Retrospect Implementation Notes

* Object initialization
  + Native objects
    - Generate no-argument constructor
    - Technically works – VM doesn’t verify final fields are assigned
    - Not ideal because VM might be free to cache values, ignore assignments, etc.
    - Could also use FieldMapMirrors instead – may be faster than reflection-based NativeObjectMirrors
    - Should refactor mirror interface to use constant pool offsets instead of field names
      * Can dynamically generate native mirrors that use a switch to select the right field
    - Note: two different kinds of “native” objects
      * Instantiated inside MirageClassLoader classes – all fields lifted to mirages already
      * Instantiated outside – all fields have to be lifted as they are accessed
    - Direct datatype implementation may still be fastest since all fields must be of type ObjectMirror?
      * Maybe not, can still separate primitives from references
  + Mirage objects
    - Add extra mirror field to constructors, pass up super chain
      * Doesn’t handle reflection (i.e. Class#newInstance)
      * In fact, messes up reflection entirely since the parameter types are wrong
        + Need to advice reflection calls to hide changes in general – aligns with principle of mirror-based architectures
      * How to distinguish calls to new (where mirror should be instantiated) from calls to super (where mirror should be passed along?)
        + Both are just INVOKESPECIAL
        + Do dataflow analysis that respects difference between uninitialized and uninitialized this
    - Instead, two ObjectMirage constructors
      * Nullary for new statements – creates native mirror inline
      * One that takes a mirror for calls to ObjectMirage.make
      * Problem
        + In Java, anonymous inner class constructors will set fields before calling super
        + Bytecode in general can do this whenever it wants
* Static fields
  + ClassMirror interface for getting fields
  + Single ClassMirrorLoader instance per ClassLoader
  + Problem – can’t ClassLoader.loadClass() on non-public classes, even though it’s easy for many classes to get a hold of the Class instance by calling getClass() on an instance
* ClassLoading:
  + MirageClassLoader
    - Give it two class loaders for delegation
      * One for mirror implementations
      * One for loading original code (for transforming to mirage classes)
  + Mirrors turn out to be useful for bytecode verification/analysis
    - Type checking and analysis of mirror tree instead of real classes avoids class circularity errors
* Arrays
  + Each unique type generates a stub Java class that inherits ObjectArrayMirror
    - No instance methods no not necessary for the same reason as user classes, but needed for method overloading
  + Big issue: AALOAD/STORE opcode doesn’t specify expected element type the way method invocations do
    - Need to analyze bytecode to infer it!
    - Need bytecode version 1.6 or later - frame information required
      * No good, JRE only 1.5 even in later versions
* Special cases
  + All classes must extend Object, which declares toString()
    - Therefore all toString() methods must return actual java.lang.String instances
  + Object identity
    - How to define ==?
      * Could define ObjectMirror.sameObject(Object other)
      * Could allow mirages to be different, even if calling ObjectMirage.make() on the same mirror object twice
    - How to define hashCode()?
      * Supposed to be consistent across a single VM execution
    - How to define equals()?
  + Native methods
    - Enumerate replacements
      * Use AspectJ binaries even at this point (around execution)?
    - Automatically link all extra methods on ClassMirror according to idiom
    - Any missing methods are stubbed with a method that throws an exception
  + Reflection
    - Should intercept methods like Class.getFields() to interface with mirrors instead
  + Enums
    - Just remove the flag – no need?
* Optimization ideas
  + Cache consecutive field reads on “this” in same method?
    - No, not valid in light of concurrent writes
    - Maybe in specific well-understood/constrained cases
    - Or with clearer understanding of Java memory model