Defective Java™ Code: Turning WTF code into a learning experience

William Pugh, Professor, Univ. of Maryland

TS-6589

Java

Sun microsystems

Examine some defective Java™ code to become a better developer, and discuss how to turn mistakes into learning experiences.

GOAL

Learning from Mistakes

- Both FindBugs and Java Puzzlers work best when inspired by real bugs
- Many different sources for bugs
  - List of bugs fixed in a build
  - Bug reports against a compiler or library marked as Not a bug
    - Indicate something that confused people
  - Your own mistakes
  - Issues and controversies in discussions of good programming practice
  - The Daily WTF Code Snippet of the day

Agenda

- Waiting for the end
- Synchronizing
- Supported after all
- Formatting
- Equality
- The end

The Daily WTF CodeSOD, 2006-11-02 (*abridged*)

```
while (!_validConnection) {
  StringBuffer _stringBuffer = new StringBuffer();
  try {
    while (true) {
      char _char;
      _stringBuffer.append(
          _char = (char)_inputStream.read());
      if (_char == -1) break;
      else if (_char == '\r') {
           …break on "\r\n\r\n"… }
      }
    } catch (OutOfMemoryError error) {
      // received a bad response, try it again!
      continue;
    }
```

Code snippet from commercial HTTP client

- Reported by Tobias Tobiasen
  - The reply from their support was something along the lines of "We are unable to see any problems in the code, please supply a test case."
- Problems
  - Casting int to char (only handles ASCII)
  - Catching OutOfMemoryErrror
  - _char will never be equal to -1
    - char is the only unsigned integral type
    - Casting -1 (indicating EOF) to char gives 0xffff

The Daily WTF CodeSOD, 2006-11-02 (*abridged*)

```
while (!_validConnection) {
  StringBuffer _stringBuffer = new StringBuffer();
  try {
    while (true) {
      char _char;
      _stringBuffer.append(
          _char = (char)_inputStream.read());
      if (_char == -1) break;
      else if (_char == '\r') {
           …break on "\r\n\r\n"… }
      }
    } catch (OutOfMemoryError error) {
      // received a bad response, try it again!
      continue;
    }
```

Wrote a detector for FindBugs

Library or application	# of methods
JDK, most versions including 1.5 – 1.7	1
Java Server Faces	2
Eclipse, various versions	5
apache bzip2 tool	2
apache structs upload	2
apache xalan	2

Source: FindBugs

Code samples

```
sun.net.httpserver.ChunkedInputStream:
  while ((c=(char)in.read())!= -1) { … }

org.eclipse.jdt.core.dom.CharacterLiteral (5 occurrences)
  nextChar = (char) scanner.getNextChar();
  if (nextChar == -1) { … }

org.eclipse.pde.internal.core.content.BundleManifestDescriber
  int first = (input.read() & 0xFF);
    //converts unsigned byte to int
  int second = (input.read() & 0xFF);
  if (first == -1 || second == -1) return null;
```

Morals

- Methods that return -1 for EOF can be tricky
  - Need to check for -1 before doing anything with result.
- Code is rarely tested for unexpected EOF
  - May need to use a mock framework
- Don't assume that any mistake is so unique that no one else could have made it

Agenda

- Waiting for the end
- Synchronizing
- Supported after all
- Formatting
- Equality
- The end

Be careful what you synchronize on

- Bug found in Jetty-6.1.3 BoundedThreadPool

```
private final String _lock = "LOCK";
...
synchronized(_lock) {
   ...
   }
```

- String constants are interned and shared in the Java™ Virtual Machine (JVM™)
- Filed and fixed as http://jira.codehaus.org/browse/JETTY-352

The fix

- Synchronize on a raw Object

```
private final Object _lock = new Object();
…
synchronized(_lock) {
…
}
```

- One of the only good uses for a raw Object

They missed another occurrence

org.mortbay.jetty.security.Credential.MD5:

```
public static final String __TYPE="CRYPT:";
...
synchronized(__TYPE) { … }
```

