowledge presents no obstacle to taking action. Even if they're aware that they don't understand a problem, they plow ahead anyway.

I have seen the above trait in nearly every bad programmer. When they are confronted by a show-stopper problem that would otherwise prompt a good programmer to stop, investigate, and understand before proceeding, bad programmers simply say, "Let's just move on" and often serious problems are either swept under the rug or plastered over with a superficial work-around.

Your closing statement of using code reviews to make management aware of the cost of bad programmers, however, relies on management that understands coding well enough to see the problems in a review. Often this is not the case, and it's this same management that hires bad programmers in the first place because they're clueless when it comes to conducting a software position interview that effectively separates the wheat from the chaff. "

— Robert Wey

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*.

Ada 2012: Ada With Contracts

The most important new feature in Ada 2012 is support for contract-based programming, which adds more validation of mission-critical code to a language already famous for its focus on reliability.

By Robert Dewar

The most recent version of the Ada standard, known as Ada 2012, brings contract-based programming to a mainstream language. Preconditions, postconditions, type invariants, and subtype predicates allow software developers to specify their programs' intent more clearly, in effect embedding requirements into the source code. This feature makes it easier to review and understand the program, and it facilitates verification by the compiler, static analysis tools, and/or runtime checks.

"New, Improved" or "No, Unproved"?

In the software world, we are used to getting new versions of things all the time. Nearly every software product undergoes frequent revision, and companies dutifully trot out press releases to trumpet the wonderful new features in each version. In practice, these updates are often not that significant or newsworthy.

Programming languages follow a similar revision pattern. For example, a new version of C, referred to as C11 (http://is.gd/WF2Vmw), was released last year. New versions of languages typically have many new features, and to those in the language design business, these always seem like major advances. However, the wonderful improvements often don't make that much difference in practice. Indeed the style of a lot of C coding has not much changed since the Kernighan and Ritchie days 35 years ago. Most C programmers don't use the new features in C99, let alone the ones that recently appeared in C11.

Ada is no exception to the rule of periodic updates and is subject to the same caveat that often these revisions have "nothing earth-shaking" to offer. For example, Ada 2005, the previous version of the standard, introduced many new features that those of us working on the language felt were very important advances, but most Ada programmers stayed with Ada 95. The Ada 95 version of the language, by the way,

upgraded the original Ada 83 standard with significant functionality including full support for OOP. When an undertaking of comparable magnitude was carried out for C, a language with a new name was born.

A Seismic Shift

All of this leads us to Ada 2012 (http://is.gd/founuy), the latest version of the Ada language standard, formally approved by the ISO in December 2012. Is this really an important development or simply a "new version yawn"? I could certainly supply a catalog of new features, and you can find articles that do just that. But Ada 2012 does bring an "earth-shaking" advance the — introduction of contract-based programming, or what Eiffel programmers call "design by contract."

In the context of software, a "contract" is an assertion of some property of a program component reflecting a requirement that the component must meet. This is a fairly general definition; some specific examples, to be covered later, are preconditions and postconditions (which apply to subprograms, that is, functions and procedures), and invariants and predicates (which apply to types).

The concept isn't new. What is new with the inclusion of this facility in Ada is its spread into the broader domains of high-reliability, safety-critical, and security-critical programming. These are areas where Ada has a significant presence. (For example, the newly deployed iFacts Air Traffic Control (http://is.gd/QV76XX) program in England is programmed in SPARK, a high-integrity Ada subset augmented with annotations. Ada is widely used on new aircraft, both military and civilian.)

The notion of a contract is fundamental to improving a program's reliability (that is, giving increased confidence that it correctly meets its requirements). From a broader perspective, the software development process involves defining a set of high-level requirements, deriving low-level requirements, formulating a software architecture/design that meets these requirements, and writing code that implements the design. This process is not necessarily sequential (feedback/iteration is common) and may be more or less formalized.

Contracts help bridge the gap between low-level requirements and the code. They also create formal documentation of how the code relates to these requirements. Two of the most important kinds of contract are a subprogram's preconditions (which must be true on entry) and postconditions (which must be true on exit). Again, these are not unique to Ada. Their usage was well established, particularly in the area of formal proof of correctness. However, major programming languages have been slow to adopt these concepts.

Capturing Preconditions and Postconditions

Preconditions and postconditions are most easily understood in the context of an example. Assume a program has to process arrays of integers, and one required operation is to remove duplicate elements from a given array. In Ada, this entails defining an array type, then a subprogram to perform the removal:

```
type Int_Array is array (Natural range <>) of Integer;
procedure Dedupe (Arr: in out Int_Array; Last : out Natural);
```

The `Int_Array` declaration allows different objects of the type to be created with different bounds; the lower bound can be any non-negative integer (`Natural`). If `Arr` is an object of type `Int_Array`, then its bounds can be queried by the constructs `Arr'First` (lower bound) and `Arr'Last` (upper bound).

Suppose that `Dedupe` needs to meet the following requirements:

- On entry, the parameter `Arr` contains at least one duplicated element.
- On exit, all duplicates (and only the duplicates) have been removed, nothing new has been added, and the parameter `Last` shows the new upper bound.

English comments could be added to the source code to express these requirements, but Ada 2012 supports much more precision through preconditions and postconditions:

```
procedure Dedupe (Arr: in out Int_Array; Last : out Natural) with
Pre => Has_Duplicates(Arr),
Post => not Has_Duplicates( Arr(Arr'First .. Last) )
          and then (for all Item of Arr'Old =>
                     (for some J in Arr'First .. Last =>
                        Item = Arr(J)))
                   -- Only duplicates removed
          and then (for all J in Arr'First .. Last =>
                     (for some Item of Arr'Old =>
                        Item = Arr(J)));
                   -- Nothing new added
```

where the helper function `Has_Duplicates` can be defined as follows:

```
function Has_Duplicates(Arr : Int_Array) return Boolean is
begin
   return (for some I in Arr'First .. Arr'Last-1 =>
            (for some J in I+1 .. Arr'Last => Arr(I)=Arr(J)));
end Has_Duplicates;
```

The pre- and postconditions make use of quantification syntax introduced in Ada 2012. They precisely specify what the `Dedupe` procedure does without constraining how it is implemented. The example also illustrates that the value of a parameter on entry to the subprogram (`Arr'Old`) can be referenced in a postcondition.

If you are just writing comments in English, achieving completeness and precision is a formidable challenge; but that's a critical part of making sure that the code meets low-level requirements. Expressing the requirements as contracts forces precision, and the analysis required may help tease out pre- or postconditions that otherwise might be overlooked (for example, the terms in the postcondition corresponding to "only duplicates removed" and "nothing new added"). Such "forgotten" conditions correspond to very typical bugs.

Using Preconditions and Postconditions

Once you get into the habit of including contracts with subprograms, how do you make use of them? There are three principal styles of usage.

First, they can act simply as a substitute for English language comments to guide a human reader in understanding the code. In this context, they perform the same function as comments, but with the great advantage of unambiguity and much higher precision. At first glance, this may seem equivalent to simply reading the source code of the subprogram to see what it does. However, contracts are different in two crucial respects: The contracts are at a much higher level of abstraction; and most importantly, the contracts specify everything that is needed and nothing more. Several implementations of `Dedupe` are possible; some would leave the order unchanged and some might leave it sorted. If you relied only on the code, you might infer that one of these two possibilities was part of the requirements, but that would be wrong.

The second use of contracts is to enable the compiler to turn them into runtime checks optionally. Testing then does not simply check the

final results of the program, but also verifies on entry and exit from each subprogram that the code is doing what it is supposed to do. A runtime exception (a well-defined concept in Ada) is raised if a precondition or postcondition fails. Typically, such an exception occurrence will precisely identify the test that fails, thereby easing the debugging process. Like all runtime checks, these checks can optionally be removed for the final production build. This is often done in safety-critical applications, such as aircraft software, where the verification process provides sufficient confidence that the checks will not fail. (And in any event, it would not be much help to the pilot to get a message on the screen saying that some precondition has failed!) In this context, the runtime checks are a means to the end of making sure they are not needed. But in other contexts, if the extra inefficiency is not a problem, it may well make sense to leave the checks enabled. Better to have an ATM machine shut down than to get an undetected error and hand out the wrong amount of money.

