Applied Static Analysis
2016

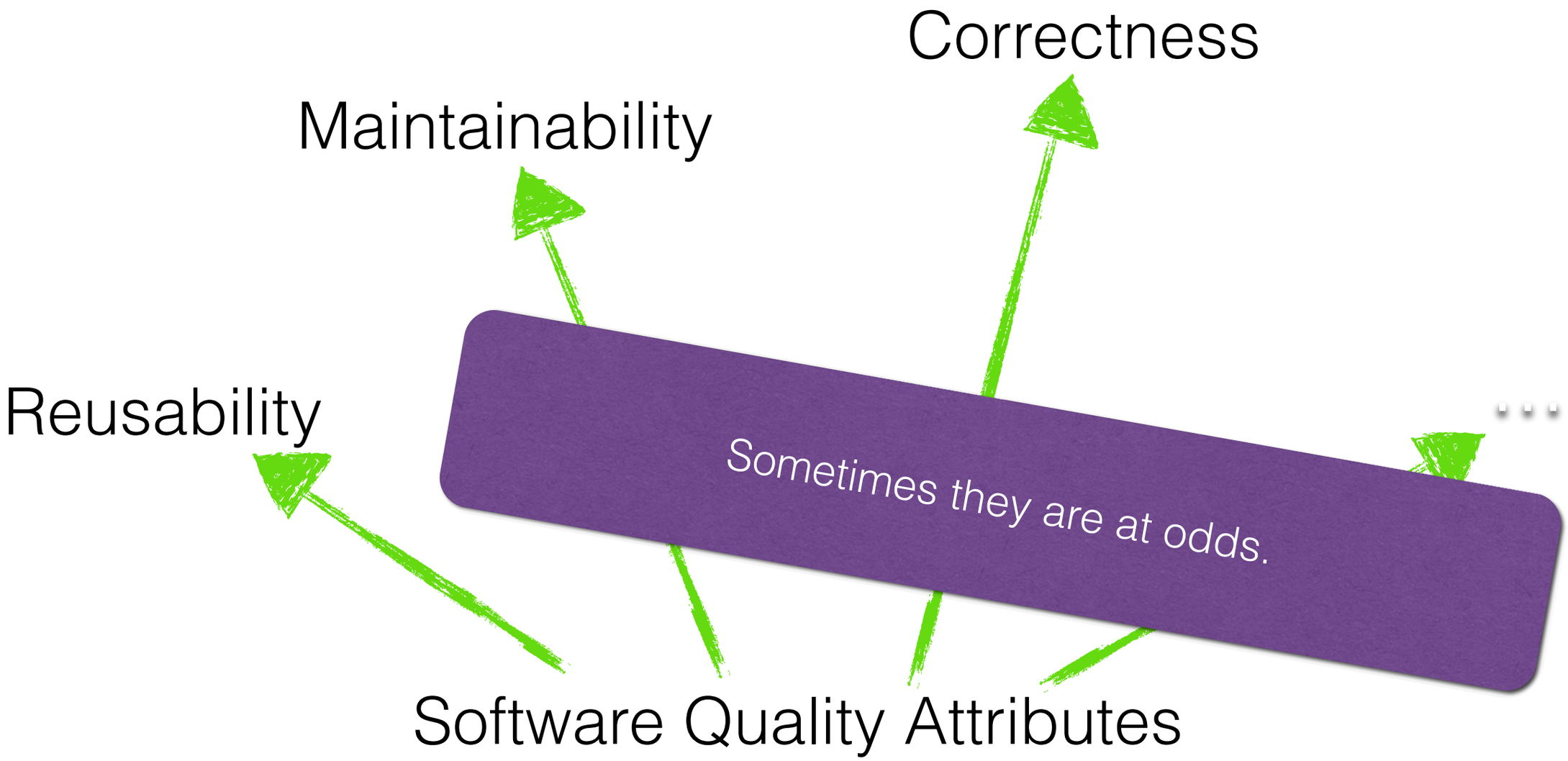
Dr. Michael Eichberg (Organizer)
Johannes Lerch, Ben Hermann, Sebastian Proksch, Karim Ali Ph.D.

Software Technology Group
TECHNISCHE UNIVERSITÄT DARMSTADT

Static Analysis Tools

(A BRIEF OVERVIEW)

Static Analyses are done to Improve Software Quality!



