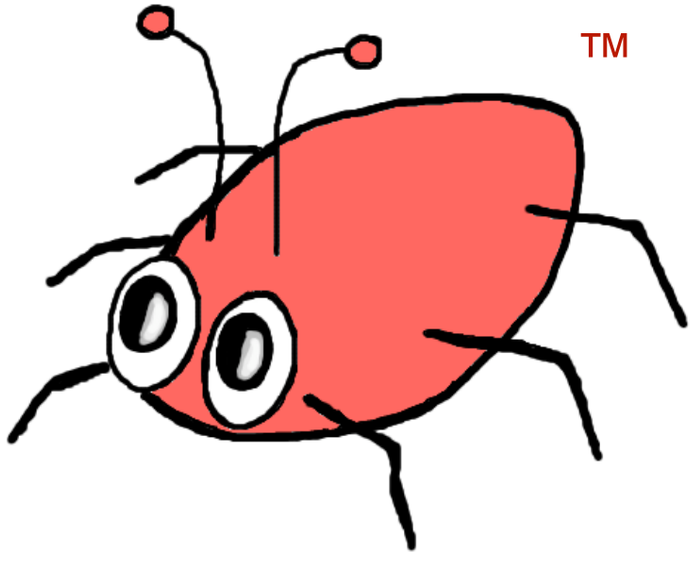
OOPSLA
Montréal, Canada 2007
October 21-25
Palais des Congrès

# Using FindBugs™ in Anger

David Hovemeyer
York College of Pennsylvania

William Pugh
Univ. of Maryland



# In Anger

- British slang
- in depth or comprehensively. Within the context of using something 'in anger.'
  - “No commander, I can't say for certain that our missile guidance system is accurate until we have used it in anger.”
- http://www.urbandictionary.com/define.php?term=anger, definition 6

# Static Analysis

- Analyzes your program without executing it
- Doesn't depend on having good test cases
  - or even any test cases
- Generally, doesn't know what your software is supposed to do
  - Looks for violations of reasonable programming
    - Shouldn't throw NPE
    - Shouldn't allow SQL injection
- Not a replacement for testing
  - Very good at finding problems on untested paths
  - But many defects can't be found with static analysis

# Common Wisdom about Bugs and Static Analysis

- Programmers are smart
- Smart people don't make dumb mistakes
- We have good techniques (e.g., unit testing, pair programming, code inspections) for finding bugs early
- So, bugs remaining in production code must be subtle, and finding them must require sophisticated static analysis techniques
  - I tried lint and it sucked: lots of warnings, few real issues

# Can You Find The Bug?

# Can You Find The Bug?

```
if (listeners == null)
    listeners.remove(listener);
```

- JDK1.6.0, b105, sun.awt.x11.XMSelection
  - lines 243-244

# Why Do Bugs Occur?

- Nobody is perfect
- Common types of errors:
  - Misunderstood language features, API methods
  - Typos (using wrong boolean operator, forgetting parentheses or brackets, etc.)
  - Misunderstood class or method invariants
- Everyone makes syntax errors, but the compiler catches them
  - What about bugs one step removed from a syntax error?

# Effectively Using Static Analysis to Improve Code Quality

# No silver bullets

- Static analysis isn't a silver bullet
  - won't ensure your code is correct or of high quality
- Other techniques are just as valuable, if not more so
  - careful design
  - testing
  - code review

# Finding the right combination

- Everything you might do to improve software quality
  - is very effective at finding some kinds of problems
  - is subject to diminishing returns
- Want to find an effective/profitable way to use static analysis to improve software quality

# But static analysis is nice...

- Your first 1-4 hours with a static analysis tool will find a lot more issues than your first 1-4 hours writing tests
  - skim the cream
- Once static analysis is an automated part of your software development process
  - looking at just the new high confidence issues is effective and efficient