Finally, contracts can be used as a basis for formal verification. The `Dedupe` procedure in our example includes a set of preconditions and postconditions expressed in quantificational logic; you (or your automated tools) can have a formal understanding of these conditions and of the code that implements the procedure. The process of proving that the code does in fact guarantee that the postconditions hold on exit if the preconditions hold on entry is explicitly defined. Very often in proofs of correctness, the critical issue is knowing what to prove. The systematic use of contracts allows this question to be answered precisely, and also allows the proof to be carried out in a modular fashion. The `Dedupe` procedure can be proved correct in isolation, and then callers must ensure that the preconditions are met. It is also possible to combine testing and proving in the same program, conveniently using the contracts as a boundary between the two. For example, you might prove the `Dedupe` procedure correct, but rely on testing to verify that the preconditions are met on each call. Such a synthesis between formal methods and testing shows great promise (http://is.gd/fokAvV).

Preconditions, Postconditions, and Type Inheritance

Object-oriented programming raises an interesting issue in connection with pre- and postconditions. In a hierarchy of derived types (type derivation is Ada's mechanism for extending a type through inheritance), a parent type's subprogram may be overridden at each level, with each declaration providing its own set of pre- and postconditions. If such a subprogram is invoked on a polymorphic variable (class-wide, in Ada terms) of the root type, what can be assumed about the pre- and postconditions that apply to the call?

It is certainly possible to regard a subprogram's preconditions for a derived type to be independent of those for the parent type (and, likewise, for the postconditions), so that the set that applies at runtime is determined by the actual subprogram being called. However, this approach can be problematic for formal verification, or even for informal analysis of the code, since you don't know at compile time which set of conditions will need to be met. The only preconditions and postconditions visible are the ones defined for the root type, and that should be the extent of the caller's obligation. If a derived type strengthens the root type's precondition or weakens the root type's postcondition, then the call (on an object of such a type) may fail.

One solution, embodied in Eiffel, is to consider that preconditions on derived types weaken the combined precondition to be applied (this basically corresponds to insisting that on any call, only the root type's preconditions need to be met). And the postconditions are correspond-

ingly strengthened (which corresponds to considering that on return, at least the root type's postconditions must hold).

Ada 2012 has taken a similar approach through two additional constructs, `Pre'Class` and `Post'Class`, which automatically weaken the preconditions and strengthen the postconditions. This is done by automatically `or`'ing the ancestor types' preconditions for the given subprogram, and analogously `and`'ing the postconditions.

"Many buffer overruns in C could be avoided if the language supported range constraints."

Other Contracts in Ada 2012

Besides preconditions and postconditions for subprograms, Ada 2012 supports several other kinds of contracts. One category involves predicates on types: conditions that must always be met by values of the type. One of the most important such predicates, ranges on scalar types, has been part of the language since Ada 83:

```
Test_Score : Integer range 0 through 100;
Distance : Float range -100.0 .. 100.0;
```

This capability (sadly missing from C, C++, Java, and many other languages) is extremely important for program understandability and reliability. It is hardly new (it's been part of Pascal for more than 40 years), but for some reason, it did not catch on in many other languages. And the consequences are significant. Many buffer overruns in C could be avoided if the language supported range constraints. An important project at Microsoft is to add such range declarations to the Windows code base (http://is.gd/tbZIt3). Hundreds of thousands of such statements have been added, and have detected a large number of potential buffer overruns.

Range constraints are important but limited in that they can capture only contiguous subranges of a scalar type's value space. Ada 2012 has generalized this concept: It enables the programmer to specify arbitrary subsets of a type's values through subtype predicates.

Ada 2012's subtype predicates come in two forms, `Static_Predicate` and `Dynamic_Predicate`, employed depending on the nature of the expression that defines the predicate. For example:

```
type Month is
  (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

subtype Long_Month is Month with
Static_Predicate => Long_Month in (Jan, Mar, May, Jul, Aug,
Oct, Dec);

subtype Short_Month is Month with
Static_Predicate => Short_Month in (Apr, Jun, Sep, Nov);

subtype Even is Integer with
Dynamic_Predicate => Even rem 2 = 0;
```

The predicate is checked on assignment, analogous to range constraints:

```
L : Long_Month := Apr;       -- Raises Constraint_Error
E : Even := Some_Func(X, Y); -- Check that result is even
```

The static forms allow more compile-time checking. For example, in this case statement:

```
case Month_Val is
   when Long_Month => …
   when Short_Month => …
end case;
```

the compiler will detect that `Feb` is absent. Dynamic predicates cannot be checked at compile time, but violations can still be detected at runtime. In general, predicates require runtime checks; when you assign a `Month` value to a `Short_Month` variable, a runtime check verifies that it has an appropriate value.

Type Invariants

Another form of contract introduced by Ada 2012 is the type invariant. An invariant applies to a "private" type (somewhat analogous to "protected" in C++ and Java), which can be viewed from two perspectives. One is that of a client, or user, of the type, which can only access the type through subprograms supplied by the type and does not have access to its implementation details. The other viewpoint is that of the module (package) in which the private type is defined. This code has full access to the type's implementation details.

A type invariant (for example, a table that is always sorted) ensures that values that are visible to clients are always appropriate, and that subprograms taking the type as parameter can rely on appropriate values being passed. Since the client cannot access the internals of a private type, there is no possibility for the client to corrupt this value. However, inside the implementation of the package defining the type, no such checks apply. This is important in practice because a complex object needs to be consistent after it has been constructed, but not at the intermediate stages as it is being built.

Ada's private type interface provides exactly the partitioning that is needed to meet the two seemingly conflicting requirements: Sometimes, the values must be guaranteed to be consistent; and at other times, no such guarantees apply. As with preconditions and postconditions, predicates and invariants serve three possible purposes. They are formalized comments concerning the allowed values of variables. They can activate optional runtime checks that the required values are in fact present. Finally, they can act as a of proof of correctness, which must show formally that appropriate values are being assigned.

Conclusion

As with all new versions of languages, Ada 2012 has many new features, but its most notable advance is the introduction of contracts, which will change the way that Ada can be used for constructing large-scale, highly reliable software.

For programmers accustomed to pre- and postconditions in other languages, Ada 2012 will seem immediately familiar, although with some interesting new twists made possible by Ada's design. For others, it may require some education and practice. But for all users, the techniques of contract-based programming represent a significant advance.

Note: Students and developers of Free Software can gain access to a Free Software implementation of Ada 2012 at http://libre.adacore.com/. This implementation is available for several targets, including Windows and Mac, as well as the Lego Mindstorm system, where contract-based programming can be put to effective use in robotics projects.

— Robert Dewar is an emeritus professor of computer science at New York University and the president of AdaCore (http://www.adacore.com/).

Comment

The Clojure Philosophy

Simplicity, freedom to focus, empowerment, consistency, and clarity:
Nearly every element of the Clojure programming language is designed to promote these goals.

By Michael Fogus and Chris Houser

Learning a new language generally requires significant investment of thought and effort, and it is only fair that programmers expect each language they consider learning to justify that investment. Clojure was born out of creator Rich Hickey's desire to avoid many of the complications, both inherent and incidental, of managing state using traditional object-oriented techniques. Thanks to a thoughtful design based in rigorous programming language research, coupled with a fervent look toward practicality, Clojure has blossomed into an important programming language playing an undeniably important role in the current state of the art in language design. On one side of the equation, Clojure utilizes Software Transactional Memory (STM), agents, a clear distinction between identity and value types, arbitrary polymorphism, and functional programming to provide an environment conducive to making sense of state in general, and especially in the face of concurrency. On the other side, Clojure shares a close relationship with the Java Virtual Machine, thus allowing prospective developers to avoid the costs of maintaining yet another infrastructure while leveraging existing libraries.