- Filed and fixed as http://jira.codehaus.org/browse/JETTY-362

More synchronization oddities

javax.management.relation.RelationService:

```
private Long myNtfSeqNbrCounter = new Long(0);
private Long getNotificationSequenceNumber() {
  Long r;
  synchronized(myNtfSeqNbrCounter) {
   r = new Long(myNtfSeqNbrCounter.longValue() + 1);
   myNtfSeqNbrCounter = new Long(r.longValue());
   }
  return r;
 }
```

What goes wrong

- Doesn't provide mutual exclusion
  - One thread could synchronize on an old value, another thread on a newer value, both be in the critical region at the same time
    - could result in duplicated values being handed out
  - Long objects aren't shared, but only a "remove explicit autoboxing" quickfix away from synchronizing on interned Long objects
- Key mistakes
  - Synchronizing on (the contents of) a field in order to guard access to changing the field
  - Synchronizing on shared constants (Strings, boxed primitives)

Synchronization on `getClass()`

`java.awt.Panel:` (JDK™ version 1.6.0)

```
private static int nameCounter = 0;

String constructComponentName() {
    synchronized (getClass()) {
     return base + nameCounter++;
    }
}
```

- Doesn't provide mutual exclusion in subclasses,
  - synchronize on **`Panel.class`**, or use **`AtomicInteger`**

Synchronization Morals

- Lots of mistakes
  - FindBugs reports 31 related issues in Sun's JDK version 1.6.0
- Testing is relatively useless for finding concurrency errors
- Don't think about what to synchronize on
  - Instead, think about who to synchronize with
  - Insure that you are synchronizing on something that they all synchronize on
    - And avoid synchronizing on something that unrelated code might synchronize on

Agenda

- Waiting for the end
- Synchronizing
- Supported after all
- Formatting
- Equality
- The end

Supported after all?

com.sun.corba.se.impl.io.IIOPInputStream:

```
protected final Class
  resolveClass(ObjectStreamClass v)
  throws IOException, ClassNotFoundException {
        throw new IOException(
              "Method resolveClass not supported");
  }
```

- Class extends **java.io.ObjectInputStream**
- Surprisingly, calling **resolveClass** works same as in OIS, doesn't throw exception

Does it override the superclass method?

```
java.io.ObjectInputStream:

 protected Class<?>
    resolveClass(ObjectStreamClass desc)
    throws IOException, ClassNotFoundException { … }

com.sun.corba.se.impl.io.IIOPInputStream:

 protected final Class
  resolveClass(ObjectStreamClass v)
  throws IOException, ClassNotFoundException { … }
```

Look at those elided imports

```
com.sun.corba.se.impl.io.IIOPInputStream:

import com.sun.corba.se.impl.io.ObjectStreamClass;

protected final Class
   resolveClass(ObjectStreamClass v)
   throws IOException, ClassNotFoundException { … }
```

- Parameter types are different: same simple name, different packages
- Doesn't override method in superclass

Morals

- `@Override` is your friend
  - Do an autofix to apply it uniformly throughout your code base
  - If you don't see it somewhere it should appear, something is wrong
- Minimize overloading simple names
  - Autocompletion makes it far too easy to get the wrong one
- Avoid overloading the simple name of a superclass
  - Given a class `alpha.FooBar`
  - don't define a subclass `beta.FooBar`
  - Too many possibilities for collisions

Agenda

- Waiting for the end
- Synchronizing
- Supported after all
- Formatting
- Equality
- The end

Formatting a date

```
org.jfree.data.time.Day:
 protected static final DateFormat DATE_FORMAT

  = new SimpleDateFormat("yyyy-MM-dd");


public static Day parseDay(String s) {
   try {
          return new Day (
           Day.DATE_FORMAT.parse(s)
           );
   } catch (ParseException e1) { … }
```

`DateFormat` is not thread safe

- A `DateFormat` stores the date to be formatted/parsed into an internal field
- If two threads try to simultaneously parse or format dates, you may get runtime exceptions or incorrectly formatted dates
- Not just a theoretical possibility
  - Has caused field failures
  - Easy to replicate with simple test code
- Reported to the FindBugs project by someone who got bit by this bug

