Immutable Objects in Java

Christian Haack* Erik Poll Jan Schäfer Aleksy Schubert†

April 28, 2006

Abstract

Immutability is a very useful and desirable property for objects. This paper investigates different possible notions of immutability for Java objects, to find out which notion is the most intuitive and useful, both when it comes to ways of enforcing immutability of objects, and when it comes to exploiting information about (im)mutability of objects in program verification and in various static analyses. Our ultimate aim is to agree on a semantics for an `immutable` keyword in JML.

1 Introduction

Immutable classes are classes whose instances cannot be modified, i.e. whose instances are immutable objects. Favoring immutability is a recommended Java programming practice [Blo01, Goe03]. Benefits include avoiding aliasing problems, data races and integrity violations on objects passed to malicious or simply buggy code. Therefore, immutability often greatly simplifies writing correct and secure programs. Given that immutability is a common property that is very useful for program correctness, we propose to explicitly document it, as a JML-annotation [BCC+05] or possibly a Java-annotation [Sun06], and to try to exploit immutability in formal program verification.

Using object immutability for program verification. A major difficulty in verification of imperative programs is the *aliasing problem*. Verification systems have to keep track of aliasing, because memory locations can be modified through different aliases. Many systems for alias control have been proposed, e.g. [AC04, DM05, VB01] to cite just a few. Because the state of immutable objects is never modified, aliasing immutable objects is harmless and no alias control is necessary for immutable objects, although alias control may be needed to prove that objects are in fact immutable.

Immutability is particularly useful in multi-threaded programs, where *race conditions* can lead to unpredictable program behavior. Because methods of immutable objects do not write to their own state (and typically do not write to foreign state either), such methods avoid race conditions without the need for synchronization. Avoiding synchronization has, for instance, the benefit of *avoiding deadlocks*. Work on verification for multi-threaded programs often makes use of Lipton's theory of reduction [Lip75] to reduce the number of possible thread-interleavings to consider. This is particularly important when one wants to reason about methods in terms of pre- and post-conditions [RDF+05, FQ03]. Methods of immutable objects are typically *constant*, that is they do not depend on mutable state. In terms of Lipton's theory, constant methods are *movers*, a notion that this theory greatly exploits. Movers are special cases of *atomic methods*.

Knowing that methods always return the same results can be exploited in program verification. Indeed, Cok [Cok05] explains how ESC/Java2 already exploits a keyword `immutable`.

Object immutability is often important for *security*. For instance, when assigning a `Permission` to a `URL`, it is crucial that the `Permission` and `URL` objects are immutable. Otherwise, their *integrity* could possibly be violated, for instance, by an attack that exploits a race condition.

*Supported by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

†Supported by an EU Marie Curie Intra-European Fellowship.

Indeed, immutability of strings is essential for the security of the Java platform. NB. immutability of strings is only guaranteed thanks to the recent revision of the Java Memory Model [JSR, MPA05].

Defining and enforcing immutability. Our ultimate goal is to specify and verify object immutability in order to exploit it for verifying other program properties, for instance, functional specifications. Syntactically, specification is simple: we could use a JML keyword `immutable` or a Java metadata tag `@Immutable`. But what would be its precise semantics? Intuitively, an immutable object is simply one that cannot be modified. Unfortunately, the details are more involved than this. On the one hand, a good semantics for an `immutable` keyword should be useful for verifying program properties other than immutability itself. On the other hand, we would like to have sound rules for statically verifying immutability. In this paper, we attempt to define the semantics of object immutability. We propose two kinds of definitions. One definition is *observational*: an object is immutable if an observer cannot tell the difference between two instances of the same object at different points in time. The other definition is *state-based*: an object is immutable if its associated state does not mutate after initialization. In both cases, we call a *class* immutable if its instances are immutable *objects*. We believe that the notion of observational immutability best captures the intuition behind immutability, whereas the notion of state-based immutability better lends itself for static checking. We do not come up with ultimate, conclusive definitions, but hopefully provide a good start towards more finished solutions in the future.