Static Analyses - Classified

	Reliability	Maintainability	Correctness	Efficiency
lightweight static analyses	✓	✓	✓	
semi formal methods			✓	
formal methods			✓	
structure analyses	✓	✓		
style conformance checking		✓		
architecture conformance checking	✓	✓		

code clone detection

violations of best practices/ bug patterns

there are no guarantees

there are no guarantees

there will be proofs

detects data- and control-flow issues

application and domain independent

Static Analysis Tools
A Few (Well Known)

- FindBugs
  Lightweigt static analyses on top of Java Bytecode.

- PMD
  Lightweight static analyses on top of the AST using Java Visitors or XPath based rules.

- CheckStyle
  Lightweight static analyses on top of the AST using Java Visitors.

- CheckerFramework
  Static analyses using pluggable types.

- ConQAT
  Code Clone Detection.

- Julia
  Static anaylsis of Java and Android code.

Static Analysis Tools
A Few (Well Known)

- JDepend
  Structural analysis on top of Java Bytecode.

- DependencyFinder
  Structural analysis on top of Java Bytecode.

- Stan4J
  Structural analysis on top of Java Bytecode.

- Sonargaph (SonarJ)
  Analyzes the structure of applications.

Static Analysis Tools
A Few (Well Known)

- ESC/Java2
  Formal verification using JML Annotations.
- Key
  Formal verification.
- …

On the Challenge of Finding Code Smells

Java

```
static int noInc(int i) {
    return i++;
}
```

How to find such issues?

Bytecode

PC	Instruction
0	iload_0
1	iinc 0 1
4	ireturn

Here, a register value/local variable is incremented, but the incremented value is not used afterwards.

Finding such issues is very simple!

Scan the bytecode for the instruction pattern:
`*, iinc, ireturn, *`

On the Challenge of Finding Code Smells

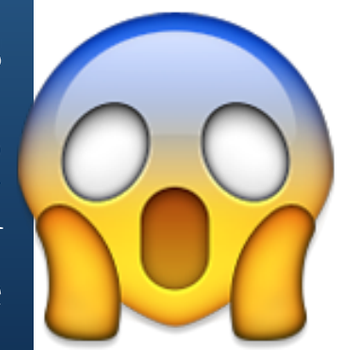
Java

```
static BigDecimal noScaling(BigDecimal d) {
    d.setScale(1002);
    return d;
}
```

Where is the issue?

[…]
Note that since BigDecimal objects are immutable, calls of this method do not result in the original object being modified, contrary to the usual convention of having methods named setX mutate field X. **Instead, setScale returns an object with the proper scale; the returned object may or may not be newly allocated.**

(JavaDoc)



On the Challenge of Finding Code Smells

Java

```
static BigDecimal noScaling(BigDecimal d) {
    d.setScale(1002);
    return d;
}
```

Bytecode

PC	Instruction
0	aload_0
1	sipush 1002
4	invokevirtual java.math.BigDecimal { java.math.BigDecimal setScale (int) }
7	pop
8	aload_0
9	areturn

Here, the return value of a method call is ignored. But is this - in general - an issue?

No!

Finding such issues is hard(er)!

On the Challenge of Finding Code Smells

javax.crypto.CryptoPolicyParser (Java 8u25)

```
494     private boolean More ...isConsistent(String alg, String exemptionMechanism,
495             Hashtable<String, Vector<String>> processedPermissions) {
496         String thisExemptionMechanism =
497             exemptionMechanism == null ? "none" : exemptionMechanism;
498
499         if (processedPermissions == null) {
500             processedPermissions = new Hashtable<String, Vector<String>>();
501             Vector<String> exemptionMechanisms = new Vector<>(1);
502             exemptionMechanisms.addElement(thisExemptionMechanism);
503             processedPermissions.put(alg, exemptionMechanisms);
504             return true;
505         }
506
507         if (processedPermissions.containsKey(CryptoAllPermission.ALG_NAME)) {
508             return false;
509         }
510
[…]
525     }
```

Finding such issues is requires sophisticated analyses.

Where is the issue?

Examples of Data- and Control-Flow Dependent Code Smells in Open JDK 8u25

Class javax.swing.SwingUtilities

```
945     private static String layoutCompoundLabelImpl(
946         JComponent c,
947         FontMetrics fm,
948         String text,
949         Icon icon,
950         int verticalAlignment,
951         int horizontalAlignment,
952         int verticalTextPosition,
953         int horizontalTextPosition,
954         Rectangle viewR,
955         Rectangle iconR,
956         Rectangle textR,
957         int textIconGap)
958     {
[...]
975         boolean textIsEmpty = (text == null) || text.equals("");
976         int lsb = 0;
977         int rsb = 0;
```

[... MANY LINES]

always false

```
1121         if (rsb > 0) {
1122             textR.width -= rsb;
1123         }
1124
1125         return text;
1126     }
```

- The value is constant in the method.
- A reason why this issue was not detected is probably the length of the method. (The definition of “rsb” is not visible on the screen when you scroll to the if statement.)

