State of the Values

April 2014: Infant Edition
John Rose, Brian Goetz, and Guy Steele

"Codes like a class, works like an int!"

This is a sketch of proposed enhancements to the Java Virtual Machine instruction set, and secondarily to the Java Language, to support small immutable, identityless *value types*. (They may also be considered as identityless aggregates, user-defined primitives, immutable records, or restricted classes.) This is an early draft, intended to describe the overall approach. Readers are expected to be familiar with the JVM bytecode instruction set.

Background

The Java VM type system offers two ways to create aggregate data types: heterogeneous aggregates with identity (classes), and homogeneous aggregates with identity (arrays). The only types that do not have identity are the eight hardwired primitive types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. Object identity has footprint and performance costs, which is a major reason Java, unlike other many object oriented languages, has primitives. These costs are most burdensome for small objects, which do not have many fields to amortize the extra costs.

> *More detail:* In terms of footprint, objects with identity are heap-allocated, have object headers of one or two words, and (unless dead) have one or more pointers pointing at them. In terms of performance, each distinct aggregate value must be allocated on the heap and each access requires a dependent load (pointer traversal) to get to the "payload", even if it is just a single field (as with `java.lang.Integer`). Also, the object header supports numerous operations including `Object.getClass`, `Object.wait`, and `System.identityHashCode`; these require elaborate support.

Object identity serves only to support mutability, where an object's state can be mutated but remains the same intrinsic object. But many programming idioms do not require identity, and would benefit from not paying the memory footprint, locality, and optimization penalties of identity. Despite significant attempts, JVMs are still poor at figuring out whether an object's identity is significant to the program, and so must pessimistically support identity for many objects that don't need it.

> *More detail:* Even a nominally stateless object (with all final fields) must have its identity tracked at all times, even in highly optimized code, lest someone suddenly use it as a means of synchronization. This inherent "statefulness" impedes many optimizations. Escape analysis techniques can sometimes mitigate some of these costs, but they are fragile in the face of code complexity, and break down almost completely with separate compilation. The root problem is identity.

Running example: `Point`

Consider a `Point` class:

```
final class Point {
    public final int x;
    public final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
```

```
        }
    }
```

Even though `Point` is immutable, the JVM doesn't usually know that you have no intention of using its identity (for example, casting one to `Object` and using it as an intrinsic lock), and hence represents `Point` as a "box" object for `x` and `y`. An array of `Point` objects requires an extra object header (8 to 16 bytes) and a reference (4 to 8 bytes), meaning those 8 bytes of data take up 20 to 32 bytes of heap space, and iterating this array means a pointer dereference for every `Point` visited, destroying the inherent locality of arrays and limiting program performance. (Plus, allocation entails work for the garbage collector.) Programmers often resort to tricks like representing an array of points as two arrays of ints to avoid these costs, but this sacrifices encapsulation (and maintainability) just to reclaim the performance lost to identity. If, on the other hand, we could represent a `Point` in the same manner we do an `int`, we could store them in registers, push them on the stack, iterate through an array with locality, and use far less memory, with no loss of encapsulation.

Thesis

The goal of this effort is to explore how small immutable user-defined aggregate types without identity can be surfaced in the language and the JVM instruction set to support memory- and locality-efficient programming idioms without sacrificing encapsulation.

We believe that the design of the Java VM and language can be gently extended with a new kind of type, a *value type*, which usefully combines the properties of Java's current classes and primitive types.

We expect to borrow most of the definition and encapsulation machinery from classes, allowing users to easily and safely build new value-based data structures. In particular, we believe that it is useful, both at the JVM and language levels, to regard value type definitions as specially marked, specially restricted class definitions.

At the same time, users of value types will be free to treat them as if they were simply a new kind of primitive type.

If we succeed, the user experince with value types can be summed up as, "Codes like a class, works like an int!"

As will be seen below, we also believe that, while the value types are usefully similar to classes, it is useful (as with primitives) to make clear and systematic distinctions between a value type and a reference type, at the operational level of bytecodes.

We will touch on how these features might be surfaced in the Java Language, but this is a secondary concern for the time being. (The interplay of value types and generics is an area of great interest, but will be treated as a separate exploration.) Earlier explorations of the ideas contained herein include Value Types in the VM and Tuples in the VM. This has been a long-running design problem for Java, as may be seen in James Gosling's call for "efficient classes".

Use cases

Thorough integration of values into the JVM can support a number of language features which may be desirable additions to the Java language, or features already implemented suboptimally by other JVM languages. Some examples include:

- *Numerics.* The JVM offers exactly eight efficient numeric types; if you can't cram your problem into these, you're out of luck. Numeric types like complex numbers, extended-precision or unsigned integers, and decimal types are widely useful but can only be approximated (to the detriment of type safety and/or

performance) by primitives or object classes.

- *Native types*. Modern processors support a wide variety of native data types, and (as with numerics) only a few of them map directly to Java primitives. This makes it difficult or impossible to write Java code which compiles directly to (for example) vector instructions.
- *Algebraic data types*. Data types like `Optional<T>` or `Choice<T,U>` should not need an `Object` box. Many such types (especially product types like tuples) have natural representations in terms of small identityless aggregates. Unit types (both `void`-like and metric) are sometimes useful, but only if their footprint overhead can be driven to zero.
- *Tuples*. A tuple of values should itself be regarded as a value, and should not need an `Object` box. (Even if the language does not support tuples in its type system, many languages support multi-valued return, which is a special sub-case of tuples.)
- *Cursors*. An iterator or other cursor into a complex data structure should not require an `Object` box. Moreover, it should be possible for clients of data structures (managed and/or native) to pass around iterators and other "smart pointers" into the data structures with full encapsulation and type safety.
- *Flattening*. Values provide a natural way to express data structures with fewer pointer indirections. Although value types are not designed as a general mechanism for layout control, using them will permit JVMs to arrange some data structures more efficiently.