Morals

- `DateFormat` (and subtypes) are not thread safe
- Immutability is your friend
- When designing an API, understand your use cases.
  - If the class feels like an immutable constant, people will use it like an immutable constant
  - Even if the Javadoc™ tool says not too

Agenda

- Waiting for the end
- Synchronizing
- Supported after all
- Formatting
- Equality
- The end

Equality

```
java.sql.Timestamp:
public class Timestamp extends java.util.Date {
  …
  public boolean equals(java.lang.Object ts) {
      if (ts instanceof Timestamp) {
        return this.equals((Timestamp)ts);
      } else {
        return false;
      }
    }
```

- Not symmetric

Requirements for equals

- `equals(null)` returns false
- If `x.equals(y)`,
  then `x.hashCode() == y.hashCode()`
- equals is reflexive, symmetric and transitive

```
Date d = new Date();
Timestamp ts = new Timestamp(d.getTime());
System.out.println(d.equals(ts)); // true
System.out.println(ts.equals(d)); // false
```

Symmetry is important

- A primary use of **`equals`** methods is in containers (e.g., Sets or Maps).
- Various methods might invoke **`equals`** as **`a.equals(b)`** or **`b.equals(a)`**
- Non-symmetric **`equals`** can produce confusing and hard to reproduce results

The equals debate

- Should equals use **`instanceof`** or **`getClass()`** to check for a compatible argument?
- In class Foo:

```
public void equals(Object o) { // use instanceof
     if (!(o instanceof Foo)) return false;
     …
     }
public void equals(Object o) { // use getClass
     if (o == null
       || this.getClass() != o.getClass())
          return false;
     …
     }
```

Poll

a) Weren't aware that there was a debate
b) Think you should use **`instanceof`**
c) Think you should use **`getClass()`**
d) Think both are OK in some situation
e) Are confused about what to use

Problems

- If you use **`instanceof`** and override the equals method, you may break symmetry
- If you use **`getClass()`**, it is impossible for to define a subclass such that instances could be equal to an instance of the base class

Designing for subtyping is hard

- This issue only matters when implementing a class that is designed to be subclassed
- Doing so is hard in general
  - even more so when you anticipate arbitrary third parties extending the class
- Specifying how equals should behave is only one of many tricky decisions you will have to make

Defining equality is hard

- “the problem with equals as a class method is that there are too many senses of equals for only one such method to support, and the author of a class won't have all of them in mind.”
  - Doug Lea, 20+ years ago

Possible semantics for equality

- Object equality: no two distinct objects are equal
  - the definition inherited from class Object
  - useful and sufficient more often than you would suspect
- Value equality: An object represents an abstract value
  - any two objects that represent the same value should be equal
  - An **`ArrayList`** and a **`LinkedList`** representing {1, 2, 3} are equal
- Behavior equality
  - objects are equal if you can't distinguish them based on their behavior

Behavioral equality is subtle

- Object and Value equality are fairly straightforward
- Using **`getClass()`** generally implies behavioral equality
- Behavioral equality is the most subtle, and to some confusing or just wrong
  - how can B be a subclass of A if it is impossible for a B to be equal to an A?
- With **`getClass()`**, any extension of a class splits the equality relation
  - Even an extension to just add some performance monitoring

`getClass()` equality breaks Hibernate

- Hibernate creates proxy classes for persistent object model classes
  - value semantics is clearly intended and required
  - various situations can cause mixing of pure and proxied versions of object model classes
- Use of `getClass()` in equals methods for Hibernate objects will cause failures

