L-World Value Types

Created by Karen Kinnear, last modified just a moment ago

L-World Value Type Terminology:

- Reference types: object class types, value class types, array types, interface types
    - represented by LFoo; signature for maximum backward compatibility, thence the name L-World
- value class type: defines a class whose instances are identity-less and immutable
- object class type: neither a value class type nor an array class type

Goals and Assumptions

Goals

Functional

- Add value classes which have no identity commitment and have immutable instances
    - allowing optimizations such as flattening when contained in a variable
    - Perform quickly and potentially use less javaheap memory
- support
    - value classes in instance and static fields
    - value class arrays
    - value class methods and method invocation
    - value classes inherit from interfaces with default methods

Migration and Compatibility

- Existing interfaces should be implementable by both object classes and value classes - without requiring recompilation
- Existing code should be able to handle both object class and value class dynamic arguments - without requiring recompilation
- Migration:
    - object class -> value class migration: for value-based classes
        - author must opt-in: by declaring in source (language policy) - requires compilation
            - any existing class that meets the requirements could become a value class, value-based classes are candidates (with additional restrictions)
        - Client challenges with object class -> value class migration for value-based classes with separate compilation
            - For a given method argument or return:
                - caller, callee, actual type: caller and callee may each assume object class or value class, only the actual class when loaded gives the actual class type
                - prior to loading the type, existing code may pass a null and we do not want to have to preload all classes for all signatures
            - fields:
                - client, declarer, actual type: client and declarer may each assume object class or value class, only the actual class when loaded gives the actual class type
    - Sample test cases:
        - migrate Optional, have Stream interface continue to work with its subclasses unchanged

Risks

- Customer Compatibility Risks
    - Existing code that takes an argument that is an Object or interface, which expected object classes and is

passed value classes may see unexpected results
        - use of if_acmp_eq/ne without subsequent .equals() call
        - attempts to synchronize on an argument which is dynamically a value class, will throw an exception
- Performance Risks
    - Can we get the performance we need for value types without performance loss for object types?

Non-Goals

- No support for value class > object class migration for classes that do not currently meet valuebased class restrictions
    - any client that attempts to create an instance of an existing object class via "new/dup/<init>" that has migrated to be a value class will fail
    - Brian pointed out that if you have no separate compilation issues, then you could migrate other object classes to value classes
        - This assumes that java compiler will catch incompatibility issues such as
            - "new" usage
            - identity assumptions
            - immutability assumptions (including use of setAccessible())
        - This assumes that the opt-in author is aware of all uses of a given type - which is not something we can actually check
    - Karen: if you have no separate compilation issues, you can change the name and guarantee complete coverage, so we don't need to provide migration on a non-guarantee
- No support for value class -> object class migration
- Primitives as value types - is a future phase, not part of LWorld value types

Assumptions of L-World model

1. New root: java.lang.Object - for all object classes and value classes

- no separate root for value classes

2. Value Type characteristics:

- value-based class characteristics:
    - final
        - no subclasses
    - shallowly immutable (unmodifiable instance fields) (language may appear to update, but actually creates new instance underneath) (may contain references to mutable objects)
    - no identity commitment:
        - have implementations of equals, hashCode, toString computed solely from state (not from identity)
        - equals solely based on equals() (not on ==)
        - freely substitutable when equal, no visible change in behavior if equals()
        - unpredictable results if sync, identity hash, serialization, ...
    - no non-private constructors: instantiated through factory methods, no identity commitment
- additional characteristics:
    - Nullability proposal:
        - A class declaring an instance field can declare it as non-nullable and therefore potentially flattenable in the declaring class
            - Non-nullable is a property of the field, not a property of a value class
            - Only a value class may be stored in a non-nullable field today
                - note: in future we may explore non-nullability for non-value types. This would not make them flattenable.
            - clarify: flattenable, JVM makes per-implementation/per-platform decisions about actual flattening
            - you can NOT individually address and update flattened fields
        - A class declaring an instance field containing an array can declare the array FieldType as non-nullable (in the classfile) and thereby potentially flattenable
    - no boxing
        - no default box, no boxing at all
        - all fields for an instance in the heap will be contained in the heap, whether through a reference (indirection) or through flattening in the container