Requirements

An efficient value-type facility at the JVM level should offer the following beneficial characteristics:

- *Scalarization*. Instances of value types should be routinely broken down into their components and stored in registers or the stack, rather than on the heap. When they must be stored in the heap as part of a containing object, the JVM should be free to use *flattened*, pointer-free representations.
- *Wrapping*. Just as the eight primitive types have wrapper types that extend `Object`, value types should always have both an unboxed and a boxed representation. The unboxed representation should be used where practical; the boxed representation may be required for interoperability with APIs that require reference types. Unlike the existing primitive wrappers, the boxed representation should be automatically generated from the description of the value type.
- *Behavior*. Value types are not simply tuples of data; they should be able to define behavior, in the form of methods. (For example, a complex number class would define methods to perform complex arithmetic. The JVM could then choose to intrinsify these if a hardware primitive is available, say for 128-bit numerics. Similarly, a data structure cursor would define factory and access methods, while keeping its component values securely encapsulated.) Standard methods like `toString` and `equals` should not be forced on value classes, but should be customizable.
- *Names*. The JVM type system is almost entirely nominal as opposed to structural. Likewise, components of value types should be identified by names, not just their element number. (This makes value types more like *records* than *tuples*.)
- *Interoperability*. There are a number of operations on `Object` that need to have a values counterpart, such as `equals`, `hashCode`, and `toString`. The

compiler and/or VM should be free to implement these componentwise in the absence of other instruction from the value class's author. Besides `Object`, other types such as `Comparable` should also be able to interoperate with suitable value types, opening the door to types like `TreeSet<UnsignedInteger>`. (This implies that interfaces may sometimes claim subtypes which are not also subtypes of `Object`.)

- *Encapsulation*. Values should be able to have private components, just as objects have private fields. This enables, for example, secure foreign "pointers" or cursors. Would-be attackers must not be able to subvert encapsulation by extracting private components or forging illegal values by strained combinations of legal operations.
- *Verifiability*. Support for values in the bytecode instruction set must be verifiable just like other JVM types.
- *Variables*. Existing variable types (fields, array elements, local variables, and method parameters) should all be able to hold values. Methods should be able to return values. The qualifiers `final` and `volatile` should apply regularly.
- *Arrays*. Arrays of value types should be packed, without indirections, as arrays of primitives are now. (This does not mean that the arrays themselves are packed inside other objects. Separate proposals are investigating multi-dimensional and inlined arrays.)

Since values do not have identity, there are certain operations that are identity-specific and either need to be disallowed on values or assigned a new meaning for use in the context of values.

- *Locking*. Using a value as the lock object of a synchronized statement, or calling the related methods `wait` and `notify`, should be disallowed.
- *Identity comparison*. The `==` operator on objects performs an identity comparison; on primitives it performs a bitwise comparison. (…With an asterisk for `NaN`.) Since a value type does not (necessarily) have a pointer, this operator does not apply to value types. The logical candidates for interpreting `==` on value types would be componentwise `==` comparison, or invoking an `equals` method. (If the former, the VM needs to provide a mechanism to perform the bitwise comparison, since some components may be private. Even if the `equals` method is used uniformly, the implementation of the method may benefit from a bitwise operator at the bytecode level. See `vcmp` below.)
- *Identity hash code*. The default identity-based hash code for object, available via `System.identityHashCode`, also does not apply to value types. Internal operations like serialization which make identity-based distinctions of objects would either not apply to values (as they do not apply to primitives) or else they would use the value-based distinction supplied by the value type's `hashCode` method. (A componentwise hash computation would usually be a reasonable default, though it might leak information about private components.)
- *Clone*. Cloning a value type does not strictly make sense, but it might be reasonable to interpret clone as an identity transform for value types.
- *Finalization*. Finalization makes no sense for values and should be disallowed (though values may hold reference-typed components that are themselves finalizable.)

Many of the above restrictions correspond to the restrictions on so-called value-based classes. In fact, it seems likely that the boxed form of every value type will be a

value-based class.

A "soft" restriction on values is that they are "not too large". Values with tens of components, though legal, are not the design center of this proposal. Since values are passed, well, *by value*, the overhead of copying many components is increasingly likely to overwhelm any savings obtained by removing the overhead of a pointer and an object header, as the number of components grows. Large groups of component values should usually be modeled as plain classes, since they can be passed around under single pointers.

Put positively, a well-conceived use of a value type gets a performance and complexity win by omitting the object pointer and header, and routinely passing the components by value. A value type with a small number of components will neatly match the capabilities of present CPUs, which have relatively small limits on the size of their register file. When those limits are passed, the JVM will also have flexibility to spill values onto stack stack and/or managed heap memory.

Life without pointers

Also a value does not have its own pointer, or a least such a pointer is a secret inside the VM's implementation. Therefore there are certain additional operations that also either need to be disallowed or assigned a new meaning.

- *Null*. Null is a valid value of every reference type, meaning "no instance". But assignment to, and comparison to, `null` for value types should be disallowed. This restriction does not apply to the boxed form of a value type. (We may wish to give an alternate "opt-in" interpretation, primarily to support migration compatibility for classes migrating to value types.)
- *Side effect*. Although a variable containing a whole value can be reassigned to a different value, the individual parts of a value should not be subjected to piecewise side effects, as if they were accessed through a pointer to the value. You can change an int from 2 to 3, but you cannot change the lowest bit of 2 and expect a variable somewhere else to become 3. The most Java gives in that direction are compound assignment operations like `intval |= 1`, which, despite attempts in some languages, do not appear readily adaptable to general composites. (This is perhaps a controversial position, since many other languages allow values to be accessed through pointers, up to and including component mutation. At present this conceptual complexity appears to belong elsewhere, perhaps where it can assist with native data structures as well as JVM data.)
- *Reference cast*. Like a primitive, a value type must be boxed before it can be used as a reference. The language should have rules for implicitly boxing and unboxing values, as for primitives.
- *Pointer polymorphism*. Since Java objects incorporate run-time type information, methods can be virtual, types can be tested at runtime, and (many) static reference types can point to a range of concrete subclasses at runtime. All this works because a pointer variable, though fixed in size, can refer to data of any size and type. Since values have no header to carry type information, and are designed to be flattened into their containing objects, it follows that values cannot dynamically type tested and cannot be of variable size. This means that subclassing would be of little use for value types. (But there is a probable role for things like abstract super types of value types, at least for interfaces like `Comparable`.)
- *Reflection*. As an extreme case of pointer polymorphism, any object can be cast to the top type `Object` and be inspected for its string representation, its class, its

set of fields and field values, etc.