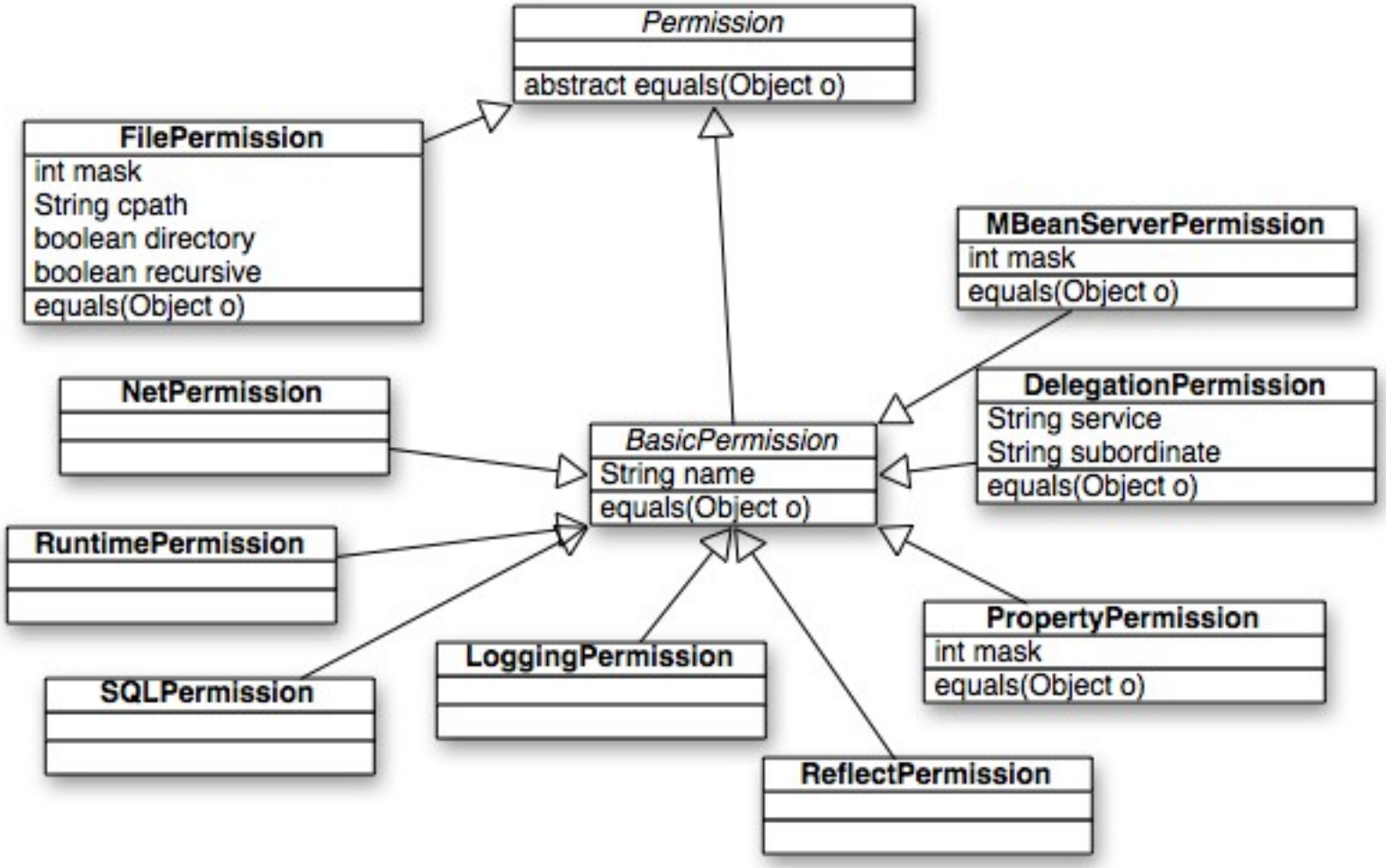
Using `instanceof` in equals

- If you use **`instanceof`** in equals, and override equals in a subclass
- you must not change the semantics of equals
- the only thing you are allowed to do is use a more efficient algorithm, or instrument for debugging or performing monitoring

Using `getClass()` in an abstract base class

- `getClass()` is sometimes used in an abstract base class
  - compare any common fields/values and the class
- Example: `java.security.BasicPermission`
  - two instances are equal if they are of the same class and have the same canonical name
  - No `BasicPermission` objects exist, establishes equality partition between subclasses
    - some subclasses check additional fields

BasicPermission uses getClass()