        - all arrays in the heap will have every index either contain a null, a heap allocated reference or a value type flattened in the container
        - if you want identity, create an object instance storing a value type field
        - note: a value type does NOT have a box in this model. In future we may need to special case primitives as value types and java.lang.Integer etc. but that will need corner case handling.
        - A given runtime type will either be an object type or a value type, determined when the class is loaded
        - There is NO such thing as a conversion operation, no heisenboxes, no accidental identity
    - support interfaces
    - java.lang.Object as only superclass (so not all value-based classes will meet the migration requirement, although current JDK value-based classes do)

Expected Behaviors for Value Types

JDK java.lang.Object Methods

- final wait/notify/etc: if isValue(): throw exception (IMSE or ICCE? - see open issues)
- final getClass: normal behavior (no ambiguity with no boxes)
- toString: nothing special
- clone: nothing special
- finalize: ICCE, note: no one should ever call it (but old code will)
- equals: if isValue(): JDK component-wise comparison
- hashcode: must work with equals

Java level APIs

- Class.isValue()
- System.isSubstitutableValue(), System.getSubstituteableHashCode() (to wean folks off of System.identityHashCode for values)
- System.identityHashCode() - should not work for values
- setAccessible() does NOT give you the ability to write to value instance

LWVT bytecodes vs. JVMS 9

- special handling:
    - if_acmpeq/if_acmpne: false/true if either is a value instance. They should fall back to .equals
- needs dynamic different handling:
    - aaload: no semantic change, implementation based on element type and properties (e.g. non-nullable, flattened, atomic, ...)
    - aastore: today throws NPE if arrayref is null, change: if non-nullable array and passed null: NPE. no other semantic change, implementation based on element type and properties (e.g. non-nullable, flattened, atomic, ...)
    - areturn: no semantic change
- exception if wrong:
    - putfield: field of a value class: IllegalAccessError (already throws), null to ACC_FLATTENABLE: NPE (already throws due to null object ref)
    - monitorenter/exit: objectref instance of value class : IllegalMonitorStateException (already throws)
    - new: InstantiationError if symbolic reference to value class (already throws for existing interface, or abstract class)
    - **withfield**: field of object class type: ICCE
    - **defaultvalue**: symbolic reference resolves to an object class: InstantiationError if
- unchanged or already implemented (in MVT) or should fall out:
    - aload/astore: handle object class or value class
    - getfield: handle field of an object class or value class, handle field that is an object class or value class dynamically
    - anewarray/multianewarray: handle object class or value class, the type of the reference is resolved before array creation already
    - athrow: always an object class (subtype of Error) - unchanged
    - invoke*: handle object class vs. value class arguments and return values

    - checkcast/instanceof: keep current behavior
    - ldc: should fall out
    - ifnull/ifnonnull: no change
- aconst_null: only return object class
- **defaultvalue**: only returns an initialized value class (initialized to the default value)

Design Issues

Open Design Issues

Nullability and migration

Migration of an object class to a value class (e.g. value-based-class) and nullability expectations

- Goal is to allow as much existing code to work as possible in the face of migration
    - without requiring preloading classes for all fields
- Proposal: Have the declarer of an instance field declare flattenable (prototype syntax TBD) for a field or array if it wants to allow flattening
    - cases:
        - Legacy declaration of LFoo; field
            - field is nullable in this container
            - it is ok to write null, it is ok to read null, field is initialized to null
            - Foo continues to be lazily loaded
            - when Foo is loaded, regardless of whether it is actually an object class or a value class, the behavior does not change
            - in the java heap, an instance field will always be also stored in the java heap, whether it is a reference to an object class or a reference to a value class
        - Flattenable declaration of LFoo; field or [LFoo; array
            - Foo is pre-loaded (for a field, before completing loading of the declaring class, for an array before creating the array - unchanged)
            - when Foo is loaded, if it actually is a object class, throw an exception (e.g. ICCE) on the declaring class
            - If Foo is actually a value class
                - attempts to store a null fail with a NullPointerException
                - fields are initialized to the default value, so you can never read a null
                - This allows the JVM implementation to flatten the field if it deems it beneficial
                - In the java heap, a field will always be also stored in the java heap, whether it is a reference to a value class or the value fields are flattened in the container
- Proposal: only detect nullability errors when we publish a value type to a field declared as ACC_FLATTENABLE
    - aastore - do not allow storing a null to a non-nullable array: throw NPE
    - putfield, withfield for a field declared as non-nullable: throw NPE
- Note: we do not perform null checks for:
    - Local variable table/expression stack
    - argument passing, argument return
- Note:
    - Future may want to explore non-nullable non-value type fields and arrays

