

CONTENTS INCLUDE:

- Motivations
- Language Foundations
- REPLs
- Function Catalog
- Interoperability
- The Clojure Ecosystem and more...

Functional Programming with Clojure

Simple Concurrency on the JVM

By Tim Berglund and Matthew McCullough

MOTIVATIONS

It all started with John McCarthy, the year 1958, and a small language named Lisp. Why are we, five decades later, discussing a sudden springing-to-life of incarnations of this programming language predated only by Fortran? It turns out that Lisp never really died. It was merely in hibernation all those years, "pining for the fjords" of the Java Virtual Machine, powerful CPUs, and abundant RAM. That day has arrived. In fact, the multi-core architecture and ever-harder concurrency problems computer science is currently aiming to solve are a perfect fit for a Lisp dialect. Rich Hickey realized this and began work on a language we now know as Clojure (pronounced: *ˈklɔ̃-zhər*).

In just a few short years, the Java Virtual Machine (JVM) has gone from a platform solely running the Java language, to becoming the preferred platform for the execution of hundreds of cutting edge programming languages. Some of these languages, such as Groovy, Scala and Clojure are vying for the community's respect as the most innovative language on the Java Virtual Machine. Let's

explore why Clojure is one of those frontrunners and why it deserves a place in the tool belt of any leading edge JVM developer.

Don't let yourself think Clojure is only for the elite, or for solving a narrow class of programming problems. It is a general-purpose dynamic language of the JVM, appropriate for a broad variety of tasks by any developer willing to learn a few potentially unfamiliar—but accessible—concepts.

Why the strange syntax?

Lisps are often ridiculed as having a plethora of parenthesis, but they serve a very useful purpose. They reduce all ambiguity and require no lengthy set of evaluation precedence rules; deepest parens execute first. For the benefit of the developer, the frequency of parentheses in Clojure were engineered down to the barest minimum Lisp has ever seen.

Clojure's simple syntax rules are a direct benefit of the code being a direct version of the abstract syntax tree (AST). Following the execution of the code becomes a matter of traversing that tree, which is often a simple recursion to the deepest pair of parentheses, then proceeds outward. For example, Figure 1 shows a basic math problem's Clojure syntax and evaluation.

LANGUAGE FOUNDATIONS

Code Is Data

The fact that Clojure's unconventional syntax asks you to type in the AST directly has a surprising implication: namely, that all Clojure code is data. In fact, there is no formal distinction between code and data in Clojure. When code is represented in text files, it exists as a set of nested forms (or S-expressions).

When these forms are parsed by the reader (which is like a part of the compiler), they become Clojure data structures, no different in kind than the data structures you create yourself in Clojure code.

We call this property homoiconicity. In a homoiconic language, code and data are the same kind of thing. This is very different from a language like Java, in which variables and the code that manipulates them live in two separate conceptual spaces.

As a result of Clojure's homoiconicity, every Clojure program is a data structure, and every Clojure data structure can potentially be interpreted as a program.

The data structure that is the program is available for the program to modify at run time. This arguably allows for the most powerful metaprogramming possible in any language.

Data Types...?

Since all Clojure code is a Clojure data structure, we don't

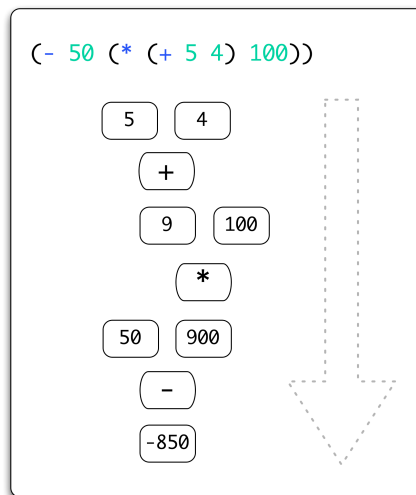


Figure 1: Parsing of a Clojure syntax tree

Get More Refcardz! Visit refcardz.com

www.dzone.com

Getting Started with Clojure

Don't Miss An Issue!
 Get over 90 DZone Refcardz
 FREE from Refcardz.com!

New Release Every Monday

Visit Refcardz.com to get them all Free!

speak of data types and data structures the way we do in a conventional language. Instead, we speak of forms. Forms are text strings processed by the Clojure Reader.