# This tutorial

- What FindBugs is and does
- Using FindBugs well and wisely
  - Customizing FindBugs to your needs
- Adapting FindBugs to your time budget
  - Find your sweet spot
- Making FindBugs part of your continuous build and test framework

# Running the analysis and finding obviously stupid code is easy

- Often, the hard part is stuff like:
  - Figuring out who is responsible for that code
  - Understanding what the code is actually supposed to do
  - Figuring out if stupid code actually causes the application to misbehave
  - Writing a test case that demonstrates the bug
  - Getting approval to change the code

# FindBugs success stories

# Small projects

- Lots of small projects run FindBugs as part of their process
- Examine and fix or filter all FindBugs issues
  - Java Server Faces
  - Sleepycat Java database
  - Hadoop

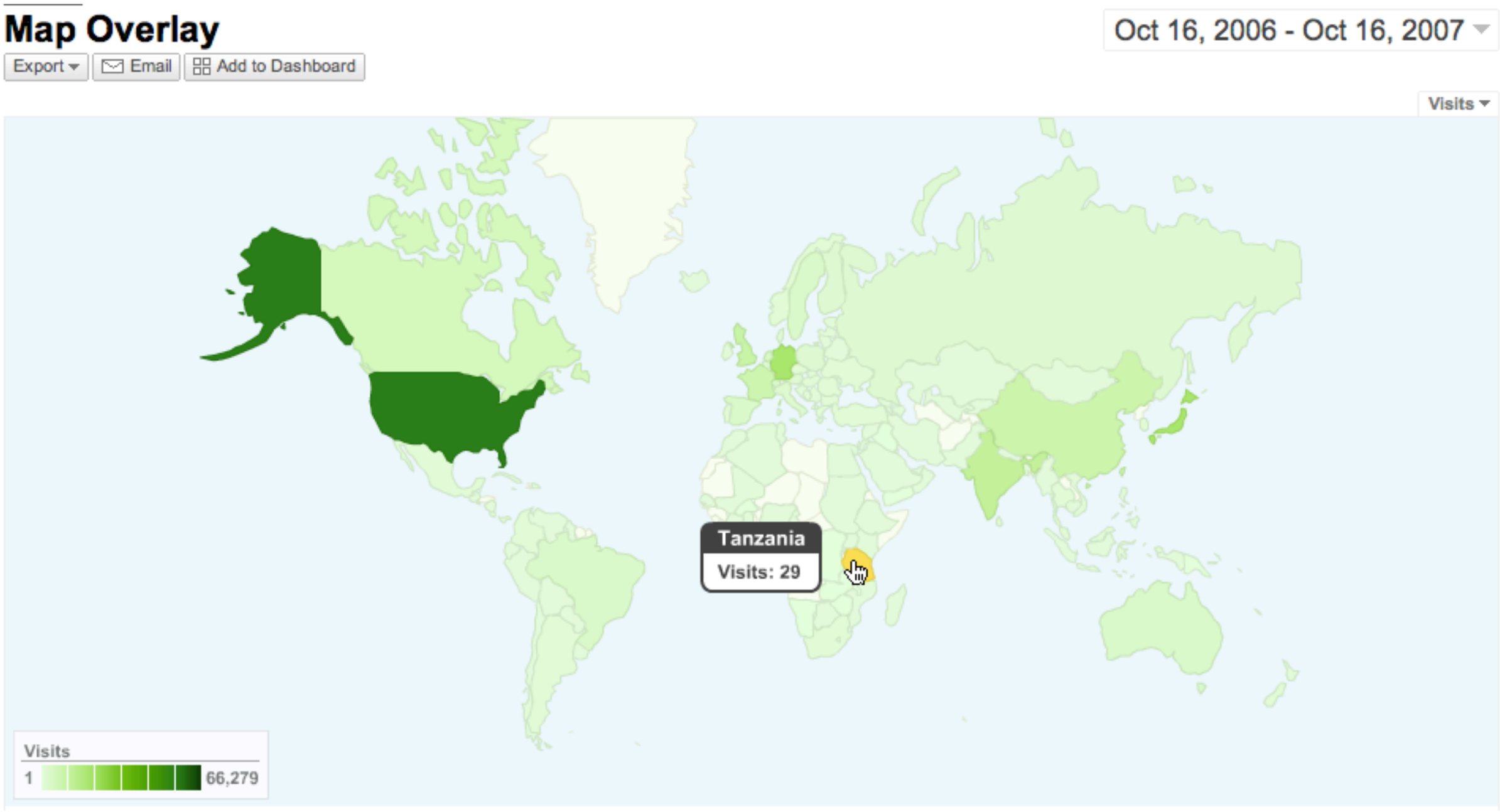
# Google

- Google runs FindBugs over all Java code in their main repository
- Check out poster for more details
- As of this summer
  - 1,663 issues identified and reviewed
  - 1,190 reported as bugs to developers
  - 805 fixed by developers

# EBay

- Using 2 developers to audit/review FindBugs warnings was 10 times more effective at finding P1 bugs than using two testers
- Has put a lot of work into reviewing which bug patterns are significant for EBay
- Check out OOPSLA poster: *Understanding the Value of Program Analysis Tools*, Cierra Jaspan, I-Chin Chen, Anoop Sharma

# Widely used; more than 450,000 downloads



# Bug Categories

- Correctness - the code seems to be clearly doing something the developer did not intend
- Bad practice - the code violates good practice
- Dodgy code - the code is doing something unusual that may be incorrect
- Multithreaded correctness
- Potential performance problems
- Malicious code vulnerability

# Bug Patterns

- Some big, broad and common patterns
  - Dereferencing a null pointer
  - An impossible checked cast
  - Methods whose return value should not be ignored
- Lots of small, specific bug patterns, that together find lots of bugs
  - Every Programming Puzzler
  - Every chapter in *Effective Java*
  - Many postings to http://thedailywtf.com/

# Analysis Techniques

Whatever you need to find the bugs

- Local pattern matching
  - If you invoke `String.toLowerCase()`, don't ignore the return value
- Intraprocedural dataflow analysis
  - Null pointer, type cast errors
- Interprocedural method summaries
  - This method always dereferences its parameter
