L-World Nullability

Created by Karen Kinnear, last modified just a moment ago

Proposal

- We can offer new value types the language benefits of catching nullability problems at compile time
- And offer the ability to migrate value-based classes to value types
- If we continue the JVM design of maximum leniency in handling nulls, based on container-based flattenability
- And - if we add for javac, the ability to distinguish between new value types and value-based classes migrating to value types
- Summary: Give javac the ability to be strict with new value types, and to choose appropriate leniency/strictness for migrating value types

Use Cases

- new value types
- backward compatibility:
    - live value types treated as Object, Interface or [Object, [Interface
    - so usable by existing code for fields, method arguments, method bodies
    - this is a key advantage of L-World design with Object as root
- forward migration: value-based classes that migrate to value types
    - note: this is a bounded problem - there are a limited number of known value-based classes and probably a limited number of candidates for migration

Migration and Nullability

- Migration includes multiple steps
    - value-based class migrations to value class: e.g. FIELDC
    - Container choosing flattenable for a field that is now a value class: CONTC declares FIELDC
        - Container needs to opt-in to make this choice
        - Container may need to change method contents
        - Container may need to change APIs
    - Client of container: CLIENT
        - Assumes null is a valid value to write to existing field CONTC.FIELDC
        - Uses existing APIs, assumes null is a valid argument/return value
    - Subclass of container: SUBC
        - Assumes null is a valid value to write to inherited field CONTC.FIELDC
        - May need to change its own method contents
        - May need to change its own APIs
    - Client of subclass: SUBCCLIENT
        - Assumes null is a valid value to write to existing field SUBC.FIELDC which is inherited from CONTC
        - Uses Subclass existing APIs, assumes null is a valid argument/return value
- separate compilation can occur for each of the above classes
    - obviously migrating class change must be first to matter, with this model, container needs to be second
    - others can recompile separately in any order
    - so class file version is not sufficient to indicate awareness of a value-based class migrating to a value type, or of field flattening
    - there are multiple players, often in a single bytecode - e.g. SUBCCLIENT putfield SUBC.FIELDC which is inherited from CONTC.

Goal of the chart is to describe the multiple potential steps of migration, with worst case having separate recompilation of each class.

- The chart tries to highlight the touch points for javac, i.e. recompilation - so that opportunities to choose javac behavior.
- key: "null" means - expects nullability

Migration: expect null	**today**	**vbc→value class FIELDC**	**container: flattenable CONTC**	**recompiled client: CLIENT**	**recompile subclass: SUBC**	**recompile client of subclass: SUBCCLIENT**
FIELDC	null	Value Type	Value Type	Value Type	Value Type	Value Type
CONTC	null	null	Flattenable. Maybe API changes? JAVAC? Warnings?	Flattenable	Flattenable	Flattenable
CLIENT	null	null	null	JAVAC? warnings?	null	null
SUBC	null	null	null	null	JAVAC? warnings?	null
SUBCCLIENT	null	null	null	null	null	JAVAC? warnings?

JVM Perspective - leniency

- The JVM needs to be able to handle all the use cases above
- As well as non-java languages
- backward compatibility
    - L-World design proposal of L<>; signatures which do not distinguish value types from identity objects allows backward compatibility
- migration implications
    - Container-based flattenable/non-nullable allows JVM to support migration (and languages that want other options)
    - Nullability is ok for LVT, operand stack, argument passing
    - Nullability is only checked when published to a flattenable container
    - JVM does not need to know about migration, it needs to be lenient to allow all the migration steps to work in whatever order they occur
        - i.e. I do not think it helps the vm know that a given value type has been migrated or not
        - there are multiple class files involved in key interactions - so the jvm need to be as lenient as it can

- bytecodes (including instanceof and checkcast) therefore need to retain the existing behaviors relative to null except explicitly for publishing
    - putfield - NPE if attempting to store null to a flattenable field
    - aastore/aaload: - NPE if attempting to store null to a flattenable value type array/ if loading null from a flattenable value type array
        - note: initial prototype assuming all value type arrays are flattenable

Javac perspective - new proposal

- The ideal world for javac is to make all value types flattenable by default, whether in fields or arrays
    - to be able to give early warning at compile time for nullability
    - to be able to insert null checks in the bytecodes

Proposal: by allowing javac to distinguish between newly created value types and migrated value types, javac can choose different strictness/leniency options

- new value types
    - javac could make all new value types flattenable by default, whether in fields or arrays
    - because value types are final, and we disallow conversion from a value type to an identity object:
        - when javac knows it is always statically dealing with a new value type
            - disallow assignment of null to a new value type
            - disallow casted to null or comparing null to a new value type
            - disallow comparison with == or != for statically known new value types
        - and javac can inject null checks before bytecodes when it knows it is always dealing with a new value type - e.g.
            - withfield, aastore
            - checkcast, instanceof
    - javac communicates to the JVM by setting fields to ACC_FLATTENABLE for all new value types
        - javac could choose to offer customers an option (as could any language)
        - this proposal is based on an understanding that java would prefer to make this simpler from a user point of view

- migration of value-based-classes to value types
    - recommend that migration to a value type requires communication with the language - e.g. an annotation
    - For migrating value types
        - make migrated value types NOT flattenable by default
        - give the container author a chance to opt-in
    - javac now has a choice of how to handle warning and byte code generation
        - choice of strictness/leniency
        - potential to ease migration by handling clients and subclasses of migrating types differently (independent of flattenable opt-in by author)

Joint Effort

- Brian has stated a goal that where possible, javac issue a warning where runtime would throw an exception, as is done today for ClassCastExceptions
- For new value types, I believe that model is possible
- For migrating value-based-class to value types I believe we need a joint effort

- Identity handling
    - value based classes are "supposed" to already not assume identity, so we expect fewer surprises there
    - the proposal for value based classes is to offer multiple levels of assistance
        - JDK 11
            - add a runtime flag to catch identity issues with value based classes
        - javac
            - javac would disallow
                - calling java.lang.Object methods that do not support identity
        - runtime
            - core libraries
                - java.lang.Object methods such as wait*, notify* - will throw an exception if operating on a live value type
            - JVM
                - JVM would throw an exception e.g. sync - if operating on a live value type
- nullability handling for migration
    - value based classes support nullability today, so we expect to run into this issue far more often
    - propose multiple levels of assistance
        - JDK 11
            - we could add a runtime flag to catch nullability issues with value based classes
        - javac
            - this is a language decision
            - a couple of thoughts
                - I don't know if it helps to know if files are recompiled together or not, but it might
                - John suggested that if the container of the value-based class is in the JDK, does javac know when the JDK "is the latest"? (-target? -release?)
        - runtime
            - JVM would throw NPE at publishing

Potential Experiments

- Recommend experiments with core libraries to migrate some value-based classes to value classes
    - In particular we need experiments with client, subclasses and clients of subclasses that are separately compiled
- initial prototype is assuming all arrays of value types are flattenable - we will want experimental feedback for migrated value types, the declarers of their arrays, clients and subclasses which are independently compiled
    - we should consider later finding out the cost of not requiring all value type arrays to be flattenable (in case it is not too onerous), so as to ease migration for arrays