- *Atomicity*. A pointer provides an access path to its object which can be changed atomically. That is, racing threads can observe either the old object or the new object, but not a mix of both. By comparison, a multi-field value might be subject to data races in which a mix of old and new field values might be observed by racing threads. This problem of "struct tearing" can sometimes be observed as a corner case with `long` and `double` types, but is more pressing with value types, since it can severely damage encapsulation. We believe that the JVM should supply atomicity on request at both type use and type definition sites, as it does now (at type use sites) with `long` and `double` variables declared `volatile`. The cost of guaranteed atomicity in present hardware is large enough to demand that it be an option, rather than required behavior. The precise tradeoffs vary across platforms, much as they once did with `long` and `double`.

General approach: definition

We've already identified several requirements of value types that are common with classes: a collection of named, type components (fields), behavior (methods), privacy (access control), programmatic initialization (constructors), etc. It makes sense, therefore, to use existing elements of classfiles to the extent possible to describe value types, with additional metadata as needed. Similarly, it makes sense to re-use syntactic elements from classes in the source language.

We have good precedent for taking this approach, since interfaces are closely similar to classes. Apart from some mode bits and minor syntax, the binary of an interface is identical to that of a proper class, except that interfaces have additional syntactic and semantic restrictions. The same point applies to the source language.

Using a strawman syntax (please, no comments on syntax!), we could express our `Point` example as a value as follows:

```
final __ByValue class Point {
    public final int x;
    public final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Point that) {
        return this.x == that.x && this.y == that.y;
    }
}
```

The syntactic difference between this and its reference equivalent start with the keyword `__ByValue` marking the class. (This syntax is a placeholder only; the key point being illustrated here is that we intend to reuse many coding idioms from classes, where appropriate, to describe values, such as fields, constructors, and methods.) Both the whole class and the individual fields `x` and `y` are required to be final. Other relevant restrictions are enforced as well (e.g., no use of value-sensitive operations). We presume the compiler or VM would fill in an appropriately specified componentwise `hashCode`, `equals`, and `toString` method, if the user does not provide one (and the language allows this omission).

If another class declared a field of type `Point`, the VM would allocate storage for the `x` and `y` components in the hosting class, rather than a reference to a `Point` box. (This can be viewed as object inlining or flattening.) Similarly, an array of `Point`s would be laid

out as a packed array of alternating `x` and `y` values, rather than a series of references.

As you can see, this clearly "codes like a class". In fact, except for the funny annotation-like keyword, the meaning of this definition is immediately clear to any Java coder.

What are the pitfalls for the coder of this class? It appears that they are mainly due to the restrictions the user agrees to in exchange for getting "by value" semantics. For example, it is unlikely that the keywords "extends" or "protected" will be accepted in a value type definition.

The language already has sufficient power to express some of the necessary restrictions, such as the immutability of subfields after construction or the inability. If the user leaves off a "final" keyword, the compiler will explain that it is necessary.

> *Alternate route:* We could also, as with interfaces and closures, switch the defaults and silently add the "final" keywords where required. This seems like an easy win for brevity, but since clarity is more important than brevity, and since all asymmetries between normal classes and value types will create surprises, we wish to be cautiously sparing with symmetry-breaking. Also, switching the defaults would make it hard to loosen restrictions later on, if that should prove desirable. Eliding the `Object` methods is not so surprising, and is likely to be permitted, since system-supplied defaults are expected here.

General approach: usage

The next question is, how does it "work like an int"? Here are some simple uses of our `Point` class:

```
static Point origin = __MakeValue(0, 0);
static String stringValueOf(Point p) {
    return "Point("+p.x+","+p.y+")";
}
static Point displace(Point p, int dx, int dy) {
    if (dx == 0 && dy == 0)
      return p;
    Point p2 = __MakeValue(p.x + dx, p.y + dy);
    assert(!p.equals(p2));
    return p2;
}
```

Either of these methods would work well as members of `Point` itself. In these examples, a point can be a field, argument, local variable, or return value. In all those cases, the JVM is likely to place the components of the point into independently managed locations, such as machine registers. Note that `displace` returns a new point value, which the JVM is likely to return by value and not by (invisible) reference.

Like primitives, value type can guide overloading. Here are some more overloads for `stringValueOf` which are compatible with the above:

```
static String stringValueOf(int i) {
    return Integer.valueOf(i);
}
static String stringValueOf(Object x) {
    return x.toString();
}
```

How do these usages differ from a plain `int`? First of all, the dot syntax for member selection applies to value types, as with `p.x` or `p.equals`. Primitives do not support the dot syntax at all. (It is possible that the value/box duality gives us a foot in the door for applying methods on primitives, by defining methods on pseudo-types `int`, `float`, etc.

Many of the methods of wrappers like `Integer` would be candidates for such treatment.)

Second, there are no literals for `Point`. Instead, the constructor of the definition must be invoked. Encapsulation demands that component-wise value creation must not be possible unless provided for by a suitable constructor. We use a place-holder keyword `__MakeValue` to mark the two places above where new point values are created. (Some possible spellings for the keyword are "new Point", "Point", and the null string. Some ongoing design efforts for quasi-literal expressions apply here also, but are beyond the scope of this proposal. It seems plausible that omitting "new" would be a tasteful way to express the presence of a non-heap value.)

Just as there are no literals, there is no concept of compile-time constant for values. This might not seem urgent for `Point`, but will likely be felt as a burden when programmers use value types for unsigned numbers or other very simple quantities.