Structure of the paper. Section 2 presents examples that illustrate some of the difficulties and subtleties for defining the semantics of immutability. Section 3 discusses observational immutability and Section 4 state-based immutability, and Section 5 the relation between these notions. Section 6 proposes rules for checking immutability.

2 Complications and Pitfalls

In this section, we collect examples that illustrate complications and pitfalls for defining and enforcing immutability.

2.1 Instruction reorderings and the need for `final`

The following class seems immutable in every sense: its only field is private and cannot possibly be modified after construction.

```
public class UnsafeInteger{
  private int n;
  public UnsafeInteger(int argn) {n = argn;}
  public int getValue(){ return n;}
}
```

However, in a multi-threaded context this class is *not* immutable. To make it immutable we need to declare the field `n` as `final`.

As a counterexample to its immutability, consider two threads, t_1 and t_2:

- t_1 executes `shared = new UnsafeInteger(5);` for some static field `shared`, and
- t_2 executes `local = shared.getValue();` for some variable `local`.

A problem may arise if t_1 and t_2 execute simultaneously. Execution of `shared = new UnsafeInteger(5)` takes three steps:

1. memory is allocated for the new object, say at position p on the heap;
2. the field `n`, at some offset in this memory, is set to 5;
3. the field `shared` is set to point to p.

The Java compiler or virtual machine is allowed to interchange the order in which 2 and 3 are executed, because changing the order will not make any difference to the thread t_1, as these assignments are to different fields [GJSB05, MPA05]. However, if the order of 2 and 3 is interchanged,

then thread t_2 can invoke `getValue()` and observe the value 0, in the following thread interleaving:

t_1	t_2
memory allocated at position p	
`shared` $= p$	
	`getValue()` invoked on `shared`, returning 0
p.`n` $= 5$;	

Declaring `n` as `final` solves the problem. This is *only* thanks to the relatively recent revision of the Java Memory Model [JSR]. Note that it is easy to forget final declarations. In fact, many supposedly immutable classes in the reference implementation of the Java API (version 1.4) fail to declare their `private` fields as `final`; this seems to have been fixed for Java 1.5.

2.2 Letting `this` escape

The root cause of the problem with `UnsafeInteger` above is that even an immutable object will be mutable for some span of time, namely during the execution of its constructor. (The only exceptions are rare cases where the object leaves all its fields initialized to the Java default initial values.) If other objects can get hold of references to a supposedly immutable object before its constructor has terminated, this breaks immutability, as the object can be observed to change state. This can even happen in single-threaded programs:

```
public class UnsafeWindow implements EventListener {
  final public int x, y;
  public UnsafeWindow(EventHandler e, int argx, int argy) {
      e.register(this);
      x = argx; y = argy;
  }
  ...
}
```

Here `e.register(this)` may possibly call back a method on `this`, whose constructor has not yet terminated. This way, the object `e` may observe two different states of `this`: one state in the middle of construction and another state perhaps later after `e`'s constructor has terminated.

For a thorough discussion of this issue see [Goe02].

Of course, the example above is a bit odd in that the call to `e.register(this)` is not done as the last statement of the constructor, and happens before `x` and `y` have been initialized. However, even moving `e.register(this)` to the very end of the constructor may not solve the problem, due to the possibility of instruction reorderings discussed in Section 2.1. Also, there is no way to ensure that in subclasses, which may invoke a super-constructor, the call to `register` does not happen before construction is complete.

2.3 Compound objects and representation exposure

If an immutable object `o` has a mutable sub-object `s`, whose state is meant to be part of `o`'s state, it is important that no other object can obtain a reference to `s`. Otherwise, `o`'s state can be modified indirectly through this reference. For instance, Java's `String` class has a character array as a mutable sub-object[1]. The implementation of the constructor `String(char[] a)` is careful to copy the parameter `a` instead of assigning it to the private character array field. Like this:

```
public class MyString {
  final private char[] a;
  MyString (char[] a) {
    this.a = new char[a.length];
    System.arraycopy (a, 0, this.a, 0, a.length);
  }
  ...
}
```

Representation exposure is dangerous under other circumstances, too, and there have been many proposals on how to avoid it, using various notions of ownership, alias control, and encapsulation [VB01, DM05, AC04]. This is an active research field, and the best solution(s) are still not clear.

