Value Types Consistency Checking

Optimizations in the JIT depend on Value Type Consistency.

Assumptions:
There is no migration from value types to identity types ever.

Value Types Consistency checking proposal:

In some cases, value types are eagerly loaded, in this cases, consistency checks are performed against the real type at the time these types are loaded:

- instance value fields declared with the ACC_FLATTENABLE flag set have their type pre-loaded during the loading phase. If the loaded class is not a value class, an ICCE is thrown
- static fields declared with the ACC_FLATTENABLE flag set have their type pre-loaded during the link phase, prior to preparation. If the loaded class is not a value class, an ICCE is thrown (post-LW1)
- methods declared or implemented by the current class can also use value types in their arguments or return value, these types are eagerly loaded during the preparation phase, and if any of these classes is not a value class, an ICCE is thrown
- Analogously to loader constraint checking, during preparation and selection cache creation, any method overriding needs to perform value type consistency checking between declarer of the overridden method and the declarer of the overriding method for any types declared as value types by either declarer.

In the cases above, the check is performed between the assumptions of the current class and the real kind of the type. If the current class doesn't make the assumption that a type is a value class, no pre-loading is done, and the type is assumed to not be a value class.

There are other cases where value types are used but not eagerly loaded, in those cases, consistency checks are performed at resolution time:

- at CONSTANT_Class resolution time, there will always be a check performed between the expectations of the caller and the real loaded type. This covers the following byte-codes: new, defaultvalue, anewarray, multianewarray, checkcast, instanceof, ldc. In all cases, at resolution, a consistency check is performed between the real kind of the type and the current class assumption. If a mismatch is detected an ICCE is thrown. If a mismatch between bytecode and value or identity type is detected an ICCE is thrown.
- when a field or a method from another class is resolved, a consistency check is performed between the assumptions of the current class and the assumptions of the resolved class, if a mismatch is detected, an ICCE is thrown. In addition, if putfield is performed on a value type or withfield on an identity type, and ICCE is thrown, although the verifier can catch this.

The checks for value type consistency should check both value types and array elements. For example, if caller-callee consistency is not checked for array elements, a caller that is unaware that a type, e.g. Point, is a value type can obtain an instance of a value type through an array read from a field or returned from a method.

Open issue 1: consistency checks for remote fields and methods are performed between the assumptions of the current class and the assumptions of the resolved class, not against the real kind of the type. This property is important to prevent loading of non value types that are not used (null being used instead). The drawback is that both classes could have wrong the assumption. For example: class V is a value class, class A declares a field of type V but assumes it is not a value class, class B accesses A's field and doesn't assume that V is a value class either. All consistency checks pass without throwing an ICCE. However, classes A and B can only use null when dealing with this field. Any attempt to create an instance of V, to resolve a field of type V or method with V in its signature from a class with the right assumption about V would trigger an ICCE.

Summary of Consistency Type Checking for Foo in ValueTypes attribute when Foo is not a Value Type

Foo in VTs attribute, Foo is NOT a VT	**ClassA: container/declarer**	**ClassB: caller Foo in VTs attribute**
local instance field, flattenable	preload: check vs. real mismatch declarer: ICCE	N/A
local instance field, not flattenable	No eager loading *1 ok: read/write null locally	
local method	< preparation: check vs. real mismatch declarer: ICCE	N/A
CP resolution (defaultvalue, new, anewarray, checkcast)	class resolution: real mismatch: ICCE	N/A
field access	field resolution: ClassA vs. ClassA - always passes *1: ok: read/write null locally instance creation/checkcast: vs. real: ICCE	field res: ClassB vs. Class A mismatch: ICCE *1 ok: read/write null instance creation/checkcast vs. real: ICCE
method invocation: caller vs. resolved method holder	method resolution: ClassA vs. ClassA - always passes *2	method res: ClassB vs. ClassA mismatch: ICCE *2
method overriding: resolved method holder vs. selected method holder	*5 preparation: selection cache creation: check if in VTs attribute for either one	
local static field, flattenable	< preparation: real mismatch: ICCE *3	*1

Consistency Checking Examples

In each of the caller-callee consistency checks we have three players:
  type in Question for these examples we use:
      Point - for real value type, Foo for not value type
  Caller: ClassA
  Callee: ClassB (for remote fields and methods, this is the declaring class)

Key question for examples which ensure caller-callee consistency is: what if the caller-callee are consistent, but neither one has the real information?

