Valhalla_Goals

Introduction

A number of people describe Valhalla recently as being *"primarily about performance"*. While it is understandable why people might come to that conclusion -- many of the motivations for Valhalla are, in fact, rooted in performance considerations -- this characterization misses something very important. Yes, performance is an important part of the story -- but so are **safety**, **abstraction**, **encapsulation**, **expressiveness**, **maintainability**, and **co mpatible library evolution**.

Goals

The major goals of Valhalla are:

1. Align JVM memory layout behavior with the cost model of modern hardware;
2. Extend generics to allow abstraction over all types, including primitives, values, and even void;
3. Enable existing libraries -- especially the JDK -- to compatibly evolve to fully take advantage of these features.

1. Align JVM memory layout behavior with the cost model of modern hardware

Java's approach of *"(almost) everything is an object"* was a reasonable match for the hardware and compilation technology of the mid-nineties, when the cost of a memory fetch and an arithmetic operation were of comparable magnitude. But since then, these costs have diverged by a factor of several hundred. At the same time, memory costs have come to dominate application provisioning costs. These two considerations combine to make the graph-of-small-objects data representation, which typical Java programs result in, suboptimal in both program performance and provisioning cost.

The root cause of this is a partly accidental one: object identity. Every object has an identity, but not all objects *need* an identity -- many objects represent values, such as decimal numbers, currency amounts, cursors, or dates and times, and do not need this identity. This need to preserve identity foils many otherwise powerful optimizations.

Our solution for this is *value types*; value types are aggregates, like traditional Java classes, that renounce their identity. In return, this enables us to create data structures that are *flatter* (because values can be inlined into objects, arrays, and other values, just as primitives are today) and *denser* (because we don't waste space on object headers and pointers, which can increase memory usage by up to 4x), with a programming model more like objects -- supporting nominal substructure, behavior, subtyping, and encapsulation. "*Codes like a class, works like an int.*"

If you view values as being "faster objects", you could indeed view this fundamental aspect of Valhalla as being primarily about efficiency. But equally, you could view them as "programmable primitives", in which case it also becomes about better abstraction, encapsulation, readability, maintainability, and type safety.

Which is the real point -- that we need not force users to choose between abstraction/encapsulation/safety and performance. We can have both. Whether you call that "cheaper objects" or "richer primitives", the end result is the same.

2. Extend generics to allow abstraction over all types, including primitives, values, and even void

Generics are currently limited to abstracting only over reference types. Sometimes this is merely a performance cost (one can always appeal to boxing), but in reality this not only increases the cost, but decreases the expressiveness, of libraries.

Methods like Arrays.fill() have to be written nine times; this is nine methods to write, nine methods to test, and nine methods for the user to wade through in the docs.

Real-world libraries like Streams often resort to hand-coded specializations like IntStream; not only is this an inconvenience for the writer, but was also a significant constraint in the design of the stream library, forcing some undesirable tradeoffs. And this approach rarely provides total coverage; users complain that we have IntStream and LongStream but not CharStream. (For generic types with multiple type variables, like Map, the number of specializations starts to explode out of control.) The functional interfaces introduced in Java 8 are another consequence of the limitations of generics; because generics are boxed and erased, we had to provide a large number of hand-specialized versions (and still, not all the ones people want.) You don't need Predicate if you have can simply say Function<T, boolean> and not suffer boxing or erasure.

Everyone would be better off if we could write a generic class or method once -- and abstract over all the possible data types, not just reference types. (This includes not only primitives and values, but also void. Treating void uniformly is no mere "abstract all the things"; HashSet is based on HashMap, for implementation convenience, but suffers a lot of wasted space as a result; we could use a HashMap<T, void>. And the XxxConsumer functional interfaces are really just Function<T, void> -- we don't need separate abstractions here.)

Being able to write things once -- rather than having an ad-hoc explosion of types and implementations (which often propagates into further explosion at the client site) -- means simpler, more expressive, more regular, more testable, more composible libraries. Without giving up performance when dealing with primitives and values, as boxing does today.

Which is to say, again, just at the data-abstraction layer instead of the data-modeling layer: we need not force users to choose between abstraction /encapsulation/safety and performance.

3. Enable existing libraries -- especially the JDK -- to compatibly evolve to fully take advantage of these features

The breadth and quality of Java libraries is one of the core assets of the Java ecosystem. We don't want people to have to replace their libraries to migrate to a value-ful world; nor do we want existing libraries to be "frozen in time." (Imagine if we did lambdas and streams, but didn't do default methods, that allowed the Collection classes to evolve to take advantage of them -- Collections would have instantly looked ten years older.) There should be a straightforward path to extending existing libraries -- especially core JDK libraries -- to supporting values and enhanced generics, in a way that makes them "look built in". This may require additional linguistic tools to allow libraries to evolve while providing compatibility with older clients and subclasses that have not yet migrated.

Just providing these features for new libraries is not enough; if widely used libraries can't compatibly evolve to take advantage of this new world, they are effectively consigned to a slow death. This might be OK for some classes -- deprecating OldX and replacing it with NewX -- but only when uses of the OldX types aren't strewn through lots of other libraries. That rules out starting fresh with Collections, Streams, JSR-310, and many other abstractions, without rewriting the whole JDK (and many third party libraries). So instead, we have to provide a compatible path for such libraries (and their clients) to modernize.

Summary

So, to summarize: Valhalla may be motivated by performance considerations, but a better way to view it as enhancing abstraction, encapsulation, safety, expressiveness, and maintainability -- *without* giving up performance.