Class javax.print.ServiceUI

- A test “null instanceof XXX” is always false.
- The developer probably wasn’t aware of the precise semantics of instanceof checks on null values.

```
154     public static PrintService printDialog(GraphicsConfiguration gc,
155                                            int x, int y,
156                                            PrintService[] services,
157                                            PrintService defaultService,
158                                            DocFlavor flavor,
159                                            PrintRequestAttributeSet attributes)
160         throws HeadlessException
161     {
  […]
191         Window owner = null;
192
193         Rectangle gcBounds = (gc == null) ?  GraphicsEnvironment.
194             getLocalGraphicsEnvironment().getDefaultScreenDevice().
195             getDefaultConfiguration().getBounds() : gc.getBounds();
196
197         ServiceDialog dialog;
198         if (owner instanceof Frame) {
199             dialog = new ServiceDialog(gc,
200                                        x + gcBounds.x,
201                                        y + gcBounds.y,
202                                        services, defaultIndex,
203                                        flavor, attributes,
204                                        (Frame)owner);
205         } else {
206             dialog = new ServiceDialog(gc,
207                                        x + gcBounds.x,
208                                        y + gcBounds.y,
209                                        services, defaultIndex,
210                                        flavor, attributes,
211                                        (Dialog)owner);
212         }
```

always false

Class com.sun.org.apache.regexp.internal.RECompiler

Method int closure(int[] flags) throws RESyntaxException

- The variable greedy will always be true in line 1091.
- Here, the developer probably wasn't aware of the first test.

```
993         if (greedy)
994         {
995             // Actually do the closure now
996             switch (closureType)
997             {
…
1089        case '*':
1090
1091            if (!greedy)
1092            {
1093                break;
1094            }
…
```

Here are ~100 lines of code!

Class com.sun.java.util.jar.pack.Fixups

Method boolean storeDesc(int loc, int fmt, int desc)

- After a cast of a value to a byte value the value will always be in the range [-128,127]. Hence, the comparison with 999 (!=) will always be true.

```
[…]
216:      (bytes[loc+1]=(byte)bigDescs[BIGSIZE])!=999
[…]
```

always true

Class com.sun.org.apache.xpath.internal.functions.FuncSystemProperty

- the tests in line 76 and 81 are useless since the value is guaranteed to be larger than 0 due to the initial test in line 74.

```
60   public XObject execute(XPathContext xctxt) throws javax.xml.transform.TransformerException
[…]
74     if (indexOfNSSep > 0)
75     {
76       String prefix = (indexOfNSSep >= 0)
77                       ? fullName.substring(0, indexOfNSSep) : "";
[…]
81     propName = (indexOfNSSep < 0)
82                  ? fullName : fullName.substring(indexOfNSSep + 1);
[…]
```

Class com.sun.jmx.snmp.SnmpInt

- No int value can ever be smaller than the smallest representable value or can be larger than the largest representable value. (Line 253)

```
252     boolean isInitValueValid(int v) {
253         if ((v < Integer.MIN_VALUE) || (v > Integer.MAX_VALUE)) {
254             return false;
255         }
256         return true;
257     }
```

Class com.sun.media.sound.AuFileReader

- The first test “! (magic == AuFileFormat.AU_SUN_MAGIC)” is basically equivalent to “(magic ! = AuFileFormat.AU_SUN_MAGIC)” and, hence, makes the following checks useless. I.e., the complete check could be reduced to the same check “magic != AuFileFormat.AU_SUN_MAGIC”.

suspicious (useless) checks

```
72      public AudioFileFormat getAudioFileFormat(InputStream stream)
[…]
97          if (! (magic == AuFileFormat.AU_SUN_MAGIC) || (magic == AuFileFormat.AU_DEC_MAGIC) ||
98              (magic == AuFileFormat.AU_SUN_INV_MAGIC) || (magic == AuFileFormat.AU_DEC_INV_MAGIC) ) {
99
100             // not AU, reset the stream, place into exception, throw exception
101             dis.reset();
102             throw new UnsupportedAudioFileException("not an AU file");
103         }
104
105         if ((magic == AuFileFormat.AU_SUN_MAGIC) || (magic == AuFileFormat.AU_DEC_MAGIC)) {
106             bigendian = true;         // otherwise little-endian
107         }
108
```

