Embracing Common Lisp in the Modern World

[Jan Sulmont]

January 17, 2024

1 Introduction: Clojure, Common Lisp, and the JVM Saga

1.1 The Clojure Equation

- Spoiler alert: Clojure = Subset of Common Lisp + JVM

1.2 A Glimpse into JVM's Legacy

- Exploring the evolution and influence of the Java Virtual Machine

1.3 Common Lisp Unveiled

- Understanding Clojure's roots in Common Lisp's subset

1.4 The Timeless Legacy of Common Lisp

- A language enduring for over 40 years, set to continue for many more

1.5 My Personal Perspective

- Remember: All opinions and rantings here are solely mine

2 Tech Giants vs Microsoft in the Late '90s

2.1 Background

- Era of rapid Internet and software development
- Microsoft dominant in the software industry

2.2 The Alliance: Oracle, Sun Microsystems, IBM

- Formed to challenge Microsoft's growing influence

2.3 The Strategy

- Promoting Java and the JVM
- Aimed to counter Microsoft's .NET framework

2.4 Key Points

- JVM's "Write Once, Run Anywhere" philosophy as a competitive edge
- Collaboration to enhance JVM's capabilities and adoption
- Positioning Java as a versatile, cross-platform solution

2.5 Positive Impact

- Intensified competition in software and web development
- Encouraged open standards and cross-platform compatibility
- Laid groundwork for future enterprise solutions and cloud computing

3 The Grand Vision of JVM

3.1 Universal Platform Ambition

- Envisioned to replace traditional operating systems
- "Write Once, Run Anywhere" extends to entire system operations

3.2 Handling Massive Multitasking

- Designed to efficiently manage tens of thousands of threads...
- Promising unparalleled concurrency and performance

3.3 JVM as the Core of Computing

- Every application, service running within the JVM ecosystem
- Seamless, integrated computing environment

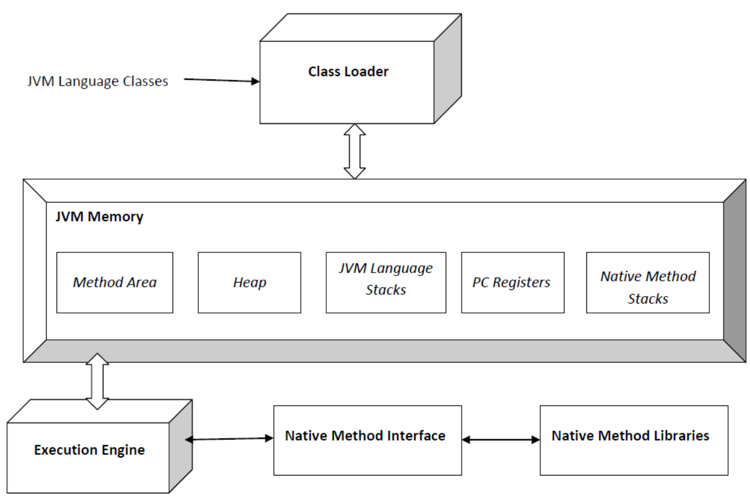
3.4 Revolutionizing System Architecture

- Moving beyond hardware and OS limitations
- Uniform experience across all devices and platforms

3.5 The Utopian Tech Future

- A world where JVM unifies and simplifies computing
- Emphasizing portability, efficiency, and scalability
- An army of Java developers

3.6 1994 Sun Micro System 32 bits



3.7 Whoop-de-doo!

4 Economic Landscape: Late '90s vs 2023

4.1 Late '90s

- Global capitalistic expansion
- Technology and dot-com boom, marked by speculative investments
- Assumption of infinite natural resources: environmental concerns overlooked

- High consumerism and stock market growth: focus on short-term gains
- Set the stage for nowadays challenges

4.2 2023: Escalating Concerns and Shifts in Global Perspective

- Civil unrest fueled by energy insecurities and geopolitical tensions
- Acute realization of finite resources: global policy shifts under pressure
- Intense energy consumption criticized: Bitcoin's PoW seen as an absurdity
- Technology and economy facing a critical juncture for sustainable transformation

5 Evolution of JVM's Vision in the Age of Containers and Clouds

5.1 From JVM as a Universal Platform to Containerization

- Original vision of JVM running on minimal OS overtaken by container technologies