Form Name	Description	Examples
String	A string of characters, implemented by <code>java.lang.String</code> .	"angry monkey", "Mutable state considered harmful"
Number	A numeric literal that evaluates to itself.	6.023E23, 42
Ratio	A rational number.	22/7, 1/3, 24/601
Boolean	A boolean literal form. <code>false</code> and <code>nil</code> evaluate to <code>false</code> ; <code>true</code> and everything else evaluate to <code>true</code> . Also returned by predicate functions.	<code>true</code> , <code>false</code>
Character	A single character literal, implemented by <code>java.lang.Character</code> .	<code>\z</code> , <code>\3</code> , <code>\space</code> , <code>\tab</code> , <code>\newline</code> , <code>\return</code>
Nil	The null value in Clojure.	<code>nil</code>
Keyword	A form beginning with a colon that evaluates to itself. Also a function that looks itself up in a map.	<code>:deposed</code> , <code>:royalty</code>
Symbol	A name that refers to something. Symbols may be function names, data, Java class names, and namespaces.	<code>str-join</code> , <code>java.lang.Thread</code> , <code>clojure.core</code> , <code>+</code> , <code>*source-path*</code>
Set	A collection of unique elements. Also a function that looks up its own elements.	<code>#{:bright :copper :kettles}</code> , <code>#{1 3 5 7 13}</code>
Map	A collection of key/value pairs. Note that commas are optional. Also a function that looks up its own keys.	<code>{:species "monkey" :emotion "angry"}</code> , <code>{"A" 23, "B" 83}</code>
Vector	An ordered collection with high-performance indexing semantics. Also a function that looks up an element by its position.	<code>[1 1 2 3 5 8]</code>
List	An ordered collection, also known as an <i>S-expression</i> , or <i>sexp</i> . The fundamental data structure of Clojure. When the list is not quoted, the first element is interpreted as a function name, and is invoked.	<code>'(13 17 19 23)</code> , <code>(map str [13 17 19 23])</code>

Note that all Clojure data structures are immutable, even things like maps and lists that we normally think of as mutable. Any time you perform an operation on a data structure to change it, you are actually creating a whole new structure in memory that has the modification. If this seems horribly inefficient, don't worry; Clojure represents data structures internally such that it can create modified views of immutable data structures in a performant way.

Mutability in Clojure

Clojure invites you to take a slightly different view of variables

from the imperative languages you are used to. Conceptually, Clojure separates identity from value. An identity is a logical entity that has a stable definition over time, and can be represented by one of the reference types (`ref`, `agent`, and `atom`). An identity "points to" a value, which by contrast is always immutable. We could bind a name to a value as follows—this is analogous to assignment in Java—but there is no idiomatic way to change the value had by that name:

```
=> (def universal-answer 42)
#'user/universal-answer
=> universal-answer
42
=> ; Doing it wrong
=> (def universal-answer 43)
#'user/universal-answer
```

In the example above, 42 is the value bound to the name `universal-answer`. If we wanted the universal answer to be able to change, we might use an `atom`:

```
=> (def universal-answer (atom "what do you get when you multiply six by nine"))
#'user/universal-answer
=> (deref universal-answer)
"what do you get when you multiply six by nine"
```

Note that we access the value of an `atom` using the `deref` function. To change the value pointed to by an `atom`, we must be explicit. For example, to change the universal answer to be a function instead of a number or a string, we use the `reset!` function:

```
=> (reset! universal-answer (fn [] (* 6 9)))
#<user$eval111$fn__12 user$eval111$fn__12@d5e92d7>
=> ((deref universal-answer))
54
```

The double parentheses around the `deref` call are necessary because the value of `universal-answer` is a function. Wrapping that function in parentheses causes Clojure to evaluate it, returning the value 54.

Note that the number, the string, and the function above are values, and do not change. The symbol `universal-answer` is an identity, and changes its value over time.

In traditional concurrent programming, synchronizing access to shared variables is the limiting factor in creating correct programs, and is an intellectually daunting task besides. Clojure provides an elegant solution in its reference types. In addition to `refs`, we have `atoms` and `agents` for concurrently managing mutable state. Together these three types form a significantly improved abstraction over traditional threading and synchronization. You can read more about them here: <http://clojure.org/refs>, <http://clojure.org/atoms>, <http://clojure.org/agents>.