Class javax.print.attribute.HashAttributeSet

- useless code hampers comprehension

```
public boolean containsValue(Attribute attribute) {
    return
       attribute != null &&
       attribute instanceof Attribute &&
       attribute.equals(attrMap.get(((Attribute)attribute).getCategory()));
}
```

Class com.sun.corba.se.impl.orbutil.ORBUtility

- The test in line 334 will always fail, because the test in line 332 already covers the case.
- The developer was probably not aware of the version numbers underlying the constants.

```
313     public static byte getEncodingVersion(ORB orb, IOR ior) {
[… comments]
321
322         if (orb.getORBData().isJavaSerializationEnabled()) {
323             IIOPProfile prof = ior.getProfile();
324             IIOPProfileTemplate profTemp =
325                 (IIOPProfileTemplate) prof.getTaggedProfileTemplate();
326             java.util.Iterator iter = profTemp.iteratorById(
327                                       ORBConstants.TAG_JAVA_SERIALIZATION_ID);
328             if (iter.hasNext()) {
329                 JavaSerializationComponent jc =
330                     (JavaSerializationComponent) iter.next();
331                 byte jcVersion = jc.javaSerializationVersion();
332                 if (jcVersion >= Message.JAVA_ENC_VERSION) {
333                     return Message.JAVA_ENC_VERSION;
334                 } else if (jcVersion > Message.CDR_ENC_VERSION) {
335                     return jc.javaSerializationVersion();
336                 } else {
337                     // throw error?
338                     // Since encodingVersion is <= 0 (CDR_ENC_VERSION).
339                 }
340             }
341         }
342         return Message.CDR_ENC_VERSION; // default
343     }
```

== 1

== 0

always false

this branch is never executed

Class com.sun.imageio.plugins.png.PNGMetadata
Method void mergeStandardTree(org.w3c.dom.Node)

```
1842     if (maxBits > 4 || maxBits < 8) {
1843         maxBits = 8;
1844     }
1845     if (maxBits > 8) {
1846         maxBits = 16;
1847     }
```

- Here, maxBits will always be 8 because the first condition (line 1842) is always fulfilled; every possible int value satisfies the condition “> 4” **or** “< 8”.
(the values 5,6,7,8,9,… satisfy the first part and the values -X,…, 3,4,5,6,7 satisfy the second part).
- The developer probably wanted to write “&&” in case of the first if statement.

Class sun.font.StandardGlyphVector

- “glyphs.length” will always be larger than or equal to 0; hence, the condition (as a whole) will never be true.

```
336     public int getGlyphCharIndex(int ix) {
337         if (ix < 0 && ix >= glyphs.length) {
338             throw new IndexOutOfBoundsException("" + ix);
339         }
```

Class sun.security.util.DerInputStream

```
545     static int getLength(int lenByte, InputStream in) throws IOException {
546         int value, tmp;
547
548         tmp = lenByte;
549         if ((tmp & 0x080) == 0x00) { // short form, 1 byte datum
550             value = tmp;
551         } else {                     // long form or indefinite
552             tmp &= 0x07f;
553
554             /*
555              * NOTE:  tmp == 0 indicates indefinite length encoded data.
556              * tmp > 4 indicates more than 4Gb of data.
557              */
558             if (tmp == 0)
559                 return -1;
560             if (tmp < 0 || tmp > 4)
561                 throw new IOException("DerInputStream.getLength(): lengthTag="
562                     + tmp + ", "
563                     + ((tmp < 0) ? "incorrect DER encoding." : "too big."));
564
565             for (value = 0; tmp > 0; tmp --) {
566                 value <<= 8;
567                 value += 0x0ff & in.read();
568             }
569         }
570         return value;
571     }
```

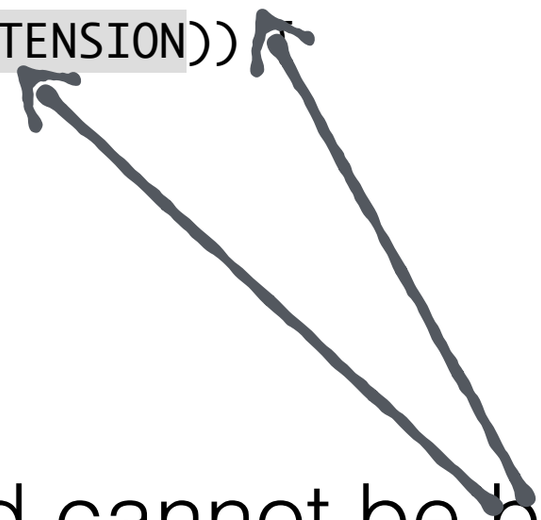
- tmp will always be larger than 0
- A binary and (&) of some value (line 552) with a positive value ensures that the resulting value is positive.
- Another instance of this kind of issue can also be found in: sun.security.jgss.GSSHeader.
  (100% copy and paste - even stated in the code!)