[1] All Java arrays are mutable.

2.4 Reading outside one's own state

Consider the following class:

```
class StateLess {
  public int getValue() { return System.currentTimeMillis(); }
}
```

The class `StateLess` is immutable in the sense that its instances do not mutate their state: `StateLess` objects do not even have state. However, `StateLess` is not observationally immutable because observers obtain different results every time they call `getValue()`. Here is a variation of this example. Should the class `StateLess2` be called immutable?

```
class StateLess2 {
  public int foo(Mutable o) { return o.m(); }
}
```

2.5 Circular dependencies of static initialization

It makes sense to allow methods in immutable classes to read final static fields, even if these are outside their state. However, final static fields can be observed to change value, and hence are not really constants, if there are circular dependencies in their initialization. For example, this occurs in

```
public class C { public final static int c = D.d+1; }
public class D { public final static int d = C.c+1; }
```

Depending on the order in which these classes are loaded, `C.c` will be set to 1 and `D.d` will be set to 2, or vice versa.

The Java Language Specification ([GJSB05], §15.28) defines the notion of (compile time) *constant expressions*. Not all final static fields are constant expressions. In particular, `D.d` and `C.c` above are not.

Of course, circular dependencies in static initialization are extremely bad programming practice. Interestingly, the source code analyzer FindBugs [HP04] will detect such circularities.

2.6 Unobservable mutations, incl. lazy initialization

Immutable objects may possibly mutate their state, as long as this is not observable to other objects. One common form of unobservable mutation is *lazy initialization*. Lazy initialization is commonly used in the implementation of `hashCode()`, where the computation of the hashcode is only performed the first time it is needed. Another common example is *memoization*. The reason for using both techniques is efficiency. In both cases, fields or data structures are initialized or updated after the constructor of the object has terminated.

2.7 Immutability vs purity

It make sense to require that immutable classes do not have methods with visible side-effects, i.e. that all methods are *pure* in the sense of JML. Note, however, that `java.lang.String` has impure methods:

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

This method copies the characters from a string into the destination array `dst`, which clearly has a side-effect on `dst`. This suggests that requiring immutable classes to be pure may be too restrictive, although examples of impure methods in immutable classes seem rare.[2]

3 Observational immutability

For simplicity, we assume in this section that immutable objects do not have public fields. To cover public final fields, too, one could treat those analogously to accessor methods.

Intuitively, an object `o` is observationally immutable if it is impossible to observe differences between two instances of `o` at different points in time. An important decision is what kind of observations we allow for discriminating instances of the same object. We choose to discriminate by equality comparisons of method results. In other words, we define an object as observationally immutable if all its public methods are constant:

[2] In fact, `getChars` is depreciated, though for reasons that have nothing to do with its impurity.

- A *method is constant* if invoking it with the 'same' arguments always returns the 'same' result.
- An *object is immutable* if all its public methods are constant.
- A *class or interface is immutable* if all its instances are immutable objects.[3]

Restricting observations to equality comparisons of method results is not the only possible choice. Instead one could allow other contexts to discriminate between different instances of the same immutable object. Other choices would give different notions of immutability. For instance, our definition permits that immutable objects visibly mutate global state that is not part of the object's state. With other choices of discriminating contexts, this may be disallowed.