5.2 Rise of Docker and Similar Technologies

- Containers now the building blocks of modern software deployment

5.3 Cloud Computing as the New Paradigm

- Gigantic, modular cloud infrastructures resembling a "Lego set"

5.4 Containers Over JVM

- Shift from JVM-centric to container-centric (Docker, Kubernetes) architectures

5.5 The Irony of Scale

- JVM's goal for universality now encapsulated within even larger cloud ecosystems

6 An Analogy with Consumerism

6.1 Consumerism and Programming Mindset

- Just as responsible consumers question the environmental cost of products, programmers should consider the resource demands of their code

6.2 The Illusion of Unlimited Resources

- Some runtimes offer seemingly unlimited memory and threads
- Similar to consumerist illusions of endless resources

6.3 Environmental Consciousness in Programming

- Recognizing the environmental and computational costs of heavy resource usage

6.4 Sustainable Programming Practices

- Choosing more efficient, resource-conscious programming approaches

6.5 It is our choice:

7 Modern Tech Stack Essentials: Cloud, Containers, Efficiency

7.1 Green Cloud Computing

- Emphasis on sustainable, environmentally-friendly cloud platforms

7.2 Containerization as the Backbone

- Adoption of container technologies (e.g., Docker) for flexible deployment

7.3 Efficiency in Runtime Environments

- Need for lightweight, resource-efficient runtimes within containers
- How does the JVM fits in that picture?
- Rust on the rise; C/C++ still very much in demand.
- Clojure - JVM = Common Lisp

8 Common Lisp Implementations Compiling to Machine Code

8.1 SBCL (Steel Bank Common Lisp)

- High-quality native compiler
- SBCL Official Site

8.2 CCL (Clozure Common Lisp)

- Compiler-only implementation, generates native code
- CCL Official Site

8.3 ECL (Embeddable Common Lisp)

- Compiles to C, capable of generating native code
- ECL Official Site

8.4 CLASP

- Interoperates with C++, uses LLVM for JIT compilation to native code
- CLASP GitHub Repository

8.5 CMUCL

- High-performance implementation from Carnegie Mellon University
- CMUCL Official Site

9 Commercial Common Lisp Environments

9.1 LispWorks

- An integrated cross-platform development tool for Common Lisp
- LispWorks Official Site

9.2 Allegro CL

- Provides the full ANSI Common Lisp standard with many extensions
- Allegro CL Official Site

9.3 MOCL

- Common Lisp as a library for mobile devices and OSX
- MOCL Official Site

10 Common Lisp vs. Clojure: Efficiency in CPU and Memory

10.1 Compiled Code Performance

- CL implementations compile to machine code, often more CPU efficient.
- Especially true for numeric and CPU-intensive tasks.

10.2 Memory Footprint

- CL generally has a smaller memory footprint compared to JVM (Clojure).
- More control over memory management in CL.

10.3 Startup Time

- Faster startup times in CL compared to JVM.

10.4 Garbage Collection

- CL offers more tunable garbage collection strategies.
- JVM's collector optimized for long-running processes but can introduce latency.

10.5 Tail Call Optimization

- CL supports efficient tail recursion in some implementations.
- Clojure has recur, but JVM support varies.

10.6 Data Structure Efficiency

- CL's mutable structures can be more memory-efficient.
- Clojure's immutable structures might have higher overhead in some cases.

10.7 Direct Hardware Access

- CL provides more efficient pathways for direct hardware access and C interoperability.

11 Clojure vs Common Lisp code

11.1 Immutability?

```
(defun merge-hash-tables (ht &rest hts)
  "From 1 or more HTS create a single one with TEST of HT."
  (if hts
```

```
    (let ((rez (make-hash-table :test (hash-table-test ht))))
      (mapc (lambda (next)
              (maphash
               (lambda (key value)
                 (setf (gethash key rez) value))
               next))
            (cons ht hts))
      rez)
    ht))
;; vs
(defun merge-hash-tables! (ht &rest hts)
  "Merge all HTS into HT. Modifies HT in place."
  (mapc (lambda (next)
          (maphash (lambda (key value)
                     (setf (gethash key ht) value))
                   next))
        hts)
  ht)
```