Class com.sun.org.apache.xerces.internal.impl.xs.XSComplexTypeDecl

Method boolean isDOMDerivedFrom(java.lang.String,java.lang.String,int)

- The && on line 280 probably should have been a || operator.
- Here, the formatting of the code is deceiving.

```
276         if (ancestorNS != null
277                 && ancestorNS.equals(SchemaSymbols.URI_SCHEMAFORSCHEMA)
278                 && ancestorName.equals(SchemaSymbols.ATTVAL_ANYTYPE)
279                 && (derivationMethod == DERIVATION_RESTRICTION
280                 && derivationMethod == DERIVATION_EXTENSION)) {
281             return true;
282         }
```



derivationMethod cannot be both…

Class com.sun.media.sound.SoftMixingClip

```
63  public int read(byte[] b, int off, int len) throws IOException {
64
65              if (_loopcount != 0) {
66                  int bloopend = _loopend * framesize;
67                  int bloopstart = _loopstart * framesize;
68                  int pos = _frameposition * framesize;
69
70                  if (pos + len >= bloopend)
71                      if (pos < bloopend) {
72                          int offend = off + len;
73                          int o = off;
74                          while (off != offend) {
75                              if (pos == bloopend) {
76                                  if (_loopcount == 0)
77                                      break;
78                                  pos = bloopstart;
79                                  if (_loopcount != LOOP_CONTINUOUSLY)
80                                      _loopcount--;
81                              }
82                              len = offend - off;
83                              int left = bloopend - pos;
84                              if (len > left)
85                                  len = left;
86                              System.arraycopy(data, pos, b, off, len);
87                              off += len;
88                          }
89                          if (_loopcount == 0) {
90                              len = offend - off;
91                              int left = bloopend - pos;
92                              if (len > left)
93                                  len = left;
94                              System.arraycopy(data, pos, b, off, len);
95                              off += len;
96                          }
97                          _frameposition = pos / framesize;
98                          return o - off;
99                      }
100             }
[…]
113     };
```

always false

- The variable pos cannot be - at the same time - smaller than bloopend and equal to bloopend.

Class com.sun.jndi.ldap.LdapSchemaParser

- "values.length" (an array's length) will never be less than zero.
- The comment (line 378) suggests the test "values.length == 0".

```
353   final private static Attribute readNextTag(String string, int[] pos)
354         throws NamingException {
355
356         Attribute       attr = null;
357         String          tagName = null;
358         String[]        values = null;
359
[…]
375
376         values = readTag(tagName, string, pos);
377
378         // make sure at least one value was returned
379         if(values.length < 0) {
380             throw new InvalidAttributeValueException("no values for " +
381                                                      "attribute \"" +
382                                                      tagName + "\"");
383         }
```

Class com.sun.org.apache.xalan.internal.xslt.Process

Method void main(String[])

- the variable “useXSLTC” is “true” and will always be “true”

```
boolean useXSLTC = true;
for (int i = 0; i < argv.length; i++)
{
    if ("-XSLTC".equalsIgnoreCase(argv[i]))
    {
      useXSLTC = true;
    }
}
```

this changes nothing…

Class javax.imageio.ImageTypeSpecifier$Indexed

Method void <init> (byte[] , byte[] , byte[] , byte[] , int, int)

- useless test

```
if (bits != 1 && bits != 2 && bits != 4 &&
    bits != 8 && bits != 16) {
    throw new IllegalArgumentException("Bad value for bits!");
}
if (dataType != DataBuffer.TYPE_BYTE &&
    dataType != DataBuffer.TYPE_SHORT &&
    dataType != DataBuffer.TYPE_USHORT &&
    dataType != DataBuffer.TYPE_INT) {
    throw new IllegalArgumentException
        ("Bad value for dataType!");
}
if ((bits > 8 && dataType == DataBuffer.TYPE_BYTE) ||
    (bits > 16 && dataType != DataBuffer.TYPE_INT)) {
    throw new IllegalArgumentException
        ("Too many bits for dataType!");
}
```

bits ∈ {1,2,4,8,16};
i.e., **bits** is never larger than 16

Class sun.tracing.MultiplexProviderFactory

- The method was probably never executed with assertions “on”. “Class.forName” will always fail.

```
117     public void uncheckedTrigger(Object[] args) {
118         for (Probe p : probes) {
119             try {
120                 // try the fast path
121                 ProbeSkeleton ps = (ProbeSkeleton)p;
122                 ps.uncheckedTrigger(args);
123             } catch (ClassCastException e) {
124                 // Probe.trigger takes an "Object ..." varargs parameter,
125                 // so we can't call it directly.
126                 try {
127                     Method m = Probe.class.getMethod(
128                         "trigger", Class.forName("[java.lang.Object"));
129                     m.invoke(p, args);
130                 } catch (Exception e1) {
131                     assert false; // This shouldn't happen
132                 }
133             }
134         }
135     }
```