- Context sensitive interprocedural analysis
  - Interprocedural flow of untrusted data
    - SQL injection, cross site scripting

# Categories, ranking, use cases

- Every tool has categories, rules/patterns, priorities
- You can generally customize what you want to look at
- Sometimes, you want to do a code audit of a newly written module with 1,000 lines of code
  - and sometimes you want to scan 1,000,000 lines of code that has been in production for a year
- Different use cases require different tunings, different tools

# Correctness issues

## Stuff you *really* want to look at

- In FindBugs, we reserve the Correctness category for issues we are most confident are wrong
  - code does something the developer didn't intend
- Many of the other categories reflect correctness issues
- But correctness issues are the things we think you should look at when scanning that million line code base
- low false positive rate, few low impact bugs

## ... Students are good bug generators

- Student came to office hours, was having trouble with his constructor:

```
/** Construct a WebSpider */
public WebSpider() {
        WebSpider w = new WebSpider();

    }
```

- A second student had the same bug
- Wrote a detector, found 3 other students with same bug

# Infinite recursive loop

## ... Students are good bug generators

- Student came to office hours, was having trouble with his constructor:

```
/** Construct a WebSpider */
public WebSpider() {
        WebSpider w = new WebSpider();

    }
```

- A second student had the same bug
- Wrote a detector, found 3 other students with same bug

# Double Check Against JDK1.6.0-b13

- Found 5 infinite recursive loops
- Including one written by Joshua Bloch

```
public String foundType() {
    return this.foundType();
}
```

- Smart people make dumb mistakes
  - 27 across all versions of JDK, 40+ in Google's Java code
- Embrace and fix your dumb mistakes

# Does this code contain a null pointer bug?

```
String s = null;
if (x > 0) s = “x”;
if (y > 0) s = “y”;
return s.hashCode();
```

# Finding Null Pointer Bugs

- FindBugs looks for a statement or branch that, if executed, guarantees a null pointer exception
- Either a null pointer exception could be thrown, or the program contains a statement/branch that can't be executed
- Could look for exceptions that only occur on a path
  - e.g., if x <= 0 and y <= 0, then a NPE will be thrown
  - but would need to worry about whether that path is feasible
  - FindBugs *doesn't* do this.