In the grand timeline of programming language history, Clojure is an infant; but its colloquialisms (loosely translated as "best practices" or idioms) are rooted in 50 years of Lisp, as well as 15 years of Java history. (While drawing on the traditions of Lisp and Java, Clojure in many ways stands as a direct challenge to them for change.) Additionally, the enthusiastic community that has exploded since its introduction has cultivated its own set of unique idioms. The idioms of a language help to define succinct representations of more complicated expressions. Although we will certainly cover idiomatic Clojure code, we will also expand into deeper discussions of the "why" of the language itself.

In this article, we discuss the weaknesses in existing languages that Clojure was designed to address, how it provides strength in those areas, and many of the design decisions Clojure embodies. We also look at some of the ways existing languages have influenced Clojure.

The Clojure Way

Let's start slowly.

Clojure is an opinionated language — it doesn't try to cover all paradigms or provide every checklist bullet-point feature. Instead it provides the features needed to solve all kinds of real-world problems the Clojure way. To reap the most benefit from Clojure, you'll want to write your code with the same vision as the language itself. As we walk through the language features, we discuss not just what a feature does, but why it's there and how best to take advantage of it.

But before we get to that, we'll first take a high-level view of some of Clojure's most important philosophical underpinnings. Figure 1 lists some broad goals that Rich Hickey had in mind while designing Clojure and some of the more specific decisions that are built into the language to support these goals.

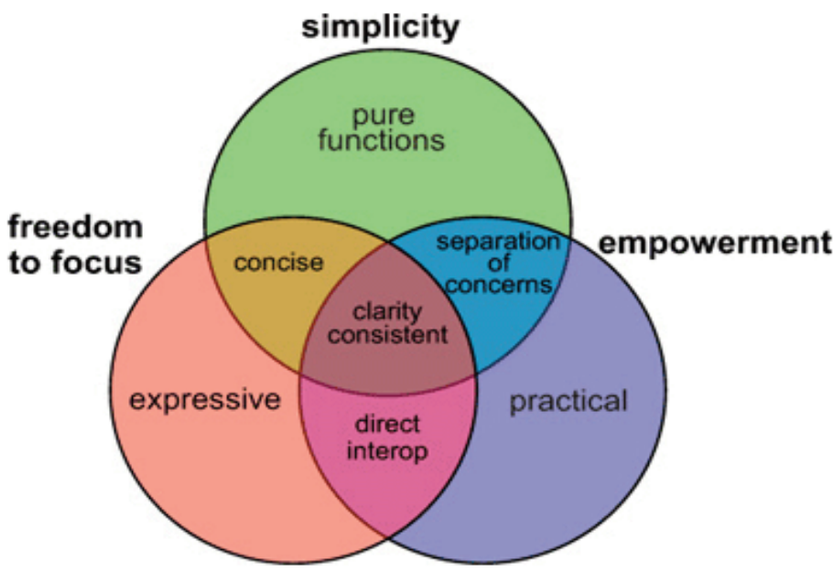

Figure 1: Broad goals of Clojure showing some of the concepts that underlie the Clojure philosophy and how they intersect.

Clojure Philosophy

As Figure 1 illustrates, Clojure's overarching goals are formed from a confluence of supporting goals and functionality, which we will touch on in the following subsections.

Simplicity

It's hard to write simple solutions to complex problems. But every experienced programmer has also stumbled on areas where we've made things more complex than necessary — what you might call incidental complexity as opposed to complexity that's essential to the task at hand (see "Out of the Tar Pit" at http://is.gd/JEgD9Z for details on the concept). Clojure strives to let you tackle complex problems involving a wide variety of data requirements, multiple concurrent threads, independently developed libraries, and so on without adding incidental complexity. It also provides tools reducing what at first glance may seem like essential complexity. The resulting set of features may not always seem simple, especially when they're still unfamiliar, but we think you'll come to see how much complexity Clojure helps strip away.

One example of incidental complexity is the tendency of modern object-oriented languages to require that every piece of runnable code be packaged in layers of class definitions, inheritance, and type declarations. Clojure cuts through all this by championing the pure function, which takes a few arguments and produces a return value based solely on those arguments. An enormous amount of Clojure is

built from such functions, and most applications can be too, which means that there's less to think about when trying to solve the problem at hand.

Freedom to Focus

Writing code is often a constant struggle against distraction, and every time a language requires you to think about syntax, operator precedence, or inheritance hierarchies, it exacerbates the problem. Clojure tries to stay out of your way by keeping things as simple as possible, not requiring you to go through a compile-and-run cycle to explore an idea, not requiring type declarations, and so on. It also gives you tools to mold the language itself so that the vocabulary and grammar available to you fit as well as possible to your problem domain — Clojure is expressive. It packs a punch, allowing you to perform highly complicated tasks succinctly without sacrificing comprehensibility.

One key to delivering this freedom is a commitment to dynamic systems. Almost everything defined in a Clojure program can be redefined, even while the program is running: functions, multimethods, types, type hierarchies, and even Java method implementations. Though redefining things on the fly might be scary on a production system, it opens a world of amazing possibilities in how you think about writing programs. It allows for more experimentation and exploration of unfamiliar APIs, and it adds an element of fun that can sometimes be impeded by more static languages and long compilation cycles.

But Clojure's not just about having fun. The fun is a by-product of giving programmers the power to be more productive than they ever thought imaginable.

Empowerment

Some programming languages have been created primarily to demonstrate some nugget of academia or to explore certain theories of computation. Clojure is not one of these. Rich Hickey has said on numerous occasions that Clojure has value to the degree that it lets you build interesting and useful applications.

To serve this goal, Clojure strives to be practical — a tool for getting the job done. If a decision about some design point in Clojure had to weigh the trade-offs between the practical solution and a clever, fancy, or theoretically pure solution, usually the practical solution won out. Clojure could try to shield you from Java by inserting a comprehensive API between the programmer and the libraries, but this could make the use of third-party Java libraries more clumsy. So Clojure went the other way: direct, wrapper-free, compiles-to-the-same-bytecode access to Java classes and methods. Clojure strings are Java strings; Clojure function calls are Java method calls — it's simple, direct, and practical.

The decision to use the Java Virtual Machine (JVM) itself is a clear example of this practicality. The JVM has some technical weaknesses such as startup time, memory usage, and lack of tail-call optimization (TCO). But it's also an amazingly practical platform — it's mature, fast, and widely deployed. It supports a variety of hardware and operating systems and has a staggering number of libraries and support tools available, all of which Clojure can take advantage of because of this supremely practical decision.

With direct method calls, `proxy`, `gen-class`, `gen-interface`, `reify`, `definterface`, `deftype`, and `defrecord`, Clojure works hard to provide a bevy of interoperability options, all in the name of helping you get your job done. Practicality is important to Clojure, but many other languages are practical as well. You'll start to see some

ways that Clojure really sets itself apart by looking at how it avoids muddles.

Clarity

When beetles battle beetles in a puddle paddle battle and the beetle battle puddle is a puddle in a bottle they call this a tweetle beetle bottle puddle paddle battle muddle. — Dr. Seuss

Consider what might be described as a simple snippet of code in a language like Python:

```
x = [5]
process(x)
x[0] = x[0] + 1
```

After executing this code, what's the value of `x`? If you assume process doesn't change the contents of `x` at all, it should be `[6]`, right? But how can you make that assumption? Without knowing exactly what process does, and whatever function it calls does, and so on, you can't be sure at all.