Illegal String

Class java.util.JapaneseImperialCalendar

- hopefully, the *“transition era”* will never be -1

```
2190    private int actualMonthLength() {
2191        int length = jcal.getMonthLength(jdate);
2192        int eraIndex = getTransitionEraIndex(jdate);
2193        if (eraIndex == -1) {
2194            long transitionFixedDate = sinceFixedDates[eraIndex];
2195            CalendarDate d = eras[eraIndex].getSinceDate();
2196            if (transitionFixedDate <= cachedFixedDate) {
2197                length -= d.getDayOfMonth() - 1;
2198            } else {
2199                length = d.getDayOfMonth() - 1;
2200            }
2201        }
2202        return length;
2203    }
```

Class com.sun.corba.se.impl.naming.pcosnaming.NamingContextImpl
Method public static String nameToString(NameComponent[] name)

- The Developer wanted to use **&&** - currently, **name.length** is called, when **name** is **null**

```
804     {
805         StringBuffer s = new StringBuffer("{");
806         if (name != null || name.length > 0) {
807             for (int i=0;i<name.length;i++) {
808                 if (i>0)
809                     s.append(",");
810                 s.append("[").
811                         append(name[i].id).
812                         append(",").
813                         append(name[i].kind).
814                         append("]");
815             }
816         }
817         s.append("}");
818         return s.toString();
819     }
```

Class com.sun.java.swing.plaf.windows.WindowsLookAndFeel

- `c` will be `null`

```
2069    static void repaintRootPane(Component c) {
2070        JRootPane root = null;
2071        for (; c != null; c = c.getParent()) {
2072            if (c instanceof JRootPane) {
2073                root = (JRootPane)c;
2074            }
2075        }
2076
2077        if (root != null) {
2078            root.repaint();
2079        } else {
2080            c.repaint();
2081        }
2082    }
```

Class java.nio.file.FileTreeWalker

Method next()

- `ioe` will be `null`

```
361         if (ioe != null) {
362             ioe = e;
363         } else {
364             ioe.addSuppressed(e);
365         }
```

Class java.awt.image.DataBufferUShort

- if `dataArray` is `null`, we will never reach this point; the `dataArray.length` call will lead to a `NullPointException`!

```
166     public DataBufferUShort(short dataArray[][], int size) {
167         super(UNTRACKABLE, TYPE_USHORT, size, dataArray.length);
168         if (dataArray == null) {
169             throw new NullPointerException("dataArray is null");
170         }
171         for (int i=0; i < dataArray.length; i++) {
172             if (dataArray[i] == null) {
173                 throw new NullPointerException("dataArray["+i+"] is null");
174             }
175         }
176
177         bankdata = (short[][]) dataArray.clone();
178         data = bankdata[0];
179     }
```

Class java.awt.image.DataBufferUShort

- if `dst` is `null`, we will never reach this point; the `dst.getNumBands()` call will lead to a `NullPointException` that will abort the evaluation of the method.

```
253     public final WritableRaster filter (Raster src, WritableRaster dst) {
254         int numBands  = src.getNumBands();
255         int dstLength = dst.getNumBands();
256         int height    = src.getHeight();
257         int width     = src.getWidth();
258         int srcPix[]  = new int[numBands];
259
260         // Create a new destination Raster, if needed
261
262         if (dst == null) {
263             dst = createCompatibleDestRaster(src);
264         }
```

Class com.sun.media.sound.SoftPerformer

Method SoftPerformer(ModelPerformer performer)

Level 1 Issue

```
746   if (isUnnecessaryTransform(conn.getDestination().getTransform())) {
747       conn.getDestination().setTransform(null);
748   }
…
752   if (isUnnecessaryTransform(src.getTransform())) {
753       src.setTransform(null);
754   }
```

```
761   private static boolean isUnnecessaryTransform(ModelTransform transform) {
762       if (transform == null)
763           return false;
764       if (!(transform instanceof ModelStandardTransform))
765           return false;
766       ModelStandardTransform stransform = (ModelStandardTransform)transform;
767       if (stransform.getDirection() != ModelStandardTransform.DIRECTION_MIN2MAX)
768           return false;
769       if (stransform.getPolarity() != ModelStandardTransform.POLARITY_UNIPOLAR)
770           return false;
771       if (stransform.getTransform() != ModelStandardTransform.TRANSFORM_LINEAR)
772           return false;
773       return false;
774   }
```