> *Alternate route:* In principle, Java could adapt some features from other languages to support user definition of literals and constant expressions. Difficulties would include adapting the design to "feel like Java" (in simplicity, class orientation, and separate compilation), and clearly defining how to run user code at compile time (to execute constant-creating constructors and methods).

Third, Java's built-in operators, which are the main way of working with primitives, do not apply to `Point`s. Plausible exceptions might be those operators which work on every kind of value, namely the relationals (`==` and `!=`) and string concatenation (`+`).

String concatenation is dangerous ground, since it is subtly overloaded precisely at the distinction between classes and primitives, which is where value types live. (As most every Java programmer has found by trial and error, the expressions `("a"+1)+2` and `"a"+(1+2)` differ in a tricky way.)

> *Alternate route:* It is wise to leave space here for true numeric addition, depending on where the story with numerics and operators goes in the future. Likewise, cursor types might work better with `for`-loops if operators like `j < k` and `j++` were supported, but this again depends on both the future story with operators and with possible enhancements to `for`-loops themselves. A final observation: user-defined operators also seem to require compile-time code execution to build constant expressions. But we do not propose to solve all this in one step.

The simple relationals probably pass muster as aliases for the `equals` method, by analogy with other primitives. (Low-level bitwise equality seems to be another candidate, departing from the operator's behavior of `float` and `double`, but that would break abstraction.) The major objection to these operators is that they can potentially lead to behavior divergences between the boxed and unboxed form of a value, as follows:

```
Point p1 = __MakeValue(0, 1);
Point p2 = __MakeValue(1, 0);
Point p3 = __MakeValue(1, 0);
assert(!p1.equals(p2));  // value equality
assert(p1 != p2);  // value equality, again
assert(p2.equals(p3) && p2 == p3);
Object box1 = p1, box2 = p2, box3 = p3;  // box 'em
assert(box1 == box1);  // x==x always (except NaNs)
assert(!box1.equals(box2)); // Object.equals calls Point.equals
assert(box1 != box2);  // must also be
assert(box2.equals(box3));
if (box2 == box3)  println("the cat died");  // indeterminate
assert((Point)box2 == (Point)box3);  // back to Point.equals
if (box2 == (Object)p2)  println("the cat died");  // indeterminate
```

With minor adjustments, all the above tests can be reproduced with `int` and `Integer`. (For some small integer values, the cat never survives, but this is a peculiarity of the `Integer.valueOf` factory method, rather than an inherent feature of primitives, values, or boxes.)

A fourth aspect of primitives is that, although they do not have values that correspond to the null reference, they do have default values, generally some sort of zero. This default value shows up in the initial contents of uninitialized fields and array elements. It is therefore necessary to specify a default value for each value type. The simplest default composite value is the one which which is recursively default in all of its subfields. Given that predefined default values are a fixture in Java, this convention does not appear harmful, so we will adopt it.

As a result, the author of methods in a value type definition will need provide behavior to default (all-zero-bits) values, even if the constructors never create them, since they can be observed in some uninitialized variables. It is likely that we will either forbid no-argument constructors or require them to produce default values, to avoid “duelling defaults”.

> *Alternate route:* We could force the explicit construction of default values by mandating the inclusion of a public null constructor, like this: `public Point() { x = 0; y = 0; }` All sub-fields would be initialized to default values, or the compiler and verifier would report an error. This amount of required boilerplate feels like overkill, even though it is logically similar to requiring the `final` keyword in various places. It would also prevent null constructors from being used for other purposes. It it more likely that we will disallow null constructors, and reserve their meaning for default values.

> *Alternate route:* A more elaborate default could be specified through some pattern or syntax imposed on standard class definitions, but there are three objections to this: It requires an special extension rather than a simple restriction. It also requires special rules to cope with side effects from the null constructor. Finally, applying user-specified defaults at runtime is much more costly than a convention of all-zero-bits; it would not be a shallow change to the JVM.

Details, details

Our mantra is “codes like a class, works like an int.” Of course, the devil is in the details; there are going to be features of classes that do not make sense for value types. Some questions we might want to ask about value classes are as follows, with tentative answers where appropriate. Where a comparison with primitives would make sense, the answers for these questions for values are the same as they would be for primitives.

- Subtyping
  - Can a value class extend a reference class type? No.
  - Can a reference class extend a value type? No.
  - Can a concrete value class extend another value type? No. (Because concrete value classes are final.)
  - Can a value class be abstract or non-final? No. (Abstract value classes do not presently seem worthwhile.)
  - Can a value class implement interfaces? Yes. Boxing may occur when we treat a value as an interface instance.
  - Can values participate in inheritance-based subtyping? No. (Just like with primitives.)
  - Is there a root type for values? No. (A value supertype like `Object` seems to require a templating mechanism.)
- Containment
  - Can a value class contain a field of reference type? Yes. (Because a value

