L-World Value Types

Created by Karen Kinnear, last modified just a moment ago

L-World Value Type Terminology:

- Q-Type: value type
- R-Type as heap object type - question: do we need this?
- U-Type: union of Q-Type or R-Type
- L-Type - in LWorld - L-Type is U-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:
    - LType > QType 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 issues with LType > QType migration for value-based classes
            - any existing code that passes null where an LFoo was expected, but a QFoo does not allow
                - e.g. new code expects m(QFoo), legacy caller with signature m(LFoo) passes in a null
                - issues with fields and methods

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

Non-Goals

- No support for LType > QType migration for classes that do not currently meet valuebased class restrictions
    - any client that attempts to create an instance of an existing LType via "new/dup/"init?> that has migrated to be a value class will fail
- No support for QType -> LType migration
- Primitives as value types - is a future phase, not part of LWVT

Assumptions of L-World model

1. New root: LObject - for all object classes and value classes

2. Value Type characteristics:

- value-based class characteristics:
    - final
        - no subclasses
    - shallowly immutable (unmodifiable) (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 ==) ** TODO
        - 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:
    - 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
        - 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 default box (buffer, not box)
        - 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.
    - support interfaces
    - java.lang.Object as only superclass (so not all value-based classes will meet the migration requirement)

Expected Behaviors for Value Types

JDK java.lang.Object Methods

- final wait/notify/etc: if QType: throw exception (IMSE or ICCE? - see open issues)
- final getClass: normal behavior (no ambiguity with no default box)
- toString: nothing special
- clone: nothing special
- finalize: ICCE, note: no one should ever call it (but old code will)
- equals: if QType: JDK component-wise comparison: open question: does this need to be overridable? Or could this be an intrinsic?
- hashcode: TBD

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

JVMS general

- Propose using term reference to cover both object classes and value classes
- Format checking: For a Value Class, the value of the super_class item must ... representing the class Object.

LWVT bytecodes vs. JVMS 9

- special handling:
    - if_acmpeq/if_acmpne: false/true if either is a Qtype. They should fall back to .equals
    - ifnull/ifnonnull: as if acmp vs. Null: false/true if QType
- needs dynamic different handling:
    - aaload/aastore: handle LType or QType dynamically (both flattening and nullability)
    - areturn: handle LType or QType dynamically
- exception if wrong: ICCE
    - putfield: QType exception: ICCE
    - monitorenter/exit: exception for QType: ICCE
    - **withfield**: exception for LType: ICCE
    - **defaultvalue**: exception for LType: ICCE
- unchanged or already implemented (in MVT) or should fall out:
    - aload/astore: handle LType or QType (always indirection, should fall out)
    - getfield: handle LType or QType dynamically (already implemented)
    - anewarray/multianewarray: handle LType or QType dynamically (already implemented)
    - athrow: always LType (subtype of Error) - unchanged
    - invoke*: handle LType or QType dynamically (should fall out)
    - checkcast/instanceof: keep current behavior
    - ldc: should fall out
- new, aconst_null: only return RTypes
- **defaultvalue**: only returns a QType

Design Issues

Open Design Issues

Nullability and migration

Migration of an L-Type to a Q-Type (e.g. value-based-class) changes 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 Non-nullable for a field or array element 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
            - when Foo is loaded, regardless of whether it is actually a L-Type or a Q-Type, the behavior does not change
        - Non-nullable declaration of LFoo field
            - Foo is pre-loaded (for a field, before completing loading of the declaring class, for an array before creating the array)
            - when Foo is loaded, if it actually is an LType, throw an exception (e.g. ICCE) on the declaring class

                - If Foo is actually a QType
                    - 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
- Proposal: only detect nullability errors when we publish a value type:
    - aastore - do not allow storing a null to a non-nullable array: throw NPE
        - same issue if an argument is an object-array, but the dynamic instance is a value-array
    - putfield, withfield, putstatic for a field declared as a QFoo: throw NPE
- Note: we do not perform null checks for:
    - Local variable table/expression stack
    - argument passing, argument return
- Open Issue:
    - checkcast/instanceof behavior given a null reference
        - Goal 1: have a way to determine if you have a null when you expect a value class
        - Goal 2: enable legacy code to work even when the type has migrated to be a value class
        - Behavior today:
            - checkcast allows a null object ref and does not resolve the symbolic reference
            - instanceof allows a null object ref and does not resolve the symbolic reference

Where do we need explicit Q-Type information in the constant pool?

- Proposal:
    - there is no Q-Type 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 Q-Type information in the class file?

- ACC_VALUETYPE for Class modifier
- ACC_NON_NULLABLE for Field modifier

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

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_NON_NULLABLE 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?
- 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
        - vefaultvalue/withfield vs. new/dup/init mechanism
- Restrictions on Value Types:
    - class must be final
    - java.lang.Object as only superclass
    - 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[]?
- We think it is possible to make this subtype check work

Value Class and top level vs. inner class

- Open Question: Can an inner class be a Value Class or only a top-level class?

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 withheld be used?
    - In the value class itself - anywhere?
    - in the value class itself - in specifically tagged methods? (factories, setters?)
    - in nestmates?

Are static fields candidates for ACC_NON_NULLABLE?

- NO
- 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

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 - QType 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 LType passed as an argument (not just Object and interfaces) would require dynamic checking of object class vs. value class

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 is a QType: 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: null handling for acmp

- as if acmp vs. NULL
- if operand is a QType: ifnull -> false, ifnonnull->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 QType->LType support?

- Customers will try migrating type Foo from QFoo to LFoo, by changing the source
- A: Need to ensure we catch failures
- 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 QFoo will fail with an LFoo
    - caller method signatures:
        - if a method descriptor contains information about value class vs. object class, i.e. expects QFoo, passing LFoo will fail

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

References

- http://cr.openjdk.java.net/~dlsmith/values-notes.html

No labels