Nullability Handling and generics over value types

- Need to think more closely about how value types will migrate to support generics over value types
- With the current nullability proposal, we get a free migration to allow existing generics to work with value classes
    - with no source changes
    - and no flattening optimizations in current classes for any fields exposed via APIs that could pass in "null"
- However, if an existing parameterized class chooses to declare a field as non-nullable
    - chooses to declare a field as flattenable for a value type
    - (potential future) for non-value type fields and arrays
    - this changes the behavior of the class and APIs and will come as a surprise
- Need to explore ways to catch the surprise at compilation time

Where do we need explicit value class information in the constant pool?

- Proposal:
    - there is no value-class information in the constant pool
    - constant pool uses CONSTANT_Class_info for both object classes and value classes
    - Descriptors all use the LFoo; signature format.

How would we represent value class information in the class file?

- ACC_VALUETYPE for Class modifier
- ACC_FLATTENABLE for Field modifier

Identity: monitorenter/exit handling

- What exception should we throw if we use monitorenter/exit/wait/notify* for a value type? IllegalMonitorStateException or IncompatibleClassChangeError?

Where does the Java language need to distinguish a value class? vs. what can javac do for you?

- Declaration of a class as a value type (translates into classfile with ACC_VALUETYPE class attribute)
- instance field declaration
    - Declare a field element as non-nullable which allows flattening (e.g. translates into classfile as ACC_FLATTENABLE on the Field_info)
        - default for field - nullable unless declared in source
        - default for an array - non-nullable if the array element is a value type unless declared in source?
            - or do we want the default to be the same for fields and arrays? i.e. nullable unless declared in source
- Would javac want to generate isnonnull checks before storing to a non-nullable field or array element so as to reduce NullPointerException throwing?
    - instance creation
        - defaultvalue/withfield vs. new/dup/init mechanism
- Restrictions on Value Types:
    - class must be final
    - java.lang.Object as only superclass (empty superclass, javac fills in)
    - no <init>
    - It is invalid to declare a field or array element as non-nullable if the actual type of the field or array element is an object class type
        - this will also be caught at runtime by the JVM for separate compilation

Array Subtyping

- Open Question: Specifically are all arrays of value types subtypes of Object[]?
- Proposal:
    - initial prototype should assume this is true and revisit if this is too expensive from a performance standpoint

Value Class and top level vs. inner class

- Open Question: Can an inner class be a Value Class or only a top-level class?
    - Yes for static inner classes
    - For instance inner classes there might be implicit fields from an enclosing class
    - TODO - discuss in more detail

Java Language questions

- Must a value class not declare a superclass? Or should it declare java.lang.Object explicitly?
    - Proposal: NOT declare a superclass to allow evolution
- Where can withfield be used?

    - Proposal:
        - In any method declared in the value class itself or declared in a nestmate
    - alternative considered:
        - in a value class factory:
            - a static method declared in a value class with a modifier (lworld prototype proposal: __ValueFactory in source)
                - the return type of the static factory method must be identical to the value class of which the static factory is a member
                - inside the factory: value instances are created with the invocation of __MakeDefault ValueType()
                - it is ok to have more than one factory
                - only the factory methods can use defaultvalue and withfield bytecodes
                - you can have additional factories that take arguments
        - client (lworld prototype) invokes __MakeDefault ValueType()

Are static fields candidates for ACC_FLATTENABLE?

- Cons:
    - There is very little gain to any flattening for statics
    - There is a significant loss forbidding constructs at the language level due to class circularity issues
    - Precedent for no parameterized types in static fields
- Pros:
    - Not want to limit this from the JVM side
- Proposal is:
    - Allow this at the language level in the initial prototype

Resolved Design Issues LWVT

Q:Do Value classes support superclasses other than java.lang.Object?

1. note: value classes have no subclasses
2. for now - value class has only jlO as superclass, may be extended in future (see if that would break any optimizations after JIT working)
    - note: if we were to change this - ANY LFoo; passed as an argument (not just Object and interfaces) would require dynamic checking of object class vs. value class
    - In addition, there would be interactions in circularity checking between superclasses and non-nullable fields.