codes like a class.)
    - Can a reference class contain a field of value type? Yes. (Because a value works like an int.)
    - Can a value class contain a component field of value type? Yes. (…Works like a class.)
    - Can an array contain elements of value type? Yes. (Works like an int. And the array itself is an object.)
    - Can a value class contain a field of array type? Yes. (But only because arrays are objects. Inline arrays are part of a different line of investigation.)
    - Can a value class contain a non-final field? No. (Logically possible, with a surprising explosion of complexity.)
    - Must all fields of a value type be recursively immutable? No. (Recursive immutability is an independent feature of types which does not presently require language support.)
    - Can values have member types? Yes. (Non-static member types have hidden fields that record containing values.)
    - Can values be member types? Yes. (If non-static, they have hidden fields that point to containing objects.)
    - When creating an array of values, what are the initial elements? They are all the default all-zero-bits value. (Just like with primitives.)
    - What is the initial value of a field whose type is a value? The default all-zero-bits value, until the first assignment or initialization. (Again, like primitives.)
    - What is the initial value of a local variable whose type is a value? Trick question: Definite assignment rules forbid observing such a value, until the first assignment.
    - Can a value class contain a field of its own type? No, neither directly nor indirectly. (This is a new restriction, but is similar to existing ones about a class not subclassing itself.)
- Compatibility
    - Are values objects? No, although they can be boxed into objects.
    - Are primitives values? Possibly. (Value classes named `int`, `boolean`, etc., would provide good successors to the existing wrapper types.)
    - Are arrays of values covariant? No. (Just like with primitives. Arrays of value **boxes** are covariant.)
    - Do values participate in implicit conversions? Probably not. (Unlike primitives; it is not clear that we need implicit conversions like `int` to `UnsignedInteger`.)
    - Does every value class define object-like methods (`toString` etc.)? Yes, so their boxed forms can interoperate with `Object` APIs.
    - Are there default "obvious" component-wise implementations for these methods? Yes. (A root type for boxes is a logical place to put default component-wise code.)
    - Can value types be serialized? Not by default, since that would violate encapsulation.
    - Can the user of a value type request atomicity? Yes. (This will continue to be part of the meaning of the `volatile` keyword. Perhaps we will also allow `__AlwaysAtomic`, if it is not spelled `volatile`.)
    - What happens when a value type is assigned to Object? It is boxed into a JVM-generated class with the same API as the value class.
    - What happens when null is assigned to (or compared to) a value type? If the value is boxed, a reference comparison will return not-equal; if the null is unboxed, an NPE will be thrown. (Same questions as for primitives.)
- Encapsulation

    - Can values have nonpublic fields? Yes.
    - Can values have nonpublic methods? Yes.
    - Can values have static methods or fields? Yes.
    - How are values constructed? By calling a constructor. Value-type constructors are really factory methods, so there is no `new;dup;init` dance in the bytecodes.
    - Can the definer of a value type demand atomicity for all its values? Yes. (Necessary for encapsulation as noted above. Will need additional syntax on the class definition, with `__AlwaysAtomic` as a placeholder for now.)
    - Can I arrange my encapsulation to exclude all default componentwise operations? Yes. (By overriding the relevant methods.)
    - Can a value change size or layout without requiring clients to recompile? Yes.
    - Can I arrange my encapsulation to exclude values all of whose fields are zero? No. (As noted above, a deliberate compromise. The default value should be reconciled with the no-argument constructor value, as noted above.)
    - Can bytecodes be used in a way to access private fields of a value? No. (E.g., you can't push a value and pop a reference.)
- Other details
    - Is there a limit on the complexity of value types? Yes. (But not low enough to make performance guarantees. Perhaps 255 words, like the limit on function parameters.)
    - Can I refactor a value class by splitting out a super-class? No, unless we allow abstract value types.
    - Can I refactor methods into and out of value types? Yes, if you use common interfaces with default methods.
    - Can I call value methods on boxed values? Yes. (The bridges will be there, though you won't usually notice that you are using them.)
    - When I say `Point` when does it mean the value type and when does it mean the box reference type? This needs work. (Hopefully you won't need to care, mostly.)

The decision to limit or prohibit subclassing and subtyping of value types is necessary to avoid pointer polymorphism. Abstract super types (like the interface `Comparable`) are only considered because they cannot be instantiated.

We can therefore ensure that all methods are resolved unambiguously in the exact type of the method receiver. Invoking a value method is always like `invokestatic` or `invokespecial` and never like `invokevirtual` or `invokeinterface`.

This decision also sidesteps a problem with arrays of values, which would seem to require variable-sized elements if an array could contain (say) both 2-field `Point` and 3-field `ColoredPoint` values. For arrays, it is convenient to lean on "works like an int", since primitive arrays already have completely homogeneous elements. This is not a perfect story. For example, if we have a value type `Bignum` that implements `Comparable` (see example below) we will allow an array of boxes `Bignum.__BoxedValue[]` to be a subtype of the interface array `Comparable[]`, but the flat array type `Bignum[]` cannot also be a subtype of `Comparable[]`.

New bytecodes and type descriptors

In determining how to surface value types in the bytecode instruction set, there are two extremes: add all-new bytecodes for values, treating values as an entirely new thing, or overload existing bytecodes to accept either values or references. While the final answer

will almost certainly be somewhere in the middle, for purposes of exposition and analysis we will initially err on the side of the first choice, for explicitness and simplicity. At worst, we anticipate fewer than a dozen new bytecodes will be needed, and one new form of type descriptor.

Currently, there are single-letter type descriptors for primitives (e.g., `I` for `int`), "L" descriptors for classes (e.g., `Ljava/lang/Object;`), and for any type descriptor, one can derive the type descriptor for an array of that type by prepending a `[`. We add one more form, for value types; the type descriptor for a value type `com.foo.Bar` would be `Qcom/foo/Bar;`, which is just like the "L" descriptor, except with a "Q". (Perhaps it stands for "quantity"; "V" already means `void`.) For example, if we had the following method:

```
public static double getR(Point p) { ... }
```

its signature would be `(QPoint;)D`, indicating it took a single argument of value type `Point`.

New bytecodes are described in the following format:

```
// Description
opcode [ constant pool operands ] stack operands -> stack result
```

For all opcodes, a "qdesc" is a Q-descriptor for a value type; for most bytecodes, it seems possible for the verifier to infer the descriptor and hence omit it from the bytecode. Local variable table slots and stack slots can hold values as well as primitives or references.

For simplicity and better binary compatibility, we define (unlike with longs and doubles) that values always consume a single slot in the local variable array. As with classes, we want values to be able to change size and layout without recompiling clients.