- Here, the bug is in the method `isUnnecessaryTransform` which always returns `false`.
- (Most likely the method `isUnnecessaryTransform was not tested at all!`)

Class javax.swing.text.GlyphView

Method javax.swing.text.GlyphView$JustificationInfo getJustificationInfo(int)

```
for (int i = txtEnd, state = TRAILING; i >= txtOffset; i--) {
        if (' ' == segment.array[i]) {
            spaceMap.set(i - txtOffset);
            if (state == TRAILING) {
                trailingSpaces++;
            } else if (state == CONTENT) {
                state = SPACES;
                leadingSpaces = 1;
            } else if (state == SPACES) {
                leadingSpaces++;
            }
        } else if ('\t' == segment.array[i]) {
            hasTab = true;
            break;
        } else {
            if (state == TRAILING) {
                if ('\n' != segment.array[i]
                        && '\r' != segment.array[i]) {
                    state = CONTENT;
                    endContentPosition = i;
                }
            } else if (state == CONTENT) {
                //do nothing
            } else if (state == SPACES) {
                contentSpaces += leadingSpaces;
                leadingSpaces = 0;
            }
            startContentPosition = i;
        }
    }
```

always true
(defensive code?)

- The last comparison of state with “SPACES” is superfluous/will always be true, because state is either TRAILING (at the very beginning) or CONTENT or SPACES and the first two cases are already covered by the preceding if statements.

Class java.util.concurrent.ConcurrentHashMap

```
2322    private final void tryPresize(int size) {
2323        int c = (size >= (MAXIMUM_CAPACITY >>> 1)) ? MAXIMUM_CAPACITY :
2324            tableSizeFor(size + (size >>> 1) + 1);
2325        int sc;
2326        while ((sc = sizeCtl) >= 0) {
2327            Node<K,V>[] tab = table; int n;
2328            if (tab == null || (n = tab.length) == 0) {
2329                n = (sc > c) ? sc : c;
2330                if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
2331                    try {
2332                        if (table == tab) {
2333                            @SuppressWarnings("unchecked")
2334                            Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
2335                            table = nt;
2336                            sc = n - (n >>> 2);
2337                        }
2338                    } finally {
2339                        sizeCtl = sc;
2340                    }
2341                }
2342            }
2343            else if (c <= sc || n >= MAXIMUM_CAPACITY)
2344                break;
2345            else if (tab == table) {
2346                int rs = resizeStamp(n);
2347                if (sc < 0) {
2348                    Node<K,V>[] nt;
2349                    if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
2350                        sc == rs + MAX_RESIZERS || (nt = nextTable) == null ||
2351                        transferIndex <= 0)
2352                        break;
2353                    if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
2354                        transfer(tab, nt);
2355                }
2356                else if (U.compareAndSwapInt(thi
2357                                             (rs
2358                    transfer(tab, null);
2359            }
2360        }
2361    }
```

always false

- The loop condition always ensures that sc is larger than “0”.

> […]So thanks for pointing out a clear case of bad code explanation! It will get fixed; most likely by commenting out the code block to prevent confusion.
>
> -Doug [Doug Lea; author and lead developer of Java’s concurrency API]

Class java.time.Duration *(, LocalDate (introduced with Java 8))*

Method minusDays *(, minusWeeks (introduced with Java 8))*

Level 2 Issue

Copy&Paste

- If daysToSubtract is Long.MIN_VALUE the method will always throw an ArithmeticException (as documented).
- However, the code is overly complex.
  “return plusDays(-daysToSubtract)” would do the job.

```
public Duration minusDays(long daysToSubtract) {
     return (daysToSubtract == Long.MIN_VALUE ? plusDays(Long.MAX_VALUE).plusDays(1) : plusDays(-daysToSubtract));
}
```

```
public Duration plusDays(long daysToAdd) {
      return plus(Math.multiplyExact(daysToAdd, SECONDS_PER_DAY), 0);
}
```

```
// @throws ArithmeticException if the result overflows a long
public static long multiplyExact(long x, long y) {…}
```

Issues than can be found using Static Analyses

- Data- and Control Flow Dependent Issues
  - (Complex) Useless Computations
  - Deadlocks/Race Conditions
  - Security Vulnerabilities
  - Privacy Issues
- Unused Methods/Fields
- …

Key Terminology

(A BRIEF OVERVIEW)

True and False Positives

True and False Negatives

True and **False** Positives

This is what static analyses should detect.