3.1 What is the 'same'?

We have to clarify what we mean by the 'same' in our definition of constant methods. If the argument and result types are primitive, the only sensible definition is `==`. If the argument and result types include reference types, there are two possibilities: *reference equality* (`==`) and *defined equality* (`.equals`). We believe that the choice must be defined equality. If we chose reference equality instead, the above definition of immutability would be too restrictive. Consider, for instance, the following interface for immutable integer lists:

```
public interface ImmIntList {
  ImmIntList nil = new Nil();
  ImmIntList cons (int hd);
}

class Nil implements ImmIntList {
  ...
}
```

```
class Cons implements ImmIntList {
   final private int hd;
   final private ImmIntList tl;
   Cons (int hd, ImmIntList tl) {
     this.hd = hd; this.tl = tl;
   }
   public ImmIntList cons (int hd) {
     return new Cons(hd,this);
   }
   public boolean equals (Object o) { ... }
}
```

If we chose the 'same' to mean reference equality, then the object `ImmIntList.nil` would not be immutable by our definition, because:

```
ImmIntList.nil.cons(42) != ImmIntList.nil.cons(42)
```

We define:

> Two *primitive values* are the 'same' if they are equal up to `==`. Two *objects* are the 'same' if they are equal up to `.equals`.

Our definition of immutability depends on `.equals` and it is important that this is implemented in a reasonable way. In particular, its implementation must satisfy the contract from the `Object` API, i.e., it must be reflexive, symmetric, transitive and consistent.

3.2 Mutable method arguments

We think that our definition of observational immutability seems reasonable. However, there are corner cases that are somewhat weird. These have to do with interactions of immutable objects with mutable ones. For instance, is our definition reasonable for methods with mutable arguments? Consider again the class `StateLess2` from Section 2.4:

```
class StateLess2 { public int foo (Mutable o) { return o.m(); } }
```

By our definition, this class is immutable if the following holds:

> `o.equals(o')` $\Rightarrow$ `o.m() == o'.m()`, for all instances `o` and `o'` of `Mutable`

This condition is often satisfied, for instance, if `Mutable` is a mutable collection class. There are other circumstances where this condition may not be satisfied, for instance, if `Mutable` implements `.equals` as reference equality.

[3] One may argue that immutable classes must be `final`, because one can always break immutability by subclassing. We instead take the view that immutability is part of a class's contract and subclasses are disallowed to break it.

3.3 Mutable type parameters

Consider immutable lists of arbitrary objects:

```
public interface ImmObjList {
  ImmObjList nil = new Nil();
  ImmObjList cons (Object hd);
}
```

In spite of its name, `ImmObjList` is not observationally immutable by our definition. The problem comes up for immutable lists whose elements are of mutable type:

```
static Vector mutable = new Vector ();
static ImmObjList list_of_vectors = ImmObjList.nil.cons(mutable);
```

Observational immutability is broken, because `list_of_vectors.equals` reads the state of the mutable list members:

```
ImmObjList other = ImmObjList.nil.cons(new Vector());
boolean x = list_of_vectors.equals(other); // first call of list_of_vectors.equals
mutable.add (new Integer(0));              // mutate list member
boolean y = list_of_vectors.equals(other); // second call of list_of_vectors.equals
// for immutability, x and y should now be the same
boolean b = (x == y);                      // b is false! x and y are not the same!
```

The lesson we learn is that generic immutable lists are only observationally immutable if the list elements themselves are immutable. This can be cleanly expressed if we implement immutable lists using parameterized (aka generic) types:

```
public interface ImmList<E> {
  ImmList<E> nil = new Nil<E>();
  ImmList<E> cons (E hd);
}
```

We define:

> A *parameterized interface or class* `T<E1,..,En>` is observationally immutable if `T<U1,..,Un>` is observationally immutable for all observationally immutable `U1,..,Un`.

By this definition, the parameterized type `ImmList<E>` is observationally immutable.

4 State-based immutability

One way of understanding and possibly enforcing immutability of objects is in terms of the *state* of object, i.e. the part of the heap that constitutes the 'state' of an object. We will use the term *location* to mean a memory cell on the heap that is used to store the value of a field or an element of an array. The idea is that an object is immutable if there are no assignments to the locations that make up the 'state' of an object, once the constructor of the object has terminated.