*1 Field Access: Foo in ValueTypes attribute, but not a VT

ClassA declaring a local: ClassA has Foo in ValueTypes attribute, but Foo is not a VT

- flattenable instance/flattenable static: fail real check at load/preparation ICCE

- non-flattenable (nullable) instance or static will not eagerly load Foo
    - ClassA read/write null from/to the local field succeeds
    - ClassA create Foo instance: CP resolution: caller vs. real Foo check: ICCE

ClassA declares non-flattenable local: ClassA has Foo in ValueTypes attribute, Foo not a VT
ClassB does NOT have Foo in ValueTypes attribute
ClassC has Foo in ValueTypes attribute

- ClassB read/write ClassA field will fail caller-callee: ICCE
- ClassB can create instance of Foo
    - If ClassB passed the instance of Foo to ClassC which has Foo in the ValueTypes attribute:
        - if ClassC.method specifies a parameter of Foo, ClassB:ClassC inconsistency: ICCE
        - if ClassC.method specifies Interface or Object, there will be no check
            - if ClassC casts Interface or Object to Foo, constant pool resolution will check ClassC vs. real: throw ICCE

ClassA declares a local: ClassA does not have Foo in ValueTypes attribute, Foo is not a VT
ClassB has Foo in ValueTypes attribute, tries to access ClassA.foo field

- ClassA attempts to create a new instance of Foo will succeed
- Class A attempts to read/write null/instance of Foo from/to the field will succeed
- ClassB attempts to read/write null/instance of Foo to/from an instance ClassA.foo will fail caller-callee consistency check: ICCE

*2 Method Invocation: Foo in ValueTypes attribute, not a VT

ClassA declares a local method: ClassA has Foo in ValueTypes attribute, Foo is not a VT

- Foo is a parameter/return value: eager load during Preparation will throw ICCE

ClassA declares a local method, ClassA does NOT have Foo in ValueTypes attribute, Foo is not a VT
ClassB has Foo in ValueTypes attribute, tries to invoke ClassA.meth

- ClassB tries to invoke ClassA.meth with Foo as parameter/return, caller-callee check fails: ICCE

*3 Statics - special handling

One goal is to allow defining a static field of the same type as the containing value class. In order to prevent class circularity errors, we need to NOT pre-load the flattenable static fields. If we load the static fields marked as flattenable at link time, prior to Preparation, we would have the size information needed to create he default value to fill in the static fields.
This removes the risk of circularity errors.

Requirement:
The default value of a value type can be created by the JVM, prior to initializing the class.
This is true for identity types, with a default value of null.
This needs to be true for value types, with a default value defined as containing the default values for all of the value type instance fields.

Class Initialization Impact on JVMS:

1. Require class initialization for local flattenable fields (instance and static) prior to any access to those fields.
   Given that non-bytecode accesses to fields today are gated on class initialization, we need the class initialization to complete initialization of any flattenable fields prior to completion of the class initialization.
   A future direction might allow lazy static initialization per field. At that time we will need to deal with non-bytecode accesses and their assumptions.
2. Additional triggers for class initialization on JVMS:
   Bytecodes that can return a default value instance of a value type would trigger class initialization for the value type. Note that the bytecode will know the actual type of the value type and won't require a consistency check:
   defaultvalue, anewarray, multianewarray

Example for Point which is NOT in ValueTypes attribute

Point NOT in VTs attribute, Point IS a Value Type	**ClassA: container/declarer**	**ClassB: caller**
local instance field, flattenable	CFE (assume even for VBC, this would be a CFE since local inconsistency)	N/A
local instance field, not flattenable	No eager loading *4	N/A
local method	No eager loading *5	N/A
CP resolution (defaultvalue, new, anewarray, checkcast)	class resolution: real mismatch: ICCE	N/A
field access	field resolution: ClassA vs. ClassA - always passes *4	field resolution: ClassB vs. Class A mismatch: ICCE *4
method invocation: caller vs. resolved method holder	method resolution: ClassA vs. ClassA - always passes *5	method resolution: ClassB vs. ClassA mismatch: ICCE *5
method invocation: resolved method holder vs. selected method holder	*5: Method overriding/selection cache check. Check if in VTs attribute for either one.	
local static field, flattenable	invalid	N/A
local static field, not flattenable	No eager loading *4	