We posit the following bytecodes for manipulating values on the stack. Again, while some of the forms here are arguably unnecessary, we err on the side of explicitness and completeness, minimizing the number of new bytecodes can come later.

```
// Load value from local variable
vload [ qdesc?, index ] -> value

// Store value into local variable
vstore [ qdesc?, index ] value ->

// Create new array of value types
vnewarray [ qdesc ] size -> arrayref

// Store value into array
vastore [ qdesc? ] arrayref, index, value ->

// Load value from array
vaload [ qdesc? ] arrayref, index -> value

// Extract a component from a value
vgetfield [ field-desc ] value -> result

// Insert a component into a value (for private use by constructors)
vputfield [ field-desc ] value argument -> value

// Construct a fresh value with all default components (for use by constructors)
vnew [ qdesc ] -> value

// Invoke a method on a value
vinvoke [ method-desc ] value, (argument)* -> (return)?
```

```
// Return a value
vreturn [ qdesc? ] value
```

Additionally, there are a number of "polymorphic" bytecodes, such as `dup`, that can operate on stack contents of any type; these would be extended to support values as well.

Many of these could be overloaded onto existing bytecodes with little discomfort. For example, the `vinvoke` functionality could be overloaded onto `invokestatic`; the `vgetfield` functionality could be overloaded onto `getfield`. Again, for clarity of design we refrain from doing such overloadings, at least at this point.

The field and method descriptors referred to above would be constant pool references of type `CONSTANT_Methodref` and `CONSTANT_Fieldref`. The `CONSTANT_Class` components of the descriptors would refer to the value types by their normal names (without any "Q" prefix). The boxed forms of the class would have their own bytecode name, perhaps derived from the descriptor language (e.g., `.LFoo;`).

> *Alternate route:* We may wish to continue the pattern of opcode cloning by also cloning the constant pool types, as `CONSTANT_ValueMethodref`, etc. But we do not yet see a reason to force us to do this.

Static methods in values do not need special opcodes; they will continue to be invoked by `invokestatic`, and similarly for static fields in values.

Constructor invocation will *not* pass in the value to be constructed by reference (since that is defined to be impossible). Instead, value type constructors will be static factory methods which will be invoked by `invokestatic`. Individual mutation steps for component initialization are represented, internally to the constructor, by use of the privileged instruction `vputfield`, which operates equivalently to the special `putfield` that initializes an normal all-final object. Note that `vputfield` returns the updated value; there is no side effect.

> *Alternate route:* We could combine the effects of the `vnew` and `vputfield` operators, as used by constructors, into a `vpack` instruction, which would take a series of components on the stack. The order and type of those components would be implicitly defined by the containing value class. Although this would make some simple constructors slightly more compact, it would introduce a new kind of coupling between an instruction and its containing class. It seems simpler to mimic the other instructions which operate on JVM composites as clusters of named fields, rather than on ordered tuples of stacked values. The correspondence between bytecode and source code is also simpler to track, if each field initialization has its own `vputfield` opcode.

Boxing and object interoperability

Every value type has a corresponding box type, just as `Integer` is the box type for `int`. Rather than having a separate source and classfile artifact and tracking their association, the box type for value types is automatically derived from the classfile for the value type. If the value type is described in the bytecode as `QFoo;`, the corresponding box type is described as `LFoo;`. For both types, the VM will know to look for a classfile called `Foo` and generate a value- or reference- view of that class.

Since many common operations, such as construction, will have different mechanics for value or reference classes, either the static compiler will generate multiple versions of class artifacts (e.g., a standard constructor with signature `(...)V` and a value constructor with signature `(...)QFoo;`) or the VM will derive one from the bytecode of the other as needed.

For sake of exposition, for a value class `Foo`, we'll write `Foo` to refer to the value form

and `Foo.__BoxedValue` to refer to the boxed form of `Foo`. (Explicit naming of the box type will rarely be needed, as autoboxing should take care of most value-reference interactions. Note that we are not speculating on syntax here.)

The conversions between the boxed and unboxed representations of value types might be represented by methods following a naming convention, as they are with the current primitive boxes (e.g., `valueOf(boxed)`) or might be represented by conversion bytecodes like these:

```
// Box a value
v2a [qdesc] value -> ref

// Unbox a value
a2v [qdesc] ref -> value
```

Comparisons between values (analogous to `acmp` and `icmp` bytecodes) might also be represented by methods following a naming convention, or with special bytecodes like these:

```
// Compare two values using componentwise, bitwise and reference equality
vcmp [ qdesc? ] value, value -> boolean
```

(In examples below we will use patterned method names, to show another possibility.)

Although we find it useful for design purposes to contemplate the "cloning" of new value bytecodes like those which manipulate primitives and references, the conversion and comparison bytecodes are likely to work best as simple method calls (using `vinvoke`).

Box types, like array types, are generated by the system, so they have a "magic" feel to them. Our intention is to make them as un-magic as possible. One way to do this is to factor box code, as much as possible, into a common abstract superclass. This is approximately the approach taken for classic objects, where the default behaviors are found in `java.lang.Object`.

Now we have enough structure to work some examples.

Example: construction

For objects, objects are initialized by first executing a `new` bytecode (which creates an uninitialized object), and then invoking the constructor (using `dup` and `invokespecial`.) Constructors for values behave more like factory methods. The following shows the bytecode generated for Point's constructor:

```
final __ByValue class Point {
    public final int x;
    public final int y;

    public Point(int x1, int y1) {
            ; public static <new>(II)QPoint;
        //implicit: initialize 'this' to all-zero-bits
            ; vnew           // stack holds [ this ]
        this.x = x1;
            ; iload 0 ('x1') // stack holds [ this x1 ]
            ; vputfield 'x'  // stack holds [ this ]
        this.y = y1;
            ; iload 1 ('y1') // stack holds [ this, y1 ]
            ; vputfield 'y'  // stack holds [ this ]
        //implicit: return 'this'
            ; vreturn
    }
```

```
}
```

To create an instance of a `Point` in a local variable, the factory method is invoked, and the result stored in the local:

```
Point p = new Point(1, 2);
    ; iconst_1
    ; iconst_2
    ; invokestatic "Point.<new>(II)QPoint;
    ; vstore 6 (create 'p')
```

Note that `invokespecial` is not relevant here, since the factory method does not take a pointer to an uninitialized value as its implicit first argument. To simplify exposition, we have changed the name of the constructor of a value type to `<new>`, to emphasize that it is a factory method.