A first issue here is determining which locations belong to a given object. Clearly it should include the instance fields of the object. But if one of these field is a reference, the question is if the fields of the object[4] are also part of the state. This is of course a well-known issue in object-oriented programming, and a variety of *ownership systems* have been proposed to resolve the issue; e.g. see [DM05, VB01, AC04], to mention just a few of the many papers on this topic.

A very simplistic approach, which avoids the need of using any ownership system, is to say the state of an object to consist of the instance fields of that object[5]. We call this *shallow state-based immutability*. If a class only declares instance fields of primitive types or of immutable types – i.e. it has no instance fields of mutable types – then this simple notion is perfectly adequate. For example, this notion would be adequate for the class `Cons` in Section 3, but wouldn't cope with the class `MyString` from Section 2.3.

More generally, one would have to rely on an ownership system to delineate what the state of an object is and to prevent leaking of references to mutable sub-objects. We call this *general*

[4] Or, in case of an reference to an array, the contents of this array; for this paper the contents of an array are simply treated as fields of the array object.

[5] Or, in the case of an array, the contents of the array and its length field.

state-based immutability. Note that the notion of immutability can be used to relax restrictions imposed by ownership systems, as references to immutable objects can always be freely shared.

In many cases, the state of an object consists simply all the locations reachable from its instance fields, i.e. its instance fields, the instance fields of its instance fields, etc. We call this *deep state-based immutability*. For example, the class `MyString` is deeply immutability; an object of this class has one instance field that is a reference to another object, namely the field `a` which points to a character array, and clearly this array and its contents are part of the state of the `MyString` object. To give another example, the class `ImmObjList` is not deeply immutable, because for an object of this class, which is an immutable list of mutable objects, the reachable state will include those mutable objects, which clearly do not belong to the 'state' of the immutable list. Many immutable classes in the Java API are be deeply immutable, for example `String` and `URL`.

5 Relating observational and state-based immutability

Classes that employ lazy initialization and memoization are observationally immutable, but not state-based immutable. Therefore, observational immutability does not imply state-based immutability. The reverse implication is not quite true either, because our definition of state-based immutability allows immutable objects to read from mutable locations outside their own state. The class `StateLess` provides a counterexample:

```
class StateLess { public int getValue() { return System.currentTimeMillis(); } }
```

We define a *readability restriction*:

> An object satisfies the *readability restriction* if each of its methods only reads its own state, the state of other immutable objects or static final fields that are compile-time constant expressions.

Note that instances of `StateLess` in Section 2.4 do not satisfy the readability restriction. Instances of `StateLess2` do not satisfy the readability restriction either (assuming that `o.m()` reads `o`'s state):

```
class StateLess2 { public int foo (Mutable o) { return o.m(); } }
```

We believe that state-based immutable objects that satisfy the readability restriction are observationally immutable.

6 Enforcing immutability

This section suggests some rules which could be used to check if a class is immutable. We believe the set of rules below is sufficient to ensure that a class is immutable. The set of rules that we arrived at is very similar to the one given in [Sch04]. The rules enforce a notion of state-based immutability plus the readability restriction, so, as discussed in Section 5, should guarantee observational immutability.

1. *All instance fields are final.*
   This is needed to avoid problems mentioned in Section 2.1. This rule could be weakened to allow for lazy initialization; see Section 6.1 below.
2. *All constructors have side-effects only on the newly allocated state.*
   This is equivalent with saying the constructor is *pure* in the sense of JML. This should also guarantee that constructors do not leak `this`, the problem discussed in Section 2.2.
   One subtle way of leaking `this` is the starting of a new thread, as explained in [Goe02]. This would only be prevented by the rule above if the specification of `start()` in `Thread` states that it has a side-effect on some global state. If not, an additional check for this would be needed.
3. *All methods are side-effect free.*
   This is equivalent with saying the method is *pure* in the sense of JML. While this is easier to enforce than the rule above for constructors, in general this may require program verification. This rule could be relaxed to allow methods that have a side effect on some mutable argument, such as the `getChars` method in `String`.