A flexible alternative?

```
class BPermission {

  protected Class<?> getEqualityClass() {
     return this.getClass(); }

  public boolean equals(Object obj) {
     if (!(obj instanceof BPermission )
       || getEqualityClass()
          != ((BPermission)obj).getEqualityClass())
            return false;
     … }
```

Defining a `getEqualityClass()`

- Provides `getClass()` like behavior for subclasses by default
  - but with the ability to override that in subclasses that want to allow equality with their superclass
- But just defining an `instanceof` equals method in each concrete subclass has the same effect
  - more lines of code, but probably more efficient

JDK version 1.6.0-b105 Statistics

- 488 **`instanceof`** equals
  - 95 in classes with subclasses
  - 37 overridden in a subclass
  - 10 overridden in a way that breaks symmetry
- 6 **`getClass()`** in abstract classes
- 11 **`getClass()`** in non-abstract classes
  - none of which have any subclasses

Eclipse 3.4M4 statistics

- 531 **`instanceof`** equals
  - 87 in classes with subclasses
  - 27 overridden in a subclass
  - 10 overridden a way that breaks symmetry
- 13 **`getClass()`** in abstract classes
- 72 **`getClass()`** in non-abstract classes
  - only 8 of which have any subclasses

Morals

- Document your equals semantics
- If you use **`instanceof`**, consider declaring the **`equals`** method final
  - if you override such an equals method, you must be very careful to avoid breaking symmetry
    - FindBugs will find most violations
- Using **`getClass()`** avoids this, but limits for all time the behaviors you can implement
  - makes it impossible to extend a class without effecting equality

Agenda

- Waiting for the end
- Synchronizing
- Supported after all
- Formatting
- Equality
- The end

Defective code is a learning opportunity

- Few mistakes are unique
  - e.g., who would have thought that an equals method that always returns false would be written more than once?
    - 2 in JDK version 1.6.0, 4 times in Eclipse
    - plus one equals method in Eclipse that always returns true
- Encourage you to set up an experience factory for mistakes
  - when a costly mistakes happens, evaluate what could be done to prevent other similar mistakes
  - perhaps a coding or testing practice, perhaps static analysis

Learning opportunities

- Students are very good mistake generators
  - and more often than you suspect, the mistakes made by students produce detectors that find mistakes in production code
    - although often manifested in different ways and for different reasons
- I've gotten very good at catching my own mistakes and turning them into bug patterns
  - I sometimes recognize a mistake before I compile the code
    - sometimes I'll even recognize it before I finish typing it
  - And I've gotten religion about dissecting mistakes that got committed

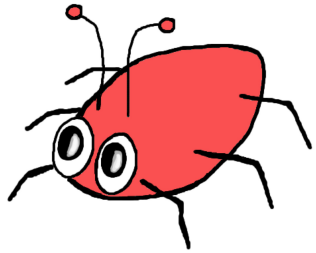
Writing your own bug detectors

- Writing bug detectors in PMD or Jackpot is fairly easy
  - Can just define an XPath expression over the syntax tree in PMD
- Writing bug detectors in FindBugs is more difficult
  - requires understanding Java class files
  - understanding the FindBugs APIs
    - several different analysis frameworks, grown over time
  - But provides deeper analysis

Announcing the FindBugs bug competition

- An easy way to turn your bug patterns into FindBugs bug detectors
- Submit examples of real code containing real bugs to FindBugs discussion mailing list
  - If we find it plausible and not to hard to detect, we'll do the bug detector implementation
    - or you can implement your detectors yourself

FindBugs bug competition



- Prizes for best bug patterns, deadline 8/31/08
  - Bug patterns where at least ¼ of the issues reported are code deemed to be should fix or must fix
    - not just stupid code, but a problem that could cause the code to misbehave
  - Evaluated based on total number of serious problems found and on should/must fix rate
- $200 for best bug pattern, 3 $100 prizes for runners up
  - evaluated by the FindBugs team/contributors, who aren't eligible
- All contributors get credit in the bug pattern

THANK YOU

Defective Java™ Code: Turning WTF code into a learning experience

William Pugh, Professor, Univ. of Maryland

TS-6589

Java

Sun microsystems