Sequences

Clojure's Aggregate forms (i.e., `String`, `Map`, `Vector`, `List`, and `Set`) can all be interpreted as sequences, or simply "seqs" (pronounced seeks). A `seq` is an immutable collection on which we can perform three basic operations:

- **first**: returns the first item in the sequence

```
=> (first [2 7 1 8 2 8 1 8 2 8 4 5 9 0])
2
```

- **rest**: returns a new sequence containing all elements except the first

```
=> (rest [2 7 1 8 2 8 1 8 2 8 4 5 9 0])
(7 1 8 2 8 1 8 2 8 4 5 9 0)
```

- **cons**: returns a new sequence containing a new element added to the beginning

```
=> (cons 2 [7 1 8 2 8 1 8 2 8 4 5 9 0])
(2 7 1 8 2 8 1 8 2 8 4 5 9 0)
```

These three functions form the backbone of seq functionality, but there is a rich library of additional seq functions in clojure.core.

See here for more details: <http://clojure.org/sequences> and <http://clojuredocs.org/quickref/Clojure%20Core#Collections+-+SequencesSequences>.

Clojure sequences can be infinite, like the set of all positive integers:

```
=> (def positive-integers (iterate inc 1))
```

Since this sequence would take infinite memory to realize, Clojure provides the concept of a lazy sequence. Lazy sequences are only evaluated when they are needed at run time. If we tried to print the sequence of all primes, we would need infinite memory and time:

```
=> (use '[clojure.contrib.lazy-seqs :only (primes)])
=> primes
=> ; requires extreme patience
```

However, we can efficiently reach into that lazy sequence and grab the members we need without constructing the whole thing:

```
=> (use '[clojure.contrib.lazy-seqs :only (primes)])
=> (take 10 (drop 10000 primes))
=> (104743 104759 104761 104773 104779 104789 104801 104803 104827 104831)
```

Of course, lazy sequences are not magical. They will require enough memory and computation to generate the values we request, but they defer that computation until needed, and usually don't attempt eager computation of the entire sequence.

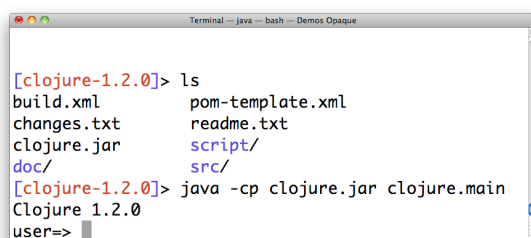
REPLs

Running a REPL

A standard tool for experimenting with Clojure is a Read Eval Print Loop, or REPL. It is an interactive prompt that remembers the results of previous operations and allows you to use those results in future statements.

The simple prerequisites are:

- Java 1.5 or greater JDK
- A download of the `clojure.zip` language archive from <http://clojure.org/downloads>



```
Terminal -- java -- bash -- Demos Opaque

[clojure-1.2.0]> ls
build.xml      pom-template.xml
changes.txt    readme.txt
clojure.jar    script/
doc/           src/
[clojure-1.2.0]> java -cp clojure.jar clojure.main
Clojure 1.2.0
user=>
```

Figure 2: Clojure's basic REPL

After obtaining the prerequisites:

- Unzip the `clojure.zip` archive.
- Run `java -cp clojure.jar clojure.main`

A Web REPL

If getting a REPL up and running seems like too much effort for a first encounter with Clojure, try "Lord of the REPLs", a Google App Engine Web application that immediately lets you try out snippets of syntax. (<http://lotrepls.appspot.com/>)

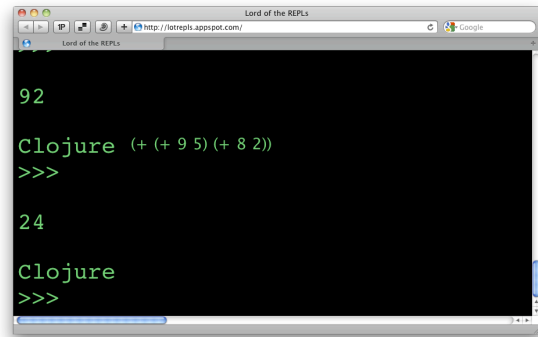


Figure 3: Web Based REPL

Simply hit CTRL + Space and choose Clojure from the language drop-down.

