What I Don’t Like About Java

# Purpose and Scope

Java is a marvel of creative language design, implementatation, and optimization technlogy. I have nearly-unbounded admiration for the Java architects and primary developers for the amazing, pragmatic, and productive ecosystem developed to date.

Having said that, I and many of my colleagues who’ve extensively used Java since JDK 1.0 was first released, have had to learn to live with Java’s limitations. Those of us old-timers who still routinely develop in Java have grown weary with these limitations, especially as we’ve watched those colleagues who’ve moved to alternative ecosystems – most notably Scala and Kotlin, and to a lesser extent, Clojure, have left a significant load of limitations for greener, more sophisticated, and frankly, more productive pastures.

Having used Scala on a couple of substantial projects, and done deep dives on Kotlin and the other JVM languages, while having designed and implemented several Actor languages, and learned a boatload of contemporary patterns, have reached the following firm conclusions:

* Java is at EOL; while Oracle is in for a ton, and will undoubtedly continue to invest in Java evolution, it’s increasingly vulnerable to less crusty, cleaner, and more productive competitors.
* None of the existing languages have bleeding-edge semantics, unifying many of the advances over the last quarter century.
* The time is ripe to create YAPL (Yet Another Programming Language), incorporating such currently-bleed-edge semantics and features.
* Assuming such a language gets even meager traction, there’s a good chance it will follow increasing adoption, rapid evolution driven by market demand, and will inescabably mature into a way more advanced and empowering ecosystems in the next few years.
* As I see it, such a language will encroach on, then dominate existing languages, and bring about a revolution in programming, across all domains.
* Such a language won’t just be an end in itself, but will power entirely new computational paradigms, and give rise to a virtual explosion of new tools, systems, and end-user experiences, today the stuff of techno-dreams, and the foundation of new work modes and environment, industries, economies, and just maybe even new forms of government.

In this document, I present a long backlog of issues and critiques of the current state of the java ecosystem, based on experience with JDK 11. The issues raised run the gamut between the mere irritations to large faults, functional short-falls, and gross limitations that all Java devs currently face.

They also run another, orthogonal gamut of features, Java syntax, and semantics that are out-of-reach, most of them unlikely to ever be added to Java in 100 or more years. In all candor, each of these issues can be viewed as significant, powerful language features, which can be easily interpreted as something between an informal, high-level language specification, and a wish-list.

# Meta Processes

* would seem a useful idea to create a UX around interacting with tradeoffs, especially pro/con or a/b selectors

# Issues

## Evolution Blocked by Backward Compatibility

* Sun, a later Oracle, went to extremes to maintain backward compatibility. While I and other observers stand in some amount of awe at the technical creativity involved, I and I believe the majority of the Java user community, if polled whether to maintain or break backward compatibililty at certain junctures, most especially the JDK 4 – 5 transition, adding generics, would (at some point) have voted to take the breaking path, getting a clean semantics. The java architects took the other path, creating a train-wreck that damages the language and applications to this day, with no end in sight.
* The ongoing cost of this decision can easily be estimated in the billions in lost productivity, code bloat, and reduced power. Perhaps the larger ultimate cost is that this one botched decision will hasten developers flight to other languages such as C#, Dart, and fully-dynamic languages like JS and Python, whose adoption continues to rise, while Java’s drops. While it’s too early to assess the damage to the Java brand, I’ll hazard a guestimate that the net of the above factors will hasten Java’s sunset, rendering it largely irrelevant in the next 10 years.
* Not having access to internal conversations about this decision at the time, it isn’t clear that anyone attempted to develop a source transformer to migrate existing JDK 4 code to fully-embrace live types.

