Immutable objects in Java

Erik Poll
Radboud University Nijmegen

Immutable objects in Java

- Context of this work: verification of Java programs annotated with JML specifications
- Goal: a notion of immutable object, that can be statically enforced, guarantees immutability, and can be exploited in program verification
- Work in progress: more inventory of (solvable) problems than a solution

Overview

Java provides final – ie. immutable - fields
What about immutable objects ?

- **Why would we want this?**
- **What does immutable mean ?**
  How to enforce and exploit it ?

Why immutability ? (1)

Good programming practice

"immutable objects greatly simplify your life"

- no problems with aliasing
- no problems with race conditions
- ...
- conceptually: immutable object is a value, as in functional programming

Why immutability ? (2)

Performance

- no need for synchronisation
- compiler and VM optimisations

Why immutability ? (3)

Specification

- interesting property to specify, not just because of (1) and (2), but as an important integrity property
- Eg immutability of Strings, URLs, permissions, etc. vital for security

Why immutability ? (4)

Reasoning/program verification

```
public void m(String str){
 if (str.equals("abc")) {
      y.f[0]='x';
      //@ assert str.equals("abc");
      ...
}
```

Why immutability ? (4)

JML has a library of – supposedly immutable - model classes, for mathematical objects such as sets, relations,

```
//@ public JMLObjectSet s;

//@ requires ! s.contains(o);
//@ ensures  s.equals(\old(s).union(o));
public void addListener(Object o) { ... }
```

Guaranteeing immutability

starting point: `pure`

JML has the notion of pure

- pure method has no side-effects
- pure constructor has no side-effects, except on newly allocated state
- pure class only has pure methods, pure constructors, and pure sub-classes

pure does not imply immutable

```
public /*@ pure @*/ class Integer{
  public int i;
  public Integer(int j){ i = j; }
  public int getValue(){ return i; }
}
```

Pure (no side-effects), but not immutable, because anyone can change the public field i

Is this pure class immutable ?

```
public /*@ immutable?? @*/ class Integer {
  private int i;
  public Integer(int j){ i = j; }
  public int getValue(){ return i; }
}
```

Still not immutable, because field i is not final:
`Integer(5)` **may be observed to change from 0 to 5 in multi-threaded programs**

Final is necessary for field immutability

```
class /*@ immutable @*/ Integer {
  private final int i;
  public Integer(int j){ i = j; }
  public int getValue(){ return i; }
}
```

Thanks to the newly revised Java Memory Model (JSR-133)

Final is not suffient for field immutability

```
public /*@ immutable?? @*/ class Integer {
  public static Integer latest;
  private final int i;
  public Integer(int j){ i = j;
                          latest = this;} // leaks
  public int getValue(){ return i;)
}
```

Constructor leaks `this`, hence field `i` not immutable:
`Integer(5)` may be observed to change from 0 to 5.

There are a few more ways to leak `this`

Immutable instance field

- final instance fields are not always immutable
- final instance field is immutable, provided the constructor doesn't leak a reference to `this`
- One of the goals of the new Java Memory Model (JSR-133)

Shallow immutability

- A pure class are shallowly immutable iff
  1. all instance fields are final, and
  2. constructors don't leak `this`

- Definition implicit in JSR-133
- Usually too weak: we often want fields of fields (sub-objects) to be immutable too

Shallow immutability too weak

```
public /*@ immutable? @*/ class BankTransfer{
   private final char[] src,dest; //account nr's
   private final Integer amount;
   ...
   char[] getDest(){ return dest; }  // not ok
   Integer getAmount(){ return amount; } // ok
  }
```

We may want sub-components `src` and `dest` to be immutable too...

If so, leaking references to them is not ok

Deep immutability

A pure class is deeply immutable if

1. all instance fields are final, and
2. constructors don't leak `this`, and
3. all instance fields that are references
   1. have immutable types, or
   2. cannot be aliased (enforced using some form of alias control)

Deep immutability too strong

```
public /*@ immutable?? @*/ BankTransfer {
  private final Integer amount;
  private final BankAccount src, dest;
  ...
}
```

Deep-immutability would require immutability of the source and destination bank accounts.

What if we only want immutability of the references `src` and `dest`, but not the objects they refer to?

Deep immutability too strong

```
public /*@ immutable @*/ BankTransfer {
  private final Integer amount;
  private final /*@ mutable @*/ BankAccount
                                      src, dest;
  ...
}
```

`src` and `dest` excluded from the "state" of the immutable `BankTransfer` object: references are part of the "state", but the objects they point to are not.

(Javari notation of mutable used here; JML actually has different notion of universe to delimit object state.)

State-based immutability

A pure class is state-based immutable if

1. all instance fields are final, and
2. constructors don't leak `this`, and
3. all instance fields that are references
   i. have immutable types, or
   ii. cannot be aliased, or
   iii. excluded from the "state" of the object

Javari of [Birka&Ernst] provides this (almost)

Exploiting immutability in program verification

Observational immutability

- Example: `bankTransfer.getAmount()` is a constant
- object is "observationally immutable" if we cannot observe any mutation by invoking its methods
- if o is observationally immutable, then

  o.m(x1,...,xn)

  always returns the same result, if xi are primitive values or immutable objects

Exploiting immutability in ESC/Java2

A method

    C m(C1 x1, ... Cn xn)

is interpreted as function

    m : GlobalState×Ref×C1×..×Cn —> C

For immutable objects we can omit state

    m :   Ref×C1×..×Cn —> C

if all Ci are primitive or immutable

Implemented by David Cok in ESC/Java2

State based immutability does not imply observational immutability

```
public /*@ immutable @*/ StrangeInteger {
   final int i;
   StrangeInteger(int j){ i = j; }
   int getValue(){ return SomeClass.someStaticField;}
}
```

Excluding such examples requires analysing read's as well as write's...

Immutable object should not write in its state and not read outside its state (or – more liberally – only read immutable fields outside its state)

Two views on immutability

1. state-based: no side-effects on the "state" of an object
2. observational: methods behave as mathematical functions, and "always" returning the same result

Proposed analyses are for 1, but for program verification we want 2, which requires a more complicated analysis: looking at read effects, as well as write effects

Related work

Javari [Birka & Ernst, *OOPSLA'04*]

- **proposal to add readonly modifier to Java**
- **more refined notion of immutability, eg allowing both mutable and immutable (readonly) references to the same object**
- **enforces state-based immutability**
- **doesn't guarantee observational immutability**

Exploiting immutability further?

```
public JMLObjectSet {
  JMLObjectSet add(Object o) {...}
  JMLObjectSet remove(Object o) {...}
  boolean contains(Object o) {...}
}
```

`s.contains(o)` always gives the same result, even if `o` is not an immutable object

Checking this would involve checking if `o` is dereferenced in the body of `contains`

Alternative approach

We could also give a native implementation (or axiomatisation) of an immutable class such as JMLObjectSet in the back-end theorem prover.

Maybe this is a better way to fully exploit the property of JMLObjectSets being immutable.

Open question

- **Notion of purity (absence of all side-effects) in practice often too strong. Sometime we want to allow harmless side-effects.**

  Eg [99.44% pure, Barnet et al.]

- **Does the same hold for immutability?**

Conclusions

- Immutability is nice property, that deserves to be documented, if not in Java then in JML
- Main gain not in program verification, but stressing design decision and lightweigth static checks
- At least two notions of immutability: state-based immutability considers write's but not read's, and hence can't guarantee observational immutability
- Good news: exploiting immutability in verification is easy
- Bad news: enforcing it is possible, but complicated
- Checking observational immutability requires alias control and effect system for reads.