# What about this code?

```
String s = null;

if (x > 0) s = “positive”;

if (x == 0) s = “zero”;

if (x < 0) s = “negative”;

return s.hashCode();
```

# FindBugs picks the low hanging fruit

- FindBugs doesn't try to determine if paths are feasible
  - you can't do a perfect job
  - doing a good job is hard
- We can find lots of bugs without checking for feasible paths
  - but assuming that all statements and branches are feasible

# Examples of null pointer bugs

## simple ones

//com.sun.corba.se.impl.naming.cosnaming.NamingContextImpl

```
if (name != null || name.length > 0)
```

//com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser

```
if (part == null | part.equals(""))
```

// sun.awt.x11.ScrollPanePeer

```
if (g != null)
   paintScrollBars(g,colors);
g.dispose();
```

# Null Pointer Bugs Found by

JDK1.6.0-b105

- 109 statements/branches that, if executed, guarantee NPE
  - We judge at least 54 of them to be serious bugs that could generate a NPE on valid input

- Most of the others were deemed to be unreachable branches or statements, or reachable only with erroneous input
  - Only one case where the analysis was wrong

# Redundant Check For Null

## Also known as a reverse null dereference error

- Checking a value to see if it is null
  - When it can't possibly be null

// java.awt.image.LoopupOp, lines 236-247

```
public final WritableRaster filter(
        Raster src, WritableRaster dst) {

    int dstLength = dst.getNumBands();

    // Create a new destination Raster,
    // if needed

    if (dst == null)
```

# Redundant Check For Null

## Is it a bug or a redundant check?

- Check the JavaDoc for the method

- Performs a lookup operation on a **`Raster`**.
  - If the destination **`Raster`** is **`null`**,
  - a new **`Raster`** will be created.

- Is this case, a bug
  - particularly look for those cases where we know it can't be null because there would have been a NPE if it were null

# Bad Method Invocation

- Methods whose return value shouldn't be ignored
  - Strings are immutable, so functions like **`trim()`** and **`toLowerCase()`** return new String
- Dumb/useless methods
  - Invoking **`toString`** or equals on an array
- Lots of specific rules about particular API methods
  - Hard to memorize, easy to get wrong

# Examples of bad method calls

// com.sun.rowset.CachedRowSetImpl

```
if (type == Types.DECIMAL || type == Types.NUMERIC)
        ((java.math.BigDecimal)x).setScale(scale);
```

// com.sun.xml.internal.txw2.output.XMLWriter

```
try { ... }
catch (IOException e) {
  new SAXException("Server side Exception:" + e);
  }
```

# Type Analysis

- Impossible checked casts
- Useless calls
  - equals takes an **`Object`** as a parameter
    - but comparing a **`String`** to **`StringBuffer`** with **`equals(...)`** is pointless, and almost certainly not what was intended
  - **`Map<K,V>.get`** also takes an **`Object`** as a parameter
    - supplying an object with the wrong type as a parameter to **`get`** doesn't generate a compile time error

# Lots of Little Bug Patterns

- checking if `d == Double.NaN`
- Bit shifting an `int` by a value greater than 31 bits
- Checking to see if a `char` variable is equal to -1

# When Bad Code Isn't A Bug

- Static analysis tools will sometimes find ugly, nasty code
  - that can't cause your application to misbehave
- Cleaning this up is a good thing
  - makes the code easier to understand and maintain
- But for ugly code already in production
  - sometimes you just don't want to touch it
- We've found more cases like this than we expected