## Semantics

* primitives aren’t objects, creating a canyon that everyone and all tools must cross many times a day
* the good news: with uniform typing and advanced code optimization, this split can be buried into the VM and runtime, hence disappeared from the surface
* no macros
* no runtime types
* no code-generation pragmas
* no multivalue returns
* overly-minimal object literals, no extensions
  + Python devs regularly laugh at lack of map literals
* not all statements are expressions (fails the “universal evaluable” convention, starting with Lisp)
* no support for masquerading proxies:
* exceptions aren’t resumable
* no object morphing primitive, e.g. akin to Smalltalk’s :become, further hobbling proxies
* no way to tell compiler “P is a proxy for instances of type T”

### Type System

* type erasure has been an umitigated disaster, across the board
* compiler rejects statements simply because they’re not backward-compatible to pre-Jdk5 language, leading to tortured syntax, extra “helper” (more like “noise”) types, etc
* type parameterization declaration compilation faults are only occationally actionable; beyond the simplest constructs, they’re mostly so generic, and ignore the specific root cause, that faults that could be cleared in seconds to a few minutes can cost hours to track-down and fix
* type erasure
* the gulf between classes and types makes metaComputation and type access and leveraging forbiddingly hard, and in the absence of type erasure, has no justification
* no traits, mixins, multiple inheritance; cost: bloated code that’s way less comprehensible
* types are cloaked under obscure and irrascible reflection APIs
* no union types
* no type aliases
* type parameters have no defaults
* functionally-weak type inferencing

## Syntax

* excessively verbose
* no provisions for embeded (aka “island”) languages

## Scopes

* java’s scopes and protection/access methods interfere with
* fast access techiques via direct pointers (can’t enforce read-only access per-se), which *sets-up direct conflicts between encapsulation and performance*
* implementing many encapsulation patterns
* thereby compromises many security provisions,
* Addition of some flexible but disciplined name-spaces, worm-holes, etc could solve
* CF. C++’s *friend* provision

## MetaLanguage Constructs

* no meta-tooling constructs, such as constraints
* inability to extend auto-type conversion
* annotations aren’t first-class types, hence can’t be annotated

## Tooling

* compilers aren’t easily browsed, extended
* currently no integrated support for literate programming, linking knowledge, bugs, docs, snippets, examples, rants, etc with code constructs
* shitty display and info accessibility of all but a few toolModel types, especially
  + exceptions
  + traces

## Libraries and APIs

### Excessive Occurrence of Final Types and Members

* final is used liberally in the early JDK libraries

### Collections

* Map doesn’t implement Collection; the JDK is therefore lacking a base type[[1]](#footnote-1) underlying all collection types
* Like Map, arrays are also not adequately abstracted, forcing eggregious duplication of code to provide similar behaviors with Collection

### File System

# Higher-Order Programming

## Mixins

instead of being a mere nuisance, for which there are widely-understood, easily coded, and acceptable work-arounds, EBase’s very fabric relies on providing modular assembly (aka “composition”) of arbitrily rich types.

this motivates me to abandon Java at the earliest opportunity

* unfortunately, Scala traits are the only available alternate substrates, and even they don’t provide the required compositionality

Use case: abstract Node to NodeTrait, creating a modular means for *nodifying* the EO-based parts

* b:/java/mixins
* $links/programming\_languages/Java/multiple\_inheritance

### Key Use Cases

### Orthogonalizing Subject Carrying

* should we define Node.subject?
* yet another use-case for mixins: devs should be able to orthogonally refine types by adding a SubjectCarrier or HasSubject trait
* taking the next step, consider the effect of having a **HasEdges** trait? **Node** is only maybe useful as a common aggregation point in type-space

### Orthogonalizing Ordering and Chaining

* to unify OrderedEntityCache and OrderPreservingMap, we must
* make OrderedMetaEntity and ChainedLink plug-interchangeable; but not possible, since both are results of mixing-in a chain trait to MetaEntity and Link, respectively
* must make chain (order-preserving) ops independent of identity and other meta properties
* bidi containers may be implemented by combining ForwardLink and BackwardLink traits

### Modularizing Collections

* define IAppendableAssociativeStore, a mixin of IAppendableStore and AssociativeStore
* sets foundation for “hybrid” (ordered and/or appendable) maps, e.g. FlyweightTypeExtent