4. *References to mutable sub-objects are not leaked or imported.*
   Here by sub-object we mean any object that can be reached following instance fields (incl. array accesses), as we consider deep immutability. The way to enforce this would be to use a system for ownership, for instance universes [DM05]. If one employs such a system we can refine the enforced notion form deep immutability to general immutability, using the notions the ownership system provides to delineate what constitutes the state of the object.
   In simple cases, instead of using a system for ownership one could use the JML keyword `\fresh`. For example, for the class `MyString` in Section 2.3, requiring that the local array `a` is `\fresh` in the postcondition of the constructor ensures the constructor doesn't import a shared reference to a mutable object.
5. *(Readability restriction) Methods can only state of the current object, and state of other immutable objects or static final fields that are compile-time constants.*
   Enforcing this in general requires the use of `readable` clauses. Boyland and Green [GB99] present an effect system that enforces such clauses.
6. *Superclass of immutable classes must be either immutable or* `Object`.
7. *Subclasses of immutable classes must be immutable.*
   Malicious code may try to subclass an immutable class `A`. If an application imports references to supposedly immutable objects of type `A`, this is a concern: malicious code might create a mutable subclass `A'` of `A`, and then sneak in a mutable object of this type `A'`, masquerading under it's parent type `A`. If an application creates all its immutable objects of class `A` itself, by invoking one of `A`'s constructor, it is not a concern.
   In the former case, a way to prevent this is to declare the immutable class as final. This would of course rule out inheritance between immutable classes.

6.1 Allowing lazy initialization

The precise conditions under which lazy initialization does not break immutability are tricky. To enforce that lazy initialization is safe, the only feasible approach seems to be to restrict programmers to very particular programming patterns.

Allowing lazy initialization relaxes the rule 1 above to

1* *All instance fields are final or lazy.*

A lazy field must (i) be private and may (ii) only be referenced in one associated getter-method, which takes care of the lazy initialization.

The associated getter-method must meet additional requirements. A safe approach is (iii) to synchronize the getter-method, (iv) to only perform initialization of the lazy field if it still has the default initial value for its type (e.g., null for a reference type or 0 for a numeric type), and (v) to always set the lazy field to value that is different from the default value.

The rules (iii-v) will guarantee that the lazy initialization is performed only once, even in a multi-threaded context. Rules (i-iv) could be enforced syntactically by a source code analyzer, but enforcing rule (v) would have to rely on program verification.

One could further relax the programming pattern above; in particular, one could relax conditions (iii-v). For instance, one may wish to leave the getter-method unsynchronized, for efficiency. However, then there is the additional complication of ensuring that concurrent initializations in different threads cannot cause problems. If the lazy field is a long or double, synchronization at some stage is unavoidable, as assignments to longs and double are not guaranteed to be atomic. If the lazy field is a reference, one has to be careful not to fall in the trap of double-checked locking, which is a standard – but notoriously unsound [BBB$^+$] – programming pattern.

If one wants to remove synchronization from the getter-method, the technique described in [BNS04] could be used to prove that the lazy initialization is a harmless – or unobservable – side-effect. The goal of [BNS04] was to allow harmless side effects in pure methods. Note that this technique relies on program verification. The technique does not cope with lazy initialization of references, e.g. lazy allocation of sub-objects.

One could use a different value than the default initial value to signal that the lazy field has not been initialized, especially in case the initialization may set the field to the default initial value,

but this also introduces an additional complication, in light of the issue highlighted in Section 2.1. Additional synchronization would be needed to ensure that the constructor has terminated before another thread first executes the lazy getter method.

Maybe we should accept that we will not come up with tool support that provides 100% guarantees for immutability. For some highly optimized code, we might have to rely on human intelligence to decide whether a class is immutable or not.

7 Related work

The set of rules that we arrived at is very similar to the one given by Jan Schäfer in [Sch04]. In the motivation for his rules, Schäfer mentions many of the issues raised in Section 2, but not the issues due to multi-threading discussed in Section 2.1 or the issue of leaking `this` discussed in Section 2.2. Schäfer considers using the system of [VB01] for alias control, and does not consider the full readability restriction (our rule 5), though he does note the problem of a method in an immutable class that reads mutable global state.