Even if you're sure process doesn't change the contents of `x`, add multithreading and now you have another whole set of concerns. What if some other thread changes `x` between the first and third lines? Worse yet, what if something is setting `x` at the moment the third line is doing its assignment — are you sure your platform guarantees an atomic write to that variable, or is it possible that the value will be a corrupted mix of multiple writes? We could continue this thought exercise in hopes of gaining some clarity, but the end result would be the same — what you have ends up not being clear at all, but the opposite: a muddle.

Clojure strives for code clarity by providing tools to ward off several different kinds of muddles. For the one just described, it provides immutable locals and persistent collections, which together eliminate most of the single- and multithreaded issues all at once.

Conflated	Separated
Object with mutable fields	Values *from* identities
Class acts as namespace for methods	Function namespaces *from* type namespaces
Inheritance hierarchy made of classes	Hierarchy of names *from* data and functions
Data and methods bound together lexically	Data objects *from* functions
Method implementations embedded throughout class inheritance chain	Interface declarations *from* function implementations

Table 1: Separation of concerns in Clojure.

You can find yourself in several other kinds of muddles when the language you're using merges unrelated behavior into a single construct. Clojure fights this by being vigilant about separation of concerns. When things start off separated, it clarifies your thinking and allows you to recombine them only when and to the extent that doing so is useful for a particular problem. Table 1 contrasts common approaches that merge concepts together in some other languages with separations of similar concepts in Clojure.

It can be hard at times to tease apart these concepts in our own minds, but accomplishing it can bring remarkable clarity and a sense of power and flexibility that's worth the effort. With all these different concepts at your disposal, it's important that the code and data you work with express this variety in a consistent way.

Consistency

Clojure works to provide consistency in two specific ways: consistency of syntax and of data structures.

Consistency of syntax is about the similarity in form between related concepts. One simple but powerful example of this is the shared syntax of the `for` and `doseq` macros. They don't do the same

thing — for returns a lazy `seq`, whereas `doseq` is for generating side effects — but both support the same mini-language of nested iteration, destructuring, and `:when` and `:while` guards. The similarities stand out when comparing the following examples:

```
(for [x [:a :b], y (range 5) :when (odd? y)] [x y])
;=> ([:a 1] [:a 3] [:b 1] [:b 3])
(doseq [x [:a :b], y (range 5) :when (odd? y)] (prn x y))
; :a 1
; :a 3
; :b 1
; :b 3
;=> nil
```

The value of this similarity is having to learn only one basic syntax for both situations, as well as the ease with which you can convert any particular usage of one form to the other if that becomes necessary.

Likewise, the consistency of data structures is the deliberate design of all of Clojure's persistent collection types to provide interfaces as similar to each other as possible, as well as to make them as broadly useful as possible. This is actually an extension of the classic Lisp "code is data" philosophy. Clojure data structures aren't used just for holding large amounts of application data, but also to hold the expression elements of the application itself. They're used to describe destructuring forms and to provide named options to various built-in functions. Where other object-oriented languages might encourage applications to define multiple incompatible classes to hold different kinds of application data, Clojure encourages the use of compatible map-like objects.

The benefit of this is that the same set of functions designed to work with Clojure data structures can be applied to all these contexts: large data stores, application code, and application data objects. You can use into to build any of these types, `seq` to get a `lazy seq` to walk through them, filter to select elements of any of them that satisfy a particular predicate, and so on. Once you've grown accustomed to having the richness of all these functions available everywhere, dealing with a Java or C++ application's `Person` or `Address` class will feel constraining.

Simplicity, freedom to focus, empowerment, consistency, and clarity. Nearly every element of the Clojure programming language is designed to promote these goals. When writing Clojure code, if you keep in mind the desire to maximize simplicity, empowerment, and the freedom to focus on the real problem at hand, we think you'll find Clojure provides you the tools you need to succeed.

Why A(nother) Lisp?

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems. — Alfred North Whitehead

Go to any open source project hosting site and perform a search for the term "Lisp interpreter." You'll likely get a cyclopean mountain of results from this seemingly innocuous term. The fact of the matter is that the history of computer science is littered with the abandoned husks of Lisp implementations. Well-intentioned Lisps have come and gone and been ridiculed along the way, and still tomorrow the search results will have grown almost without bounds. Bearing in mind this legacy of brutality, why would anyone want to base their brand-new programming language on the Lisp model?

Beauty

Lisp has attracted some of the brightest minds in the history of computer science. But an argument from authority is insufficient, so you shouldn't

judge Lisp on this alone. The real value in the Lisp family of languages can be directly observed through the activity of using it to write applications. The Lisp style is one of expressivity and empowerment, and in many cases outright beauty. Joy awaits the Lisp neophyte. The original Lisp language as defined by John McCarthy in his earth-shattering essay "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" (http://www-formal.stanford.edu/jmc/recursive.html) defined the whole language in terms of only seven functions and two special forms: `atom`, `car`, `cdr`, `cond`, `cons`, `eq`, `quote`, `lambda`, and `label`.

Through the composition of those nine forms, McCarthy was able to describe the whole of computation in a way that takes your breath away. Computer programmers are perpetually in search of beauty, and more often than not, this beauty presents itself in the form of simplicity. Seven functions and two special forms. It doesn't get more beautiful than that.

Extreme Flexibility

Why has Lisp persevered for more than 50 years while countless other languages have come and gone? There are probably complex reasons, but chief among them is likely the fact that Lisp as a language genotype fosters language flexibility in the extreme. Newcomers to Lisp are sometimes unnerved by its pervasive use of parentheses and prefix notation, which is different than non-Lisp programming languages. The regularity of this behavior not only reduces the number of syntax rules you have to remember, but also makes the writing of macros trivial. Let's take a look at one now. It's an example that we'll get working on in a moment:

```
(defn query [max]
  (SELECT [a b c]
```

```
    (FROM X
      (LEFT-JOIN Y :ON (= X.a Y.b)))
    (WHERE (AND (< a 5) (< b ~max)))))
```

We hope some of those words look familiar to you, because this isn't an article on SQL. Regardless, our point here is that Clojure doesn't have SQL support built in. The words `SELECT`, `FROM`, and so forth aren't built-in forms. They're also not regular functions, because if `SELECT` were, then the use of `a`, `b`, and `c` would be an error, because they haven't been defined yet.

So what does it take to define a domain-specific language (DSL) like this in Clojure? Well, it's not production-ready code and doesn't tie into any real database servers; but with just one macro and the three functions shown in Lisiting One, the preceding query returns these handy values:

```
(query 5)
;=> ["SELECT a, b, c FROM X LEFT JOIN Y ON (X.a = Y.b)
        WHERE ((a < 5) AND (b < ?))"
       [5]]
```

Note that some words such as `FROM` and `ON` are taken directly from the input expression, whereas others such as `~max` and `AND` are treated specially. The `max` that was given the value `5` when the query was called is extracted from the literal SQL string and provided in a separate vector, perfect for using in a prepared query in a way that will guard against SQL-injection attacks. The `AND` form was converted from the prefix notation of Clojure to the infix notation required by SQL.

Listing One: A domain-specific language for embedding SQL queries in Clojure.

```
(ns joy.sql
  (:use [clojure.string :as str :only []]))

(defn expand-expr [expr]
  (if (coll? expr)
    (if (= (first expr) 'unquote)
      "?"
      (let [[op & args] expr]
        (str "(" (str/join (str " " op " ")
                           (map expand-expr args)) ")")))
    expr))

(declare expand-clause)

(def clause-map
  {'SELECT (fn [fields & clauses]
             (apply str "SELECT " (str/join ", " fields)
               (map expand-clause clauses)))
   'FROM (fn [table & joins]
           (apply str " FROM " table
            (map expand-clause joins)))
   'LEFT-JOIN (fn [table on expr]
           (str " LEFT JOIN " table
                " ON " (expand-expr expr)))
   'WHERE (fn [expr]
           (str " WHERE " (expand-expr expr)))})

(defn expand-clause [[op & args]]
  (apply (clause-map op) args))

(defmacro SELECT [& args]
  [(expand-clause (cons 'SELECT args))
    (vec (for [n (tree-seq coll? seq args)
               :when (and (coll? n) (= (first n) 'unquote))]
          (second n)))])
```

In Listing One, we first use core string functions. Then we handle unsafe literals. Next, we convert prefix to infix. Then we support each kind of clause. We then call the appropriate converter, and finall, we provide the main entrypoint macro.