A similar Clojure-only Web REPL can be found at <http://tryclj.licenser.net/>

Documentation

Docstrings

Clojure encourages every function to have a docstring. This is similar to JavaDoc comments in Java. A docstring for a function is enclosed by double quotes and precedes the name of the function.

```
(defn sayhello
  "Introduce a friend by name."
  [friendsname]
  (str "Let me introduce you to my friend " friendsname))
```

The docstring can be viewed with the `doc` function:

```
(doc sayhello)
```

Which outputs:

```
user/sayhello
([friendsname])
 Say hello to a friend by name.
nil
```

If you can't recall what a function's name is, but you can remember some keywords from the docstring, you can search for those words in all functions with the `find-doc` function:

```
(find-doc "friend by name")
```

Which outputs:

```
user/sayhello
([friendsname])
 Say hello to a friend by name.
nil
```

Source code

If the source for a function is available, it can be shown with a straightforward call to `source` like so:

```
(source sayhello)
```

which outputs:

```
(defn sayhello
  "Introduce a friend by name."
  [friendsname]
  (str "Let me introduce you to my friend " friendsname))
nil
```

Note that in some execution environments, it is necessary to first import the repl-utils functions before calling source. Leiningen and the standard Clojure 1.2 repl have the source function available automatically.

```
(use 'clojure.contrib.repl-utils)
```

Tooling

All the major development tools have Clojure plug-ins bringing both syntax highlighting and a REPL to the table.

Emacs

Superior Lisp Interaction Mode for Emacs

<http://github.com/nablaone/slime>

Highlighting and Indentation

<http://github.com/technomancy/clojure-mode>

TextMate

Clojure-TMBundle

<http://github.com/stephenroller/clojure-tmbundle>

Eclipse

Counterclockwise

<http://code.google.com/p/counterclockwise/>

IntelliJ

La-Clojure

<http://plugins.intellij.net/plugin/?id=4050>

NetBeans

Enclojure

<http://www.enclojure.org/>

FUNCTION CATALOG

The sheer volume of functions available in Clojure can be overwhelming. However, a combination of ClojureDocs.org and docstring referencing can aid in finding one that fits a specific need. Here are a few excerpts of frequently used functions:

Arithmetic

Function	Return Value
+	Sum of numbers.
-	Subtraction of numbers.
*	Multiplication of numbers.
/	Division of numbers.
mod	Modulus of numbers.
inc	Input number incremented by one.
dec	Input number decremented by one.
min	Smallest of the provided numbers.
max	Greatest of the provided numbers.

Testing (Predicates)

Function	Description
nil?	True if the parameter is nil.
identical?	True if the parameters are the same object.
zero?	True if zero, false otherwise.
pos?	True if positive, false otherwise.
neg?	True if negative, false otherwise.
even?	True if even, false otherwise.
odd?	True if odd, false otherwise.
min	Smallest of the provided numbers.
max	Greatest of the provided numbers.



An easily-navigable set of core function documentation can be found at: <http://clojuredocs.org/quickref/Clojure%20Core>

INTEROPERABILITY

Many new languages are being developed and hosted on the JVM for its robust deployment model, excellent performance characteristics, and rich ecosystem. These languages take different approaches to interoperating with and embracing the Java tradition they inherit. For its otherwise alien syntax, Clojure has surprisingly clean Java interop, and this is no accident. The decision to host Clojure on the JVM was not merely an implementation detail; rather, the JVM is a first-class part of Clojure's world. Significant language features were added to cooperate cleanly with the JVM. Clojure's Java interop is implemented without intrusive and abstraction-leaking wrappers, but instead by direct programming of Java objects in Clojure code.