*4 Field Access: VT Point NOT in ValueTypes attribute

instance or static field, not flattenable:
ClassA declares Point field. Point is NOT in ValueTypes attribute, Point IS a Value Type

- Point set to default value of null
- ClassA can read/write null from/to local field
- ClassA create instance of Point, check vs. real: ICCE

ClassA declares Point field. Point is NOT in ValueTypes attribute, Point IS a Value Type
ClassB has Point in ValueTypes attribute.

- ClassB read/write from/to ClassA field fails caller-callee consistency check: ICCE

ClassA declares Point field. Point is NOT in ValueTypes attribute, Point IS a Value Type
ClassC does not have Point in ValueTypes attribute

- ClassC can read/write null from/to ClassA field
- ClassC can not create an instance of Point: fails ClassC vs. real check: ICCE
- ClassC calls ClassB.m which returns a Point: fails ClassC-ClassB caller-callee check: ICCE
- ClassC calls ClassB.m1 which returns an Interface or Object:

    - ClassC casts Interface/Object to Point: CP resolution checks against real: ICCE
    - ClassC calls ClassB.m2 which returns an array of Point: *** require caller-callee check: ICCE

*5 Method invocation: VT Point NOT in ValueTypes attribute

ClassA declares method with Point parameter/return. Point not in ValueTypes attribute. Point is a value type.

- No eager loading, method created

ClassA declares method with Point parameter/return. Point not in ValueTypes attribute. Point is a value type.
ClassD extends ClassA and overrides the method with the Point parameter/return. Point IS in ClassD's ValueTypes attribute. Point is a value type.

- Class preparation includes loader constraint checking, performed during selection cache creation, between resolved class and the selected class. Recommend performing value type consistency checking at the same time if either ClassA, declarer of the overridden method, or ClassD, declarer of the overriding method have a parameter/return type in their ValueTypes attribute.
- ClassD's method overrides ClassA's method: fail overridden:overrider check: ICCE

ClassA declares method with Point parameter/return. Point not in ValueTypes attribute. Point is a value type.
ClassB has Point in ValueTypes attribute:

- ClassB invoke ClassA.method: caller-callee check: ICCE

ClassA declares method with Point parameter/return. Point not in ValueTypes attribute. Point is a value type.
ClassC also does NOT have Point in ValueTypes attribute.

- ClassC invoke ClassA.method: no check
  - This works for passing/returning null for Point.

Assumption: (Is this provable?)

- ClassC and ClassA are UNABLE to get their hands on an instance of Point
    - attempts to create an instance of Point fail real check: ICCE
    - Can not read Point, or array of Point from a field of a class that knows Point is a value type
    - Can not get a returned Point or array of Point from a method of a class that knows Point is a value type (assumes that ValueTypes attribute also tracks value type array elements)
  - Can not cast a returned Object/Interface to a Point: real check: ICCE

Examples for Arrays

For LW1, the created array will correctly reflect whether the real element is a value type or not.

Local fields declared as containing arrays are never flattened.

Here is a sample correctness issue that depends on method invocation and field access ensuring consistency for array element types:

ClassA declares Point field. Point is NOT in ValueTypes attribute, Point IS a Value Type

ClassB has Point in ValueTypes attribute.
ClassC does not have Point in ValueTypes attribute

- ClassC calls ClassB.m2 which returns an array of Point
    - ClassC performs aaload which does not do CP resolution, so if the method invocation does not check for value type consistency of array elements, ClassC, which does not know about Point, can acquire a valid instance of Point without recognizing that Point is a value type
- ClassC reads ClassB field containing an array of Point
    - ClassC performs aaload as above

It is more performant to perform the value type caller-callee consistency check at method invocation and field access than for each aaload or aastore bytecode.

TODO: explore impact of Value-Based-Class migration