But the point here isn't that this is a particularly good SQL DSL — more complete ones are available; see http://gitorious.org/clojureql. Our point is that once you have the skill to easily create a DSL like this, you'll recognize opportunities to define your own that solve much narrower, application-specific problems than SQL does. Whether it's a query language for an unusual non-SQL datastore, a way to express functions in some obscure math discipline, or some other application we as authors can't imagine, having the flexibility to extend the base language like this, without losing access to any of the language's own features, is a game-changer.

Although we shouldn't get into too much detail about the implementation, take a brief look back at Listing One and follow along as we discuss important aspects of its implementation.

Reading from the bottom up, you'll notice the main entry point, the `SELECT` macro. This returns a vector of two items — the first is generated by calling `expand-clause`, which returns the converted query string, whereas the second is another vector of expressions marked by `~` in the input. The `~` is known as unquote. Also note the use of `tree-seq` here to succinctly extract items of interest from a tree of values, namely the `input` expression.

The `expand-clause` function takes the first word of a clause, looks it up in the `clause-map`, and calls the appropriate function to do the actual conversion from Clojure s-expression to SQL string. The `clause-map` provides the specific functionality needed for each part of the SQL expression: inserting commas or other SQL syntax, and sometimes recursively calling `expand-clause` when subclauses need to be converted. One of these is the `WHERE` clause, which handles the general conversion of prefix expressions to the `infix` form required by SQL by delegating to the `expand-expr` function.

Overall, the flexibility of Clojure demonstrated in this example comes largely from the fact that macros accept code forms, such as the SQL DSL example we showed, and can treat them as data — walking trees, converting values, and more. This works not only because code can be treated as data, but because in a Clojure program, code is data.

Code Is Data

The notion that "code is data" is difficult to grasp at first. Implementing a programming language where code shares the same footing as its comprising data structures presupposes a fundamental malleability of the language itself. When your language is represented as the inherent data structures, the language itself can manipulate its own structure and behavior. You may have visions of Ouroboros after reading the previous sentence, and that wouldn't be inappropriate, because Lisp can be likened to a self-licking lollypop — more formally defined as homoiconicity. Lisp's homoiconicity takes a great conceptual leap in order to fully grasp, but we'll lead you toward that understanding in hopes that you too will come to realize the inherent power.

Functional Programming

Quick, what does functional programming mean? Wrong answer.

Don't be too discouraged, however — we don't really know the answer either. Functional programming is one of those computing terms that has a nebulous definition. If you ask 100 programmers for their definition, you'll likely receive 100 different answers. Sure, some definitions will be similar, but like snowflakes, no two will be exactly the same. To further muddy the waters, the cognoscenti of computer science will often contradict one another in their own independent definitions. Likewise, the basic structure of any definition of functional pro-

gramming will be different depending on whether your answer comes from someone who favors writing their programs in Haskell, ML, Factor, Unlambda, Ruby, or Qi. How can any person, book, or language claim authority for functional programming? As it turns out, just as the multitudes of unique snowflakes are all made mostly of water, the core of functional programming across all meanings has its core tenets.

A Workable Definition of Functional Programming

Whether your own definition of functional programming hinges on the lambda calculus, monadic I/O, delegates, or java.lang.Runnable, your basic unit of currency is likely to be some form of procedure, function, or method — herein lies the root. Functional programming concerns and facilitates the application and composition of functions. Further, for a language to be considered functional, its notion of function must be first-class. The functions of a language must be able to be stored, passed, and returned just like any other piece of data within that language. It's beyond this core concept that the definitions branch toward infinity, but thankfully, it's enough to start. Of course, we'll also present a further definition of Clojure's style of functional programming that includes such topics as purity, immutability, recursion, laziness, and referential transparency.

The Implications of Functional Programming

Object-oriented programmers and functional programmers will often see and solve a problem in different ways. Whereas an object-oriented mindset will foster the approach of defining an application domain as a set of nouns (classes), the functional mind will see the solution as the composition of verbs (functions). Though both programmers may in all likelihood generate equivalent results, the functional solution will be more succinct, understandable, and reusable. Grand claims indeed! We hope that you'll agree that functional programming fosters elegance in programming. It takes a shift in mindset to start from thinking in nouns to arrive at thinking in verbs, but the journey is worthwhile. In any case, we think there's much that you can take from Clojure to apply to your chosen language — if only you approach the subject with an open mind.

Why Clojure Isn't Especially Object-Oriented

Elegance and familiarity are orthogonal. — Rich Hickey

Clojure was born out of frustration provoked in large part by the complexities of concurrent programming, complicated by the weaknesses of object-oriented programming in facilitating it. This section explores these weaknesses and lays the groundwork for why Clojure is functional and not object-oriented.

Defining Terms

Before we begin, it's useful to define terms. (These terms are also defined and elaborated on in Rich Hickey's presentation, "Are We There Yet?" at http://is.gd/XdB5J4).

The first important term to define is time. Simply put, time refers to the relative moments when events occur. Over time, the properties associated with an entity — both static and changing, singular or composite — will form a concrescence and be logically deemed its identity. It follows from this that at any given time, a snapshot can be taken of an entity's properties defining its state. This notion of state is an immutable one because it's not defined as a mutation in the entity itself, but only as a manifestation of its properties at a given moment in time.

Imagine a child's flip book to understand the terms fully. The book itself represents the identity. Whenever you wish to show a change in the illustration, you draw another picture and add it to the end of your flip book. The act of flipping the pages therefore represents the states over time of the image within. Stopping at any given page and observing the particular picture represents the state of the image at that moment in time.

It's important to note that in the canon of object-oriented programming, there's no clear distinction between state and identity. In other words, these two ideas are conflated into what's commonly referred to as mutable state. The classical object-oriented model allows unrestrained mutation of object properties without a willingness to preserve historical states. Clojure's implementation attempts to draw a clear separation between an object's state and identity as they relate to time. To state the difference to Clojure's model in terms of the aforementioned flip book, the mutable state model is different, so modeling state change with mutation requires that you stock up on erasers. Your book becomes a single page, requiring that in order to model changes, you must physically erase and redraw the parts of the picture requiring change. Using this model, you should see that mutation destroys all notion of time, and state and identity become one.

Immutability lies at the cornerstone of Clojure, and much of the implementation ensures that immutability is supported efficiently. By focusing on immutability, Clojure eliminates entirely the notion of mutable state (which is an oxymoron) and instead expounds that most of what's meant by objects are instead values. Value by definition refers to an object's constant representative amount, magnitude, or epoch (Some entities have no representative value — Pi is an example. But in the realm of computing, where we're ultimately referring to finite things, this is a moot point.). You might ask yourself: What are the implications of the value-based programming semantics of Clojure?

Naturally, by adhering to a strict model of immutability, concurrency suddenly becomes a simpler (although not simple) problem, meaning if you have no fear that an object's state will change, then you can promiscuously share it without fear of concurrent modification. Clojure instead isolates value change to its reference types. Clojure's reference types provide a level of indirection to an identity that can be used to obtain consistent, if not always current, states.

Imperative "Baked In"

Imperative programming is the dominant programming paradigm today. The most unadulterated definition of an imperative programming language is one where a sequence of statements mutates program state. During the writing of this article (and likely for some time beyond), the preferred flavor of imperative programming is the object-oriented style. This fact isn't inherently bad, because there are countless successful software projects built using object-oriented imperative programming techniques. But from the context of concurrent programming, the object-oriented imperative model is self-cannibalizing. By allowing (and even promoting) unrestrained mutation via variables, the imperative model doesn't directly support concurrency. Instead, by allowing a maenadic approach to mutation, there are no guarantees that any variable contains the expected value. Object-oriented programming takes this one step further by aggregating state in object internals. Though individual methods may be thread-safe through locking schemes, there's no way to ensure a consistent object state across multiple method calls without expanding the scope of potentially complex locking scheme(s). Clojure instead focuses on functional

programming, immutability, and the distinction between state, time, and identity. But object-oriented programming isn't a lost cause. In fact, there are many aspects that are conducive to powerful programming practice.