[Goe03] also gives some guidelines for making classes immutable. He does not consider the reachability condition, or gives rules to allow lazy initialization. JSR-133 ([JSR], §9.1) also discusses immutable objects.

[BE04, TE05] look at reference immutability, which is a different issue altogether. None of the papers above considers a notion of observational immutability.

[Cok05] explains how ESC/Java2 exploits `immutable` annotations on classes to simplify verification. He exploits the fact that methods on immutable classes are constant, but avoids the cases where these methods have arguments or results of reference types. He does not consider the issue of enforcing the correctness of immutability notations.

8 Conclusions

We discussed the notions of state-based and observational immutability, the relation between them, and proposed some rules for enforcing immutability. With the exception of rules allowing lazy initialization, it seems that a statically enforceable set of rules is feasible. For instance, Greenhouse and Boyland present a statically checkable effect system for `reads` and `writes` clauses [GB99]. This system (or a similar one) can be helpful for enforcing immutability, if we use `writes` clauses to express that methods on immutable objects must not modify their own state and `reads` clauses for the readability restriction. However, such a set of rules would be incomplete, and would not, for instance, allow lazy initialization.

It would be nice to relate our two definitions of immutability more rigorously and prove, for instance, that state-based immutability plus the readability restriction implies observational immutability. However, because immutability is particularly interesting in multi-threaded programs, it is crucial that we treat multi-threaded Java, too. Moreover, it is important that object immutability does not break in incorrectly synchronized programs. We may also have to consider the use of improperly synchronized code inside immutable classes; e.g. a highly optimized immutable class might include (safe) race conditions to do efficient lazy initialization. Formal reasoning about improperly synchronized programs is extremely difficult, because of the weak guarantees that the Java memory model makes about such programs. For these reasons, obtaining formal soundness guarantees seems very difficult at this point.

References

[AC04] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.

[BBB+] David Bacon, Joshua Bloch, Jeff Bogda, nd Paul Haahr Cliff Click, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "double-checked locking is broken" declaration. Available from `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[BE04] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[Blo01] Joshua Bloch. *Effective Java*. Addison-Wesley, 2001.

[BNS04] M. Barnett, W. Naumann, and Q. Sun. 99.44% pure: useful abstractions in specifications. In *Formal techniques for Java-like programs (FTfJP'2004)*, pages 11–18, 2004. Technical Report NIII-R0426, University of Nijmegen.

[Cok05] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005. Available online at `http://www.jot.fm/issues/issues_2005_10/article4`.

[DM05] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[GB99] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, number 1628 in Lecture Notes in Computer Science, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*, chapter 17. Sun Microsystems, 2005. Available at `http://java.sun.com/docs/books/jls/`.

[Goe02] Brian Goetz. Java theory and practice: Safe construction techniques–don't let the "this" reference escape during construction. *IBM DevelopersWork*, 2002.

[Goe03] Brian Goetz. Java theory and practice: To mutate or not to mutate? Immutable objects can greatly simplify your life. *IBM developersWork*, 2003.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[JSR] JSR-133: Java Memory Model and Thread Specification.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[MPA05] J. Manson, W. Pugh, and S. Adve. The Java memory model. *To appear in ACM TOPLAS, Special POPL'05 Issue*, 2005. `http://www.cs.umd.edu/users/jmanson/java.html`.

[RDF+05] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In Andrew P. Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 551–576. Springer-Verlag, July 2005.

[Sch04] Jan Schäfer. Encapsulation and specification of object-oriented runtime components. Master's thesis, Universität Kaiserslautern, 2004.

[Sun06] Sun Microsystems, Inc., `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`. *Annotations Java Programming Language*, Date retrieved: March 28, 2006.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[VB01] J. Vitek and B. Bokowski. Confined Types in Java. *Computer Networks*, 36(4):407–421, 2001.