Example: nested values

```
final __ByValue class Path {
    final Point start;
    final Point end;
}

class Foo {        // regular class
    Path path;     // regular field
}

int startX = path.start.x;
    ; aload 0 ('this')       // stack holds [ Foo ]
    ; getfield Foo.path      // stack holds [ path ]
    ; vgetfield Path.start   // stack holds [ start ]
    ; vgetfield Point.x      // stack holds [ x ]
    ; istore 1               // stack holds [ ]
```

Note that when a nested value is read, its enclosing values are always read first.

Example: arrays

Arrays are created with the `vnewarray` bytecode, and manipulated much like any other array type.

```
Point[] points = new Point[100];
    ; bipush 100
    ; vnewarray Point
    ; astore 1 (create 'points')
points[3] = new Point(2, 19);
    ; aload 1 ('points')
    ; iconst_3
    ; iconst_2
    ; bipush 19
    ; invokestatic "Point.<new>(II)QPoint;
    ; vastore
```

Example: method invocation

All methods on value types are resolved statically. The invokestatic instruction would be sufficient (as a bytecode syntax) for invoking both static and non-static methods of a value type, but for clarity of exposition we will use a new invocation instruction `vinvoke` for non-static methods of value types.

```
class Point {
    public final int x;
    public final int y;

    public double getR() {
        double dx = x, dy = y;
        return sqrt(dx*dx + dy*dy);
    }
}

Point p = ...
    ; vstore 1
double r = p.getR();
    ; vload 1
    ; vinvoke Point.getR()D
    ; dstore 2
```

Example: boxing

When a value is assigned a variable of type `Object`, its box type, or an interface type which it implements, it should be autoboxed by the compiler. For example:

```
Object[] array = ...
Point p = ...
array[1] = p;
    ; aload 1    // array
    ; iconst_1   // index
    ; vload 2    // p
    ; vinvoke Point.<v2a>()LPoint;  // box p
    ; aastore
Point p = ...
    ; vstore 1
Point.__BoxedValue bp = p;  // descriptor LPoint; not QPoint;
    ; vload 1
    ; vinvoke Point.<v2a>()LPoint;  // box p
    ; astore 99  // continued below...
```

Example: unboxing

When a value is assigned from a variable of type `Object`, its box type, or an interface type which it implements, the reference is first checked to see if it is the corresponding box type, and then the value is unboxed:

```
Object[] array = ...
Point p = (Point) array[1];
    ; aload 1    // array
    ; iconst_1   // index
    ; aaload
    ; checkcast LPoint;
    ; invokevirtual Point.<a2v>()QPoint;  // unbox array[1]
    ; vstore 2   // p
Point p2 = bp;
    ; aload 99   // bp; see above
    ; invokevirtual Point.<a2v>()QPoint;  // unbox bp
    ; astore 5
```

A boxed reference can have value methods invoked directly on it:

```
double r = bp.getR();
    ; aload 99   // bp; see above
    ; invokevirtual Point.getR()D
```

```
      ; dstore 3
```

Example: flattened hash table

Because value types are pointer-free, they can be used to flatten some data structures by removing levels of indirection. Flattening not only removes dependent loads from critical paths, but also (typically) moves related data onto the same cache line.

Ideally, a well-tuned resizable hash table should be able to answer a query in not much more than two cache line references, one reference to consult the table header about table size, and one reference to probe an entry in the table. This ideal can be achieved if the table entries are flattened within the data array carrying the table, and this becomes possible with the help of value types.

```
class ToStringIntHashMap {
  private static final __AlwaysAtomic __ByValue class Entry {
    final int key;
    final String value;
  }
  private Entry[] table;
  private Entry getEntry(int k1) {
    int idx = hash(k1) & (table.length-1);
    Entry e1 = table[idx];
        ; aload_0           // this
        ; getfield table    // this.table
        ; iload_2           // idx
        ; vaload            // table[idx]
        ; vstore_3          // e1
    int k2 = e1.key;
        ; vload_3
        ; vgetfield key     // e1.key
        ; istore_4          // k2
    if (k1 == k2)  return e1;
    {... else slow path ...}
  }
}
```

The above code actually touches three cache lines, in order to extract an array length from the array header. Fixing that requires hoisting the array length up into hash table header, which is an interesting exercise in array design, but beyond the scope of this proposal.

> *More detail:* The above code sidles around the problem of distinguishing a default-valued entry (zero hash, null string reference) from a failure to find an entry. This is a typical issue in adapting pointer-oriented algorithms (in Java) to values. It could be patched in a number of ways. The simplest is probably to introduce a value which expresses an optional `Entry` (entry plus boolean), and return that from `getEntry`. Note that the ability to make an default (null) `Entry` value might be useful here, at the end of the slow path for `getEntry`. Default values are part of the Java's landscape for reasons given above, and we might wish to embrace them to get more use out of them.

The effect of `__AlwaysAtomic` here is to ensure that array elements values are read consistently, without internal races. Without that modification, on some platforms, struct tearing might lead to states where a value might be associated with the wrong key. This hazard comes from flattening; it is not present in the "pointer rich" version of the data structure.

Happily, most platforms will be able to implement atomicity for this data structure without extra cost, by means of 64-bit memory references. This assumes that the `value` component can be packed into 32 bits, which is usually the case.

> *More detail:* More generally, even a value of four components is likely to fit into 128 bits, and most hardware platforms can provide atomic 128-bit reads and writes, at a reasonable cost. After those options run out, there are others, but this proposal does not attempt to make a deep review of the implementation options. For larger values, say with five or more components, the costs for atomicity will go up sharply, which is why atomicity is not the default.
>
> *Alternate route:* General data structure layout control, of the type found in `C` and `C#`, does not appear to mesh well with Java's data model, but does show up when Java code needs to interoperate with native data structures. It is therefore a direct goal for Project Panama, which is seeking to expand Java's native interconnection capabilities.