Q: acmp behavior options:

- failing: return false <- propose for try 1
- throw exception
- field-equality using ucmp as "substitutable" - field-wise comparison
    - general bit equality including floating point
    - may need to recurse on values buffered
- A: LWorld1: if >= one operand isValue(): if_acmpeq -> false, if_acmpne -> true
- John's mental model: even if both operands are values, "NaN-like" condition - still return if_acmpeq->false, if_acmpne->true

Q: What should the verifier be required to check relative to value classes?

- Goals:
    - ensure no insecure behavior based on type mismatches
    - minimize eager class loading

- Proposal:
    - verifier could continue to perform checks such as reference vs. primitive, and isAssignable checks, including value classes as well as object classes as references

    - Therefore bytecodes at runtime would explicitly check and throw exceptions if they only apply to value classes or object classes
        - note: if passed an LObject or interface we need the dynamic check anyway in many cases
- Alternatives Considered:
    - verifier could perform checks for bytecodes that require value class vs. object class
        - concerns: this would need to be delayed until the classes were loaded
        - for loaded classes such as super types, value types fields or isAssignable checks, some classes are already loaded - concern - this would throw errors at randomly different times
        - there are very few bytecodes that require an explicit value class or object class - defaultvalue, withfield, putfield, monitor enter/exit, new, <init> invocation

Q: Migration value class->object class support?

- Customers will try migrating type Foo from value class to object class, by changing the source
- A: Need to ensure we catch failures - this is not supported
- challenges:
    - field declaration of a non-nullable field should fail when loading an object class when a value class was expected
    - client instance creation: defaultvalue for value class will fail with an object class

Q: Circularity handling for Field types?

- Need to explore implementation issues relative to accurate ClassCircularityError vs. StackOverflowError.

Q: Do we need a java API for isFlattened (for a reflection Field or Array)

- John: Let's NOT provide that information. Let's have flattening be transparent from the java level.

Q: Do we need a java API isComponentValue?

- For now, let's skip this. The information is available via getComponentType.isValue().

Is there meaning to a value interface or an abstract value class?

- No. Since a value class can have no subtypes, there appears to be no meaning to a value interface or an abstract value class

How is java.lang.Object evolving?

- LObject as "more of an interface"
    - no (inheritable) fields allowed
- LOBject as "not an interface"
    - instantiable
    - allows methods that are not public/not private
    - already has a constructor - do we need a root without one?
    - order of method searching - selection searches classes/superclasses before searching superinterfaces
        - resolution searches java.lang.Object before searching super interfaces
        - overriding - j.l.Object methods are overridden by class methods but never by interface methods
        - equals and hashCode are overridable, so I have been assuming that value types can override them
            - to me this implies that the JVM/JIT can NOT optimize away calls to Object.equals (or at least not any that are overridden)
- For all interfaces and LObject, we can no longer assume identity, but must check the actual runtime subtype
- An LObject or LInterface variable can be set to null, which implies not a value instance

What is the root type?

- Proposal: java.lang.Object is the global root type is intended to help with migration, so that code that today defines a field or parameter as LObject (including erased generics) will transparently work with value types
    - If we believe this is possible, then we need to keep LObject as a super type of all value types (note: it in itself could have another super-root if needed)
    - Alternative: new root of I$Object which is an interface, super interface of all types
        - todo: figure out how existing interfaces could work with this one -
        - note: this seems to be here to clean up interface handling,
        - concerns: it isn't needed for value types
        - concerns: it breaks the ability to pass a value type for a reference which currently expects LObject which is needed for value-based-class migration

Do value types need to be able to override java.lang.Object.Equals?

- Proposal: yes

Why can't enums be value classes?

- Backward compatibility issue
    - enums have identity
    - enums have java.lang.Enum abstract class as super-class, not java.lang.Object
    - there is no clear default value
    - enums have mutable fields

Should we allow ACC_FLATTENABLE for an object class

- Out of scope for this project.
- The challenge is instance initialization
- Object classes are created via new, dup, <init>. The new bytecode initializes all instance variables of the new object to their default initial value.
    - The default initial value for an object class is null
    - Once <init> if it exists is complete, the instance class is considered initialized, and there is no requirement that <init> actually exist or update each instance field.

Should we allow any object class to migrate to come a value class?

- Migration is restricted to value-based classes because
    - they already assume no identity
    - they only have private constructors, so there is no existing code that executes new/dup/<init>

References

- http://cr.openjdk.java.net/~dlsmith/values-notes.html
- http://cr.openjdk.java.net/~fparain/L-world/L-World-JVMS-3.pdf

Like Be the first to like this

Like No labels