11.2 Multiple dispatch - Clojure

```
;; data structures
(defrecord Circle [radius])
(defrecord Rectangle [width height])
(defrecord ConsoleContext [])
(defrecord GUIContext [])

;; multi-methods
(defmulti draw (fn [shape context] [(class shape) (class context)]))
(defmethod draw [Circle ConsoleContext] [circle console]
  (println (str "Drawing a circle with radius "
                (:radius circle) " on the console.")))
(defmethod draw [Circle GUIContext] [circle gui]
  (println (str "Drawing a circle with radius "
                (:radius circle) " on the GUI.")))
(defmethod draw [Rectangle ConsoleContext] [rectangle console]
  (println (str "Drawing a rectangle with width "
                (:width rectangle) " and height "
                (:height rectangle) " on the console.")))
(defmethod draw [Rectangle GUIContext] [rectangle gui]
```

```
  (println (str "Drawing a rectangle with width "
                (:width rectangle) " and height "
                (:height rectangle) " on the GUI.")))
(let [circle (->Circle 5)
      rectangle (->Rectangle 10 20)
      console (->ConsoleContext)
      gui (->GUIContext)]
  (draw circle console)
  (draw rectangle gui))
```

11.3 Multiple dispatch - Common Lisp

```
;; Define the classes
(defclass shape () ())
(defclass circle (shape)
  ((radius :accessor radius :initarg :radius :initform 0)))
(defclass rectangle (shape)
  ((width :accessor width :initarg :width :initform 0)
   (height :accessor height :initarg :height :initform 0)))
;; Define contexts
(defclass console-context () ())
(defclass gui-context () ())
;; Define the generic function
(defgeneric draw (shape context))
;; Methods for drawing a circle
(defmethod draw ((s circle) (c console-context))
  (format t "Drawing a circle with radius ~A on the console.~%" (radius s)))
(defmethod draw ((s circle) (c gui-context))
  (format t "Drawing a circle with radius ~A on the GUI.~%" (radius s)))
;; Methods for drawing a rectangle
(defmethod draw ((s rectangle) (c console-context))
  (format t "Drawing a rectangle with width ~A and height ~A on the console.~%" (width
(defmethod draw ((s rectangle) (c gui-context))
  (format t "Drawing a rectangle with width ~A and height ~A on the GUI.~%" (width s)
;; Usage
(let ((c (make-instance 'circle :radius 5))
      (r (make-instance 'rectangle :width 10 :height 20))
      (console (make-instance 'console-context))
      (gui (make-instance 'gui-context)))
  (draw c console)
```

```
  (draw r gui))
```

11.4 XTDB - Clojure

```
(let [node (xt.client/start-client "http://localhost:3000")]
    (dotimes [i 99999]
      (let [[xt-id user-id name] (repeatedly #(random-uuid))
            tx-key (xt/submit-tx node [[:put :clojure
                                        {:xt/id xt-id
                                         :user-id user-id
                                         :name name}]])
            res (xt/q node
                      {:find ['x]
                       :where [(list '$ :clojure {:xt/* 'x :xt/id xt-id})]}
                      {:basis {:tx tx-key}
                       :default-all-valid-time? false})]
        (assert (= 1 (count res)))
        (assert (= xt-id (-> (first res) :x :xt/id))))
      (Thread/sleep 5)
      (when (zero? (mod (inc i) 10))
        (println "--> count=" (inc i)))))
```

11.5 XTDB - Common Lisp

```
(let ((node (make-xtdb-http-client "http://localhost:3000")))
  (format t "-->url: ~a  table: ~a ~%" url table)
  (loop
     for count from 1 upto 100000
     do (let* ((xt/id (uuid:make-v4-uuid))
               (tx-key (submit-tx
                         node
                         (vect (vect :|put| table
                                     (dict :|xt/id| xt/id
                                           :|user-id| (uuid:make-v4-uuid)
                                           :|text| "yeayayaya")))))
               (rc (query node
                          (dict
                           :|find|  (vect 'x)
                           :|where| (vect (xtdb/list
                                           '$
```

```
                                  table (dict :|xt/*| 'x
                                              :|xt/id| xt/id))))
                :basis (dict :|tx| tx-key)
                :default-all-valid-time? nil)))
(assert (and (= 1 (length rc))
             (uuid:uuid= xt/id (href (car rc) :|x| :|xt/id|))))
(sleep 0.005)
(when (= 0 (mod count 10))
  (format t "--> count=~a~%" count)))))
```