Most of What OOP Gives You, Clojure Provides

It should be made clear that we're not attempting to mark object-oriented programmers as pariahs. Instead, it's important that we identify the shortcomings of object-oriented programming (OOP) if we're ever to improve our craft. In the next few subsections we'll also touch on the powerful aspects of OOP and how they're adopted, and in some cases improved, by Clojure.

Polymorphism is the ability of a function or method to have different definitions depending on the type of the target object. Clojure provides polymorphism via both multimethods and protocols, and both mechanisms are more open and extensible than polymorphism in many languages.

Listing Two: Clojure's polymorphic protocols.

```
(defprotocol Concatenatable
  (cat [this other]))
(extend-type String
  Concatenatable
  (cat [this other]
    (.concat this other)))
(cat "House" " of Leaves")
;=> "House of Leaves"
```

What we've done in Listing Two is to define a protocol named `Concatenatable` that groups one or more functions (in this case only one, cat) that define the set of functions provided. That means the function `cat` will work for any object that fully satisfies the protocol `Concatenatable`. We then extend this protocol to the `String` class and define the specific implementation — a function body that concatenates the argument other onto the string this. We can also extend this protocol to another type:

```
(extend-type java.util.List
  Concatenatable
  (cat [this other]
    (concat this other)))
(cat [1 2 3] [4 5 6])
;=> (1 2 3 4 5 6)
```

So now the protocol has been extended to two different types, `String` and `java.util.List`, and thus the `cat` function can be called with either type as its first argument — the appropriate implementation will be invoked.

Note that `String` was already defined (in this case, by Java itself) before we defined the protocol, and yet we were still able to successfully extend the new protocol to it. This isn't possible in many languages. For example, Java requires that you define all the method names and their groupings (known as interfaces) before you can define a class that implements them, a restriction that's known as the expression problem.

The expression problem refers to the desire to implement an existing set of abstract methods for an existing concrete class without having to change the code that defines either. Object-oriented languages allow you to implement an existing abstract method in a concrete class you control (interface inheritance), but if the concrete class is outside your control, the options for making it implement new or existing abstract methods tend to be sparse. Some dynamic languages such as

Ruby and JavaScript provide partial solutions to this problem by allowing you to add methods to an existing concrete object, a feature sometimes known as monkey-patching.

A Clojure protocol can be extended to any type where it makes sense, even those that were never anticipated by the original implementor of the type or the original designer of the protocol.

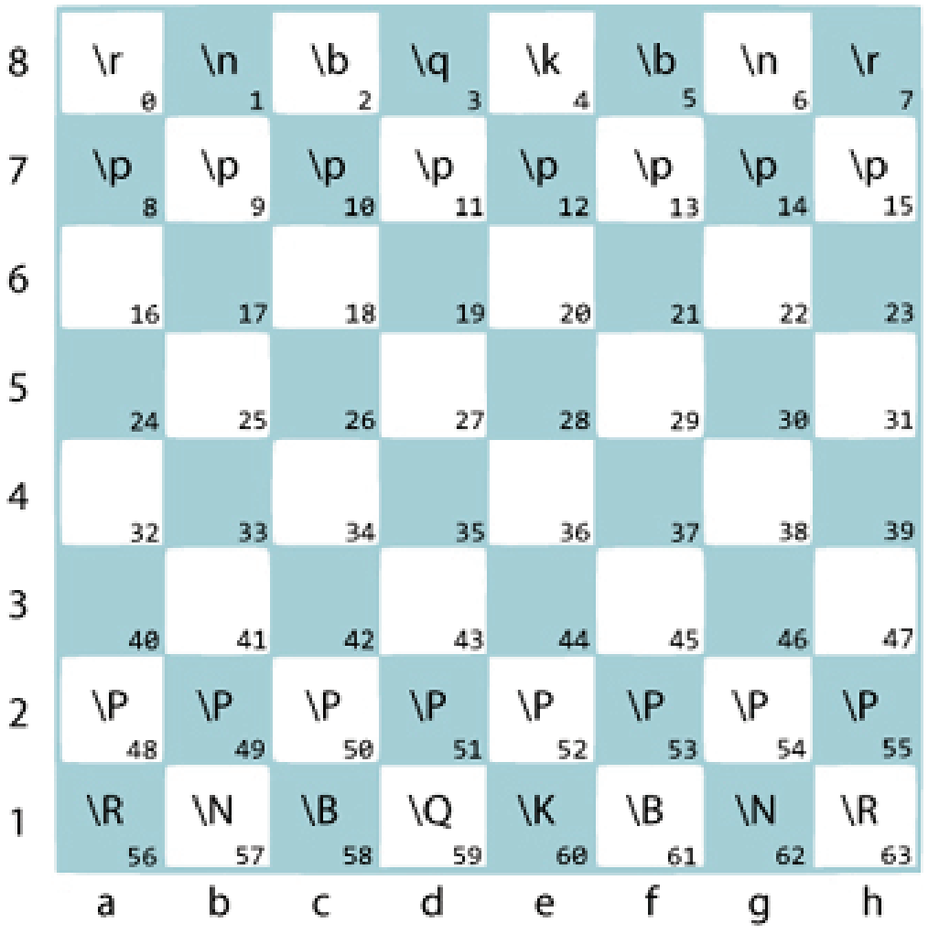

Figure 2: The corresponding chessboard layout.

Clojure provides a form of subtyping by allowing the creation of ad-hoc hierarchies. Likewise, Clojure provides a capability similar to Java's interfaces via its protocol mechanism. By defining a logically grouped set of functions, you can begin to define protocols to which data-type abstractions must adhere. This abstraction-oriented programming model is key in building large-scale applications.

If Clojure isn't oriented around classes, then how does it provide encapsulation? Imagine that you need a simple function that, given a representation of a chessboard and a coordinate, returns a simple representation of the piece at the given square. To keep the implementation as simple as possible, we'll use a vector containing a set of characters corresponding to the colored chess pieces, as shown next.

Listing Three: A simple chessboard representation in Clojure.

```
(ns joy.chess)

(defn initial-board []
  [\r \n \b \q \k \b \n \r
   \p \p \p \p \p \p \p \p
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \- \- \- \- \- \- \- \-
   \P \P \P \P \P \P \P \P
   \R \N \B \Q \K \B \N \R])
```

In Listing Three, line 5 indicates lowercase dark, and line 10 indicates uppercase light. There's no need to complicate matters with the chessboard representation; chess is hard enough. This data structure in the code corresponds directly to an actual chessboard in the starting position, as shown in Figure 2.

From the figure, you can gather that the black pieces are lowercase characters and white pieces are uppercase. This kind of structure is likely

not optimal, but it's a good start. You can ignore the actual implementation details for now and focus on the client interface to query the board for square occupations. This is a perfect opportunity to enforce encapsulation to avoid drowning the client in board implementation details. Fortunately, programming languages with closures automatically support a form of encapsulation to group functions with their supporting data (This form of encapsulation is described as the module pattern. But the module pattern as implemented with JavaScript provides some level of data hiding also, whereas in Clojure — not so much).

The functions in Listing Four are self-evident in their intent (and as a nice bonus, these functions can be generalized to project a 2D structure of any size to a 1D representation — which we leave to you as an exercise) and are encapsulated at the level of the namespace `joy.chess` through the use of the `defn-` macro that creates namespace private functions. The command for using the `lookup` function in this case would be `(joy.chess/lookup (initial-board) "a1")`.