Calling a static method on a class is easy:

```
=> (System/nanoTime)
1286079871663966000
=> (Math/E)
2.718281828459045
```

Calling an instance method on a Java object has a special syntax (note that the Clojure String is also a Java object):

```
=> (.toUpperCase "angry monkey")
"ANGRY MONKEY"
=> (.getClass :monkey)
clojure.lang.Keyword
```

Creating an OBJECT can be done with the familiar new function, or with the special dot syntax:

```
=> (new String "monkey")
"monkey"
=> (String. "monkey")
"monkey"
```

Here is some code illustrating how to import classes and interact with the Swing API:

```
(ns swing-example
  (:use [clojure.contrib.swing-utils]
        [clojure.contrib.lazy-seqs :only (fibs)])
  (:import [javax.swing JFrame JLabel JButton JTextField
            SwingConstants]
           [java.awt GridLayout]))

; Define a function that builds a JFrame that runs the supplied
; function
; on a button click, then displays the result in the frame
(defn function-frame [window-title button-text func]
  (let [text-field (JTextField.)
        result-label (JLabel. "-" SwingConstants/CENTER)
        button (doto (JButton. button-text)
                  (add-action-listener
                   (fn [_]
                     (.setText result-label
                               (str (func (Integer. (.getText text-
field))))))))))
        (doto (JFrame. window-title)
              (.setLayout (GridLayout. 3 1))
              (doto (.getContentPane)
                    (.add button)
                    (.add text-field)
                    (.add result-label)
                    )
              (.pack)))]
    ))

; Customize function-frame to calculate Fibonacci numbers
(def fib-frame (function-frame "Fibonacci"
                              "Fib"
                              #(nth (fibs) %)))

; Display the JFrame
(.setVisible fib-frame true)
```

Important Java Interop Functions and Macros

Name	Description	Example
doto	Allows many Java method calls in succession on a single object.	<pre>=>(def user (doto (User.) (.setFirstName "Tim") (.setLastName "Berglund")))</pre>
bean	Creates an immutable map out of the properties of a JavaBean.	<pre>=>(bean user) {:firstName "Tim", :lastName "Berglund"}</pre>
class	Returns the Java class of a Clojure expression.	<pre>=>(class (Date.)) java.util.Date</pre>
supers	Returns a List of the superclasses and implemented interfaces of a Clojure expression.	<pre>=> (supers (class (Calendar/getInstance))) #{java.lang.Cloneable java. util.Calendar java.lang. Comparable java.lang.Object java.io.Serializable}</pre>

Open Source Tools & Frameworks

Leiningen

The JVM has many build tools such as Maven, Gradle and Ant that are compatible with Clojure. However, Clojure has its own build tool that is "designed to not set your hair on fire." Setting it up proves that this goal has been achieved.

On a *nix platform:

```
wget 'http://github.com/technomancy/leiningen/raw/stable/bin/lein'
```

On Windows, download the binary distribution:

<http://github.com/technomancy/leiningen>

Then get help:

```
lein
```

Or create a new Clojure project:

```
lein new myfirstproj
```

Or run a REPL:

```
lein repl
```

Compojure

Clojure has a lightweight Web framework that allows for the

power of Lisp and a corresponding DSL to flex their muscles to produce HTML content with minimal effort.

<http://github.com/weavejester/compojure>

Cascalog

The Hadoop MapReduce framework has garnered enough attention for Internet-scale data processing to warrant a tool called Cascalog. It provides a Clojure language interface to query Hadoop NoSQL datastores.

<http://github.com/nathanmarz/cascalog>

THE CLOJURE ECOSYSTEM

Clojure is more than just an experiment to bring Lisp to the JVM. It is a growing component of production applications, often focusing on exciting new frontiers of parallel, graph navigation, and scalable computing.

Stories abound of Clojure in use at telecommunications firms, analytics agencies, and social media companies. The following short list can be of use in convincing colleagues that the language is worth investigating for applicability to challenging problem domains.

Revelytix: Semantic Emergent Analytics(TM) with semantic technologies and query federation capabilities.

<http://www.revelytix.com/>

Akamai: Live Transcoding of Mobile Content

<http://www.mail-archive.com/clojure@googlegroups.com/msg29948.html>

FlightCaster: Commercial Plane Delay Predictions

<http://www.infoq.com/articles/flightcaster-clojure-rails>

SoniAn: Cloud Powered Email Archiving <http://sonian.com>

GOING FURTHER

Community

Clojure has a vibrant community that happily fields questions and offers up a plethora of code examples.