### Orthogonalizing Containment

* goal: slice and dice container types, making modular functional or feature units
* goal: build containers like building legos
* pallette: fine-grain prototypal modules
* missing language support

### Realizing Source and Sink Interfaces as Specific Mixing

* Q: should FlowNode always implement Function?
* can functionhood be factored out to a trait/mixin?
* to cut through the type tangle, collapse PortedFlowNode into FlowNode,
* approach:
* the above then unblocks
  1. mixing-in source/sink interfaces to FlowNode
  2. brute-force implementing ISource in as few types as possible, starting with FlowNode, which we anticipate will cover a majority of cases
     1. option: emulate mixing ISource into FlowNode, by defining a private Stream delegate, then lifting its behavior to FlowNode’s API via stock delegation glue code
  3. refactoring FlowPort or absorbing it into FlowNode, possibly with a Port trait mixed-in

consider additional metatypes roles:

* perspective, interpretation, e.g. tagging a type or instance with &Node and &Edge could alter a view
* additional metalevel decoration, e.g. &Trait, &Mixin
* consider &Aspect, &Facet, &Property, &Proxy,
* should we consider allowing a given type to have multiple metatypes? e.g. \_Sequence might have &List and &Entity
* such should fall-out of metatypes ancestry

### Implementation Patterns

* from “Metatype Support for Traits/Mixins”
* add comment to Metatype describing singular naming convention, inheritance, etc
* merge GraphDumper.TypeDescriptor back into TypeSystem
* identify the points of tension between single metatype inheritance and cross-cutting functional chunks
* what constitutes a reference?
* should Collection be under Struct, Value or AnyReference?
* how can we dispatch on fact that Collection and CollectionEntity are both iterable?
* should CollectionEntity be under Entity or Collection?[[2]](#footnote-2)
* how to categorize type members, e.g. should Behavior and Method be under TypeComponent or AnyReference?
  + 1. should we define TypeComponent?
* hang Collection, Behavior under AnyReference
* consider how the above tension can be healed
* e.g. look into introducing an orthogonal functionality categorization

## Type Inferencing Improvements

* type inferencing should be much deeper and more complete, in particular, is able to infer that TypeLink <: IBinding<Object,Type>
* more generally, I have yet to succeed in directly casting between either
* a generic interface G and any subinterface H that binds some or all of G’s parameters with fixed classes, which (of course) satisfy any of G’s parameters’ bounds, or
* 2 subinterfaces H1 and H2 that pass identical parameter bindings
  1. one can’t cast TypeLink to TypeBinding, since Java implements nominal vs. structural typing[[3]](#footnote-3)

## Pattern and Idiom Support

* Java provides no ability to implement programming patterns and idioms
* for example, take the powerful Instance-Inheritance Pattern
* defines a type T having a several application-defined facets appFacets, plus a parent facet
* in any T-instance t and appFacet f,
  + t.f => t.f != null ? t.f : (t.parent != null ? t.parent.f : null)
* or in otherwords, t.f returns f’s value in t if bound, else returns t.parent.f’s value, recursively
* the central point of this example is to show a standardized pattern that must be repeated for appFacet sets of arbitrary size, and that Java forces the pattern to be manually expanded for each facet
* macros provide one mechanism to reduce the effort
* a more powerful mechanism is to provide a way to declaratively specify application of patterns to entire class hierarchies
* for example, core entity facet accessors must handle update semantic to be
* transacted
* generate events
* propagates changes to storage, other replicas
* maintains association invariants such as 1:1 and many:1

# rogramming

# Lab

* look at minijava: <https://github.com/csaroff/MiniJava-Compiler?files=1>
* grok Java8.g4

1. note: a type may represent a class, an interface, or other construct [↑](#footnote-ref-1)
2. the case in point is that ejc doesn’t allow casting TypeLink to TypeBinding [↑](#footnote-ref-2)
3. not sure, needs checking [↑](#footnote-ref-3)