Listing Four: Querying the squares of a chessboard.

```
(def *file-key* \a)
(def *rank-key* \0)

(defn- file-component [file]
  (- (int file) (int *file-key*)))

(defn- rank-component [rank]
  (* 8 (- 8 (- (int rank) (int *rank-key*)))))

(defn- index [file rank]
  (+ (file-component file) (rank-component rank)))

(defn lookup [board pos]
  (let [[file rank] pos]
    (board (index file rank))))
```

On lines 4-5, we calculate the file (horizontal) projection. On lines 7-8, we calculate the rank (vertical) projection. Starting on line 11, we project a 1D layout onto a logical 2D chessboard.

Clojure's namespace encapsulation is the most prevalent form of encapsulation that you'll encounter when exploring idiomatic source code. But the use of lexical closures provides more options for encapsulation: block-level encapsulation (as shown in Listing Five) and local encapsulation, both of which effectively aggregate unimportant details within a smaller scope.

Listing Five: Using block-level encapsulation.

```
(letfn [(index [file rank]
          (let [f (- (int file) (int \a))
                r (* 8 (- 8 (- (int rank) (int \0))))]
            (+ f r)))]
  (defn lookup [board pos]
    (let [[file rank] pos]
      (board (index file rank)))))
```

It's often a good idea to aggregate relevant data, functions, and macros at their most specific scope. You'd still call `lookup` as before, but now the ancillary functions aren't readily visible to the larger enclosing scope — in this case, the namespace `joy.chess`. In the preceding code, we've taken the `file-component` and `rank-component` functions and the `*file-key*` and `*rank-key*` values out of the namespace proper and rolled them into a block-level `index` function defined with the body of the `letfn` macro. Within this body, we then define the `lookup` function, thus limiting the client exposure to the chessboard API and hiding the implementation specific functions and forms. But we can further limit the scope of the encapsulation, as shown in Listing Six, by shrinking the scope even more to a truly function-local context.

Listing Six: Local encapsulation.

```
(defn lookup2 [board pos]
  (let [[file rank] (map int pos)
        [fc rc]     (map int [\a \0])
        f (- file fc)
        r (* 8 (- 8 (- rank rc)))
        index (+ f r)]
    (board index)))
```

Finally, we've now pulled all of the implementation-specific details into the body of the `lookup2` function itself. This localizes the scope of the `index` function and all auxiliary values to only the relevant party — `lookup2`. As a nice bonus, `lookup2` is simple and compact without sacrificing readability. But Clojure eschews the notion of data-hiding encapsulation featured prominently in most object-oriented languages.

Not Everything Is an Object

Finally, another downside to object-oriented programming is the tight coupling between function and data. In fact, the Java programming language forces you to build programs entirely from class hierarchies, restricting all functionality to containing methods in a highly restrictive "Kingdom of Nouns" (http://is.gd/sA82py). This environment is so restrictive that programmers are often forced to turn a blind eye to awkward attachments of inappropriately grouped methods and classes. It's because of the proliferation of this stringent object-centric viewpoint that Java code tends toward being verbose and complex. Clojure functions are data, yet this in no way restricts the decoupling of data and the functions that work upon them. Many of what programmers perceive to be classes are data tables that Clojure provides via maps and records. The final strike against viewing everything as an object is that mathematicians view little (if anything) as objects (see http://is.gd/guKQRa). Instead, mathematics is built on the relationships between one set of elements and another through the application of functions.

Conclusion

We've covered a lot of conceptual ground in this article. If you're still not sure what to make of Clojure, it's okay — we understand that it may be a lot to take in all at once. Understanding will come gradually. For those of you coming from a functional programming background, you'll likely have recognized much of the discussion, but perhaps with some surprising twists. Conversely, if your background is more rooted in object-oriented programming, then you may get the feeling that Clojure is very different than what you're accustomed to. Though in many ways this is true, Clojure does elegantly solve many of the problems that you deal with on a daily basis. Clojure approaches solving software problems from a different angle than classical object-oriented techniques, but it does so having been motivated by their fundamental strengths and shortcomings. With this conceptual underpinning in place, we encourage you to explore Clojure further.

— Michael Fogus is a software developer with experience in distributed simulation, machine vision, and expert systems construction. He's actively involved in the Clojure and Scala communities. Chris Houser is a primary contributor to Clojure and has implemented several features for the language. This article was adapted from their book **The Joy of Clojure: Thinking the Clojure Way** *(http://is.gd/qmeOhv).*

Comment

NetRexx: The Original JVM Scripting Language Returns

The first scripting language for the JVM was recently open-sourced by IBM. After a major update, it is once again finding its place by leveraging its unique benefits.

By René Jansen

NetRexx is an alternative language for the Java Virtual Machine (JVM). One of its most compelling benefits is its complete, syntax-less integration with JVM libraries. Whereas other JVM languages depend on new libraries to be written — with all the related problems like inventing new conventions — NetRexx integrates seamlessly with Java.

The NetRexx language processor can function as a pure scripting language and interpret source code, or it can compile to Java class files. In the compiled version, the right invocation is generated and there is no difference between using a NetRexx class or a Java class. NetRexx takes this integration to the extreme, by depending wholly on the JVM for I/O and other functions.

The benefit of such tight integration is that new developments in IT are available to NetRexx programmers as soon as they are available to the Java developers.

For example, when the Raspberry Pi (http://is.gd/vqt5G6) first emerged — at least, when we could get our hands on one — a whole new world of turning on and off of LEDs and relays via GPIO pins was open to NetRexx. From the beginning, there were two class libraries that could perform the low-level I/O operations, such as with this:

```
import framboos.
  led = Outpin(0)
  led.setValue(1)
  Thread.currentThread().sleep(10000)
  led.setValue(0)
  led.close()
```

Note that this is the complete program, using the scripting mode of NetRexx, in which there is no need to set up class definitions, class constructors, or a `main()` method. This program will turn on an LED, wait 10 seconds, and turn it off.

While coding, you can use an edit-and-interpret cycle for rapid development; when the code is debugged and working, it can be compiled to a class file and stored in a JAR along with the other application classes.

A slightly larger example (Listing One) shows the use of NetRexx with the Eclipse Paho client for MQTT (http://mqtt.org/), the machine-to-machine messaging protocol for the Internet of Things (which is a good combination with a Raspberry, by the way — the mosquitto message broker does an excellent job offering a complete message broker in a small memory footprint).

Listing One: NetRexx written as a class.

```
import java.sql.Timestamp
import org.eclipse.paho.client.mqttv3.

class Publish implements MqttCallback

  method Publish()
    conOpt    = MqttConnectOptions()
    conOpt.setCleanSession(0)
    tmpDir    = System.getProperty("`java.io.tmpdir"`)
    dataStore = MqttDefaultFilePersistence(tmpDir)
    clientId  = MqttClient.generateClientId()
    topicName = "`/world"`
    payload   = "`hello"`.toString().getBytes()
    qos       = 2

    do
      broker    = "`localhost"`
      port  = "`1883"`
      brokerUrl = "`tcp://"`broker"`:"`port
      client    = MqttClient(brokerUrl,clientId, dataStore)
      client.setCallback(this)
    catch e=mqttException
      say e.getMessage()
      e.printStackTrace()
    end -- do

    client.connect()
          log("`Connected  to  "`brokerUrl"`  with  client  ID
              "`client.getClientId())

    -- Get an instance of the topic
    topic = client.getTopic(topicName)

    message = MqttMessage(payload)
    message.setQos(qos)

    -- Publish the message
    time = Timestamp(System.currentTimeMillis()).toString()
    log('Publishing at: 'time' to topic "`'topicName'"` with qos 'qos)
    token = topic.publish(message)

    -- Wait until the message has been delivered to the server
    token.waitForCompletion()

    -- Disconnect the client
    client.disconnect()
    log("`Disconnected"`)

method log(line)
  say line

method messageArrived(t=MqttTopic,m=MqttMessage)

method deliveryComplete(t=MqttDeliveryToken)
  log("`Delivery Complete: "` t)

method connectionLost(t=Throwable)
  log("`Connection Lost:"` t.getMessage())

method main(args=String[]) static
  Publish()
```

For the `log()` method in Listing One, we would choose an Apache log4j implementation in production code.