# False positive isn't a useful term

- In the context of static analysis, I don't believe that "false positive" is a useful term
- Static analysis doesn't know what your application is supposed to do
- In many cases:
  - the analysis is correct,
  - the code is dumb, but
  - the dumb code can't cause your application to misbehave

# When Bad Code Isn't A Bug

bad code that does what it was intended to do

```
// com.sun.jndi.dns.DnsName, lines 345-347
 if (n instanceof CompositeName) {
     // force ClassCastException
     n = (DnsName) n;
 }
```

```
// sun.jdbc.odbc.JdbcOdbcObject, lines 85-91
void dumpByte (byte b[],int len){
  ...
  if ((b[offset] < 32) || (b[offset] > 128))
   asciiLine += ".";
```

# When Bad Code Isn't A Bug

## Code that shouldn't go wrong

```
// com.sun.corba.se.impl.dynamicany.DynAnyComplexImpl
String expectedMemberName = null;
try {
    expectedMemberName
      = expectedTypeCode.member_name(i);
} catch (BadKind badKind) { // impossible
} catch (Bounds bounds) { // impossible
}
if ( !(expectedMemberName.equals(memberName) ... ))
  {
```

# When Bad Code Isn't A Bug

## When you are already doomed

// com.sun.org.apache.xml.internal.security.encryption.XMLCiper

// lines 2224-2228

```
if (null == element) {
   //complain
}
String algorithm = element.getAttributeNS(...);
```

# Overall Correctness Results From FindBugs

*Evaluating Static Analysis Defect Warnings On Production Software*, ACM 2007 Workshop on Program Analysis for Software Tools and Engineering

- JDK1.6.0-b105
  - 379 correctness warnings
    - we judge that at least 213 of these are serious issues that should be fixed
- Google's Java codebase (#'s below updated since paper)
  - over a 6 month period, using various versions of FindBugs
  - 1,663 warnings
  - 1,190 filed as bugs
  - 805 fixed in code

# Other Categories

- Bad practice
- Dodgy code
- Multithreaded correctness
- Performance
- Vulnerability to malicious code

# Bad Practice

- A class that defines an equals method but inherits hashCode from Object
  - Violates contract that any two equal objects have the same hash code
- equals method doesn't handle null argument
- Serializable class without a serialVersionUID
- Exception caught and ignored
- Broken out from the correctness category because I never want a developer to yawn when I show them a "correctness" bug

# Fixing hashCode

- What if you want to define equals, but don't think your objects will ever get put into a HashMap?
- Suggestion:

```
public int hashCode() {
   assert false
     : "hashCode method not designed";
   return 42;
 }
```

# Use of Unhashable Classes

- FindBugs previously reported all classes that defined equals but not hashCode as a correctness problem
  - but some developers didn't care
- Now reported as bad practice
  - but separately report use of such a class in a HashMap/HashTable as a correctness warning
  - such use is almostly certainly unintended/surprising and likely to result in errors

# Dodgy code

- Dead local store - a value is stored into a local variable, but that value is never used
- Use of non-short circuit boolean logic
- Switch statement fallthrough
- Branch where code on both branches is identical

# Multithreaded correctness

- Inconsistent synchronization - a lock is held most of the time a field is accessed, but not always
- Problems with wait/notify - e.g., call to wait() not in loop
- thread unsafe lazy initialization of static field

# Performance

- Unused field
- Invocation of Boolean or Integer constructors
- Using hashCode or equals method on a URL
- final constant field that could be made static
- Loop with quadratic string concatenation
- Inner class that could be made static

# Vunerability to Malicious code

- public static non-final fields
- public static final fields that reference mutable objects
- Methods that don't defensively copy mutable arguments before storing them into fields
- Methods that don't defensively copy mutable values stored in fields before returning them

# Questions?