Official Sites

Homepage

<http://clojure.org/>

Source Code

<http://github.com/clojure/clojure/>

Blog

<http://clojure.blogspot.com/>

Support, Development, Mentoring

<http://clojure.com/>

Documentation, Tutorials

Community-authored code examples

<http://clojuredocs.org>

<http://www.gettingclojure.com/cookbook:clojure-cookbook>

Community-authored book

http://en.wikibooks.org/wiki/Clojure_Programming

Relevance Clojure Studio Training Materials

<http://github.com/relevance/labrepl>

Videos

Rich Hickey on Clojure Concepts
<http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey>

Stu Halloway on Clojure Protocols
<http://vimeo.com/11236603>

Rich Hickey on Clojure at W. Mass Developers' Group
<http://blip.tv/file/812787>

Libraries

Build Tool
<http://github.com/technomancy/leiningen>

Statistical Computing
<http://incanter.org/>

Supplementary Libs
<http://github.com/clojure/clojure-contrib/>

Web Framework
<http://github.com/weavejester/compojure/wiki>

Books

Programming Clojure by Stu Halloway
<http://pragprog.com/titles/shcloj/programming-clojure>

Joy of Clojure by Michael Fogus and Chris Houser
<http://joyofclojure.com/>

Lisp in small pieces by Christian Queinnec
<http://pagesperso-systeme.lip6.fr/Christian.Queinnec/WWW/LiSP.html>

SICP In Clojure
<http://sicpinclojure.com/>

Practical Clojure
<http://apress.com/book/view/1430272317>

Clojure in Action
<http://www.manning.com/rathore/>

Bookmarks

Clojure is still in its formative stages and new resources are popping up all the time. One of the best ways to keep up with the additions to its ecosystem is through a hand-crafted list of bookmarks about this new JVM language.

<http://delicious.com/matthew.mccullough/clojure>
<http://delicious.com/tlberglund/clojure>
<http://delicious.com/tag/clojure>

ABOUT THE AUTHOR



Tim Berglund combines a broad perspective on software architecture and team dynamics with a passion for hands-on development. He specializes in web development using the Grails framework, bringing expertise in the browser, database design, and enterprise integration to bear on browser-based solutions. His skill as a teacher and his integrative approach to technology, team, and organizational problem-solving makes him an ideal partner during periods of disruptive technology change in your organization.

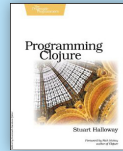
Tim embraces the Java platform, including both the Java language and its high-productivity cousin, Groovy. He also helps clients bring agility to their database development using the Liquibase database refactoring tool, having applied, coached, and lectured on it extensively. He is committed to applying and helping teams excel with agile methods in all of his engagements.

Through his partnership with ThirstyHead.com, Tim offers public and private classroom training in Groovy, Grails, and Liquibase, and is available to develop custom courseware by private engagement. Tim is a frequent speaker at domestic and international conferences, including the Scandinavian Developer Conference, JavaZone, Strange Loop, and the No Fluff Just Stuff tour.



Matthew McCullough is an energetic 15 year veteran of enterprise software development, open source education, and co-founder of Ambient Ideas, LLC, a Denver, Colorado, USA consultancy. Matthew is a published author, open source creator, speaker at over 100 conferences, and author of three of the top 10 Refcardz of all time. He writes frequently on software and presenting at his blog: <http://ambientideas.com/blog>.

RECOMMENDED BOOK



If you're a Java programmer, if you care about concurrency, or if you enjoy working in low-ceremony language such as Ruby or Python, **Programming Clojure** is for you. Clojure is a general-purpose language with direct support for Java, a modern Lisp dialect, and support in both the language and data structures for functional programming. **Programming Clojure** shows you how to write applications that have the beauty and elegance of a good scripting language, the power and reach of the JVM, and a modern, concurrency-safe functional style. Now you can write beautiful code that runs fast and scales well.

BUY NOW

books.dzone.com/books/programming-clojure

Browse our collection of over 100 Free Cheat Sheets



Free PDF

Upcoming Refcardz

- OSMF
- Clojure
- HTML 5
- Test Driven Development



DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
 140 Preston Executive Dr.
 Suite 100
 Cary, NC 27513
 888.678.0399
 919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