Note that this example data structure is "hardwired" to work on `int`s and `String`s. A corresponding generic class would gain some benefits from the flattened array, but could not work directly on unboxed `int`s. A template-like mechanism to parameterize over non-references would apply nicely to this example, but such a proposal is out of scope here.

Example: comparison

As noted above, it is useful for values to implement interfaces. Here is an example of a value which supports comparison:

```
final __ByValue class Bignum implements Comparable<Bignum> {
  private final int intValue;
  private final BigInteger bigValue;
  public Bignum(int n) { intValue = n; bigValue = null; }
  public Bignum(BigInteger n) {
    if (fitsIn32(n)) { bigValue = null; intValue = n.intValue(); }
    else             { intValue = 0;    bigValue = n; }
  }
  public int compareTo(Bignum that) {...usual stuff...}
}
```

The automatically created box class will properly implement the interface, bridging to the given `compareTo` method. A `vinvoke` instruction can also directly invoke the `compareTo` method as written.

Here are some examples of calls to the interface function.

```
Bignum bn1 = __MakeValue(24), bn2 = __MakeValue(42);
    ; bipush 24
    ; invokestatic Bignum.<new>(I)QBignum;
    ; vstore_1          // bn1
    ; bipush 42
    ; invokestatic Bignum.<new>(I)QBignum;
    ; vstore_2          // bn2
int res = bn1.compareTo(bn2);
    ; vload_1           // bn1
    ; vload_2           // bn2
    ; vinvoke Bignum.compareTo(QBignum;)I
    ; istore_3          // res
Comparable box = bn1;
    ; vload_1           // bn1
    ; vinvoke Bignum.<v2a>()LBignum;
    ; astore_4          // box
res = box.compareTo(bn2);
    ; aload_4           // box
    ; vload_2           // bn2
    ; vinvoke Bignum.<v2a>()LBignum;
    ; invokeinterface Comparable.compareTo(Ljava/lang/Object;)I
    ; istore_3          // res
```

```
res = bn1.compareTo((Bignum)box);
    ; vload_1           // bn1
    ; aload_4           // box
    ; checkcast Bignum.__BoxedValue
    ; invokevirtual Bignum.<a2v>()QBignum;  // unbox
    ; vinvoke Bignum.compareTo(QBignum;)I
    ; istore_3          // res
```

As with classes, there is often an option to make a direct, non-interface call (`vinvoke` in this case) to the method which implements the interface.

Note that the interface takes an erased `Object` parameter, as required by the current rules of Java. In the context of the value type, we allow the type parameter (`T` in `Comparable<T>`) to bind to the value type `Bignum` and thus bridge the box method through to the value method `compareTo`. The language rules which govern this are not yet defined in detail.

We expect every value to implement an ad hoc interface or abstract super type that looks something like this:

```
interface __Objectable<ThisType> {
  default String toString() {...}
  default int hashCode() {...}
  default boolean equals(ThisType) {...}
}
```

Open questions

Many questions remain, including the ones raised above in passing.

Here are a few more questions that need answering:

- *Subtyping.* Are the proposed restrictions too constraining? Does it make sense to create a new root type for values, or to support abstract value super-types for factoring? This seems logically possible but with unpleasant complexity and irregularity.
- *Reflection.* What do value types look like under reflection? What is the reflected form for the type described by `QFoo;` and `LFoo;`, respectively? And what are their bytecode names, as used by `CONSTANT_Class`? Exactly what fields and methods are reported by their `java.lang.Class` metaobjects?
- *Erasure vs. specialization.* What is the policy for deciding when the `LFoo;` form of an argument or return type is needed, as opposed to the preferred `QFoo;` form? Do boxes operate only on boxes, and values only on values? What is the syntax to explicitly request the non-default type in Java code? Which of the "obvious" bridge methods do we autogenerate, and when (statically or dynamically)?
- *Migration compatibility.* How will we support binary compatibility for migration? How will we support pointer operations like nullability, identity comparison, and identity hash code?
- *Componentwise methods.* These are are reasonable default but are not always the right answer. For example, an overly chatty `toString` method can drastically violate encapsulation of secret value components. What is the mechanism and policy for generating componentwise methods for printing, comparison, hashcode, etc.? How do we get the right mix of opt-in vs. opt-out?
- *Enhanced variable operations.* CAS and similar operations are addressed as part of a separate feature exposing CAS operations directly (JEP-193.) We expect

values to mesh into such proposals along with references and primitives, for the most part.

- *Value/primitive convergence.* Can we integrate primitives into the values family? `__ByValue class int { ... }`? Can we get to a world where primitives are just predefined value types?
- *Fieldwise side effects.* Why are value fields always final? Why can't I side-effect just one field in a value? I want all the efficiency of "by value" and all the mutability of "by reference"! This is a long conversation. There are logically consistent ways to provide such things which, on balance, appear to be dismayingly complex. Java will never be C++, though there are proposals to improve its interoperation with C++. As noted above, values will certainly play a part with such interoperation.
- *Generics over values.* It is persistent irritant that generics do not work over primitive types today. With value types, this irritation will increase dramatically; not being able to describe `List<Point>` would be even worse. Still worse would be having it mean "today's list, containing **boxed** points". Supporting generics over primitives and values, through *specialization*, is the topic of a separate investigation.
- *Array values.* The space of aggregates forms a 2×2 matrix: *{identity, identityless}* × *{homogeneous, heterogeneous}*. Objects and arrays fill out the "identity" row; values fills in the (identityless, heterogeneous) box; the remaining box could be described as "arrays value", which might act as a suitable replacement for "varargs".

Conclusion

Along with the questions, we believe we have enough answers and insights to begin prototyping value types, and verifying our thesis. That is, we think it quite likely that a restricted class-like type can be made to look enough like a primitive to be worth adding to the language and VM.

Acknowledgements

This draft has deeply benefited from the comments of many people. Outside of Oracle, conversations with Doug Lea, Karl Taylor, Dan Heidinga, and Jeroen Frijters have been especially helpful.