The point here is, we can stand on the shoulders of others. The more "componentized" these libraries are (for example, using JavaBeans signatures and events), the easier the use and integration is. On the subject of JavaBeans, the NetRexx `properties indirect` statement generates getter and setter methods for all these properties — something that Java developers depend on an IDE to generate or must write by hand. (While many NetRexx developers use editors, there is a NetRexx Eclipse mode (http://is.gd/6aXqjy) that fills in the need for and IDE, and JEdit also has a very usable NetRexx mode (http://is.gd/cTGMEh).)

The clear separation between the language and library layers makes it possible to have a lot of options for user interface technology. All user interface technology that works with the JVM also works from NetRexx (including AWT, Swing, or JavaFX), as well as server-based technology (such as servlets, Struts, JSP, or JSF). Some years ago, I even wrote a Web front-end with Flex with a Tomcat and NetRexx server back-end: Just think about the burning hoops you'd have to jump through to do that in another language.

Syntax

When looking at a NetRexx program, one of the most striking things is the lack of ceremony and punctuation. Its designer, Mike Cowlishaw, is a known opponent of superfluous punctuation. So the repetitive declaration style that plagues C, C++, Java, and JavaScript is absent from NetRexx. There are no curly braces, and semicolons are optional.

Intuitive constructs such as `loop ... end` and `select ... when ... otherwise ... end` require no explanation. In parameter declarations, the local variable comes first, followed by and equal sign and the type of the variable. When the equal sign and type are omitted, they are assumed to be type `Rexx`. A method declares a return type whenever it does not return `Rexx`. This makes for a very relaxed definition of methods. The equal sign can also be used to compare a variable to a type; once you are used to that, it is very natural and you just use it as you would in English.

"String comparison is caseless and humanized in the sense that leading and trailing spaces are ignored."

Runtime: The Rexx Data Type

Type `Rexx` (Java integration is such that nothing precludes you from using this from a Java language program) is the historical Rexx invention of having one type for strings and arbitrary precision numbers. It resides in the runtime package netrexx.lang, which is automatically imported for every program so you don't have to think about it (although it requires explicit import if you're calling NetRexx from Java).

String comparison is caseless and humanized in the sense that leading and trailing spaces are ignored. If you want strict comparison, the `==` operator can be used: It is required if you want to write classes that are not dependent on the NetRexx runtime JAR.

New Features in Version 3.02

After JRE-only compilation was introduced in version 3.01 in 2012 (before that, NetRexx required a JDK for compilation), ease-of-use was addressed in version 3.02, which will be posted to netrexx.org around the time this article appears. New is the closer integration between the `Rexx` type and the Java Collections Framework.

Where we had to cast a lot in previous versions, and did not have a natural order for elements of type `Rexx`, these are now recognized and fitted for cast-less operation. We have gone from:

```
class RexxComparator implements Comparator,Serializable
  method RexxComparator
  method compare(i1=Object, i2=Object) returns int
    i = Rexx i1.toString()
    j = Rexx i2.toString()

    if i < j then return -1
    if i > j then return +1
    else return 0

t = TreeMap(RexxComparator())
i = t.keySet.iterator
loop while i.hasNext
    r = Rexx i.next()
end
```

to

```
t = TreeMap()
t.put('foo','bar')
t.put('baz','frob')
i = t.keySet.iterator
loop while i.hasNext
    r = i.next()
end
```

Here, the keys are automatically put in the order of the now built-in comparator, and variable `r` will be of type `Rexx`. So, when getting elements out of the collection, no cast to `Rexx` is necessary.

This is generic handling without the syntax. All `Rexx` type container facilities (the associative "'indexed strings"' functionality) is usable from the elements in a collection, as are the NetRexx string methods.

Also, to ease integration in products that need on-the-fly compilation or interpretation of a string of code, an API was added to the existing, file-based compiler API. It enables program execution of a string containing the source, as in:

```
import org.netrexx.
/* NetRexx compile from string example  */
programstring = "'say 'hello there via NetRexxC'"'
NetRexxC.main("'myprogram"',programstring)
```

New methods `b2d()` and `d2b()` enable conversion of binary strings to decimal, and decimal strings to binary. For example:

```
'01110'.b2d == 14
'10000001'.b2d == 129
'111110000001'.b2d == 3969
'1111111110000001'.b2d == 65409
'1100011011110000'.b2d == 50928

'129'.d2b == 10000001
'129'.d2b(1) == 1
'129'.d2b(8) == 10000001
'127'.d2b(12) == 000001111111
'129'.d2b(16) == 0000000010000001
```

Also, the `Rexx` type can now be subclassed, a long-standing wish of our user base.

Code Examples

NetRexx is written entirely in NetRexx. Other examples can be found at rosettacode.org, which catalogs various algorithms and puzzles with equivalent implementations written in numerous languages. The examples are also included in the NetRexx distribution, as are others that show nearly every aspect of the use of NetRexx in application software, including all the IBM NetRexx Redbook example code (http://is.gd/LUOUpw). I expect these examples will convince you that NetRexx is one of the most original and productive alternatives to Java on the JVM.

— René Vincent Jansen is the current president of the Rexx Language Association (http://www.rexxla.org/); he was a systems programmer for decades and now manages technical teams in infrastructure and application projects.

Comment

From the Vault

Fantom

As part of a series of feature articles on programming languages in 2011, *Dr. Dobb's* highlighted Fantom — a language that 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 '90s 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 more sophisticated. 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 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 spend 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 class above, 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 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 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!

— Brian Frank is the co-designer of Fantom and president of SkyFoundry.

Comment

This Month on DrDobbs.com

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

WHY I USE PERL...AND WILL CONTINUE TO DO SO

Despite Perl's steady decline in usage, its users continue their romance with the language's features.

http://www.drdobbs.com/240148364

PREVENT CROSS-SITE SCRIPTING IN ASP.NET WEB APPS

Cross-site scripting threats can be greatly minimized by proper encoding. On ASP.NET apps, the Microsoft AntiXSS Library is one of the easiest ways to do the encoding correctly.

http://www.drdobbs.com/240148552

GAME OF LIFE — AVOIDING DESERTED AREAS

The default version of the *Game of Life* feels a lot like the search for an elephant in Africa. In each generation the application looks at each grid point in the simulation and counts the number of live neighbor cells to determine the state of the current cell for the next generation. Even those cells that have nothing around them to count have to be processed, just like looking at all those acres of African desert that do not contain an elephant or any other kind of animal.

http://www.drdobbs.com/240150199

GETTING TO 1 TERAFLOP ON THE INTEL PHI

The key to truly high performance with the Phi coprocessor is to express sufficient parallelism and vector capability to fully utilize the device. Here is a timing framework that enables you to measure and optimize performance and push it past 1 teraflop.

http://www.drdobbs.com/240150561

THE NEW SOCKET APIS IN WINDOWS 8

Windows 8 and Windows Server 2012 introduce important extensions to the venerable Microsoft Winsock stack. For some apps, the new APIs are your only option; for others, they might be your best performing option.

http://www.drdobbs.com/240148403

IT'S HARD TO COMPARE FLOATING-POINT NUMBERS

Suppose you have an inheritance hierarchy that lets you represent integers or floating-point numbers. How would you define comparison within your hierarchy? To restate this problem in a language-independent way: How can we compare two numbers, either of which might be integer or floating-point?

http://www.drdobbs.com/240149806

IN THIS ISSUE

Dr. Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Sean Coady Webmaster, Dr. Dobb's
sean.coady@ubm.com

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

DR DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

UBM Tech