- a **True** Positive is the correct finding *(of something relevant)*

- a **False** Positive is a finding that is just incorrect

False positives are typically caused by the weaknesses of the analysis.

Do you see the **code smell**?

(W.R.T. THE INCONSISTENT USE OF A LOCAL VARIABLE.)

```
void printIt(String args[]) {
  if (args != null) {
    System.out.println("number of elements: " + args.length);
  }
  for (String arg : args) {
    System.out.println(arg);
  }
}
```

Example of a **True** Positive

We can detect such code smells using a *basic analysis* that detects object accesses (`o.xyz`) that appear in a guarded context (`if (o != null)`) and also outside of an explicitly guarded context.

```
void printIt(String args[]) {
  if (args != null) {
    System.out.println("number of elements: " + args.length);
  }
  for (String arg : args) {
    System.out.println(arg);
  }
}
```

guard

guarded & unguarded access

Do you see a **code smell**?

(W.R.T. THE INCONSISTENT USE OF A LOCAL VARIABLE.)

```
void printReverse(String args[]) {
  int argscount = 0;
  if (args != null) {
    argscount = args.length;
  }
  for (int i = argscount - 1; i >= 0; i--) {
    System.out.println(args[i]);
  }
}
```

Example of a **False** Positive

If we use our *basic analysis* to detect object accesses that appear in a guarded context and also outside of an explicitly guarded context we may have false positives in case of *complex* conditions.

```
void printReverse(String args[]) {
  int argscount = 0;
  if (args != null) {
    argscount = args.length;
  }
  for (int i = argscount - 1; i >= 0; i--) {
    System.out.println(args[i]);
  }
}
```

guard

guarded & unguarded access

implicit guard

Example of a **False** Positive

What happens if we use our *basic analysis* to detect object accesses that appear in a guarded context and also outside of an explicitly guarded context.

Here, to avoid **false** positives you need to model the dependencies between (all relevant) different values.

```
void printReverse(String args[]) {
  int argscount = 0;
  if (args != null) {
    argscount = args.length;
  }
  for (int i = argscount - 1; i >= 0; i--) {
    System.out.println(args[i]);
  }
}
```

guarded & unguarded access

implicit guard

True and **False** Negatives

- a **True** Negative is the correct finding of no issue.

- a **False** Negative is an issue that is not reported.

Generally, only explicitly considered in the context of formal approaches.

Example of a **False** Negative

Let's assume that we have **an analysis that detects potential null-pointer exceptions.**
(But, it is only a simple intra-procedural analysis.)

```
private Object f() { return null; }

void printIt(String args[]) {
  Object o = f();
  if (args != o) {
    System.out.println("number of elements: " + args.length);
  }
  for (String arg : args) {
    System.out.println(arg);
  }
}
```

Irrelevant True Positives

- Irrelevancy is context-dependent…
  - Issues related to Serialization are irrelevant when your application doesn't use Serialization at all.
  - A violation of the `hashCode-equals` contract may be completely irrelevant for an (inner) class if the instances of it are never put in a collection that uses hashes.
  - …

Irrelevant True Positives

```
// the only caller:
isSmallEnough("XYZ")
```

```
private boolean isSmallEnough(Object i) {
    assert(i != null);
    Object o = ""+i;
    o.length < 10
}
```

They are typically related to:

- default cases in switch statements
- assertions
- a test that leads to an AssertionError

Complex True Positives

```
GeneralPath result = new GeneralPath(GeneralPath.WIND_NON_ZERO);
…
if (dx != 0 || dy != 0) {
  AffineTransform tx = AffineTransform.getTranslateInstance(dx, dy);
  result = (GeneralPath)tx.createTransformedShape(result);
}
```

This cast will always fail!

Complex True Positives

```
GeneralPath result = new GeneralPath(GeneralPath.WIND_NON_ZERO);
…
if (dx != 0 || dy != 0) {
  AffineTransform tx = AffineTransform.getTranslateInstance(dx, dy);
  result = (GeneralPath)tx.createTransformedShape(result);
}
```

This cast will always fail!

```
public Shape createTransformedShape(Shape pSrc) {
  if (pSrc == null) {
    return null;
  }
  return new Path2D.Double(pSrc, this);
}
```

interface Shape

class Path2D implements Shape, Cloneable

/*inner*/ class Double extends Path2D implements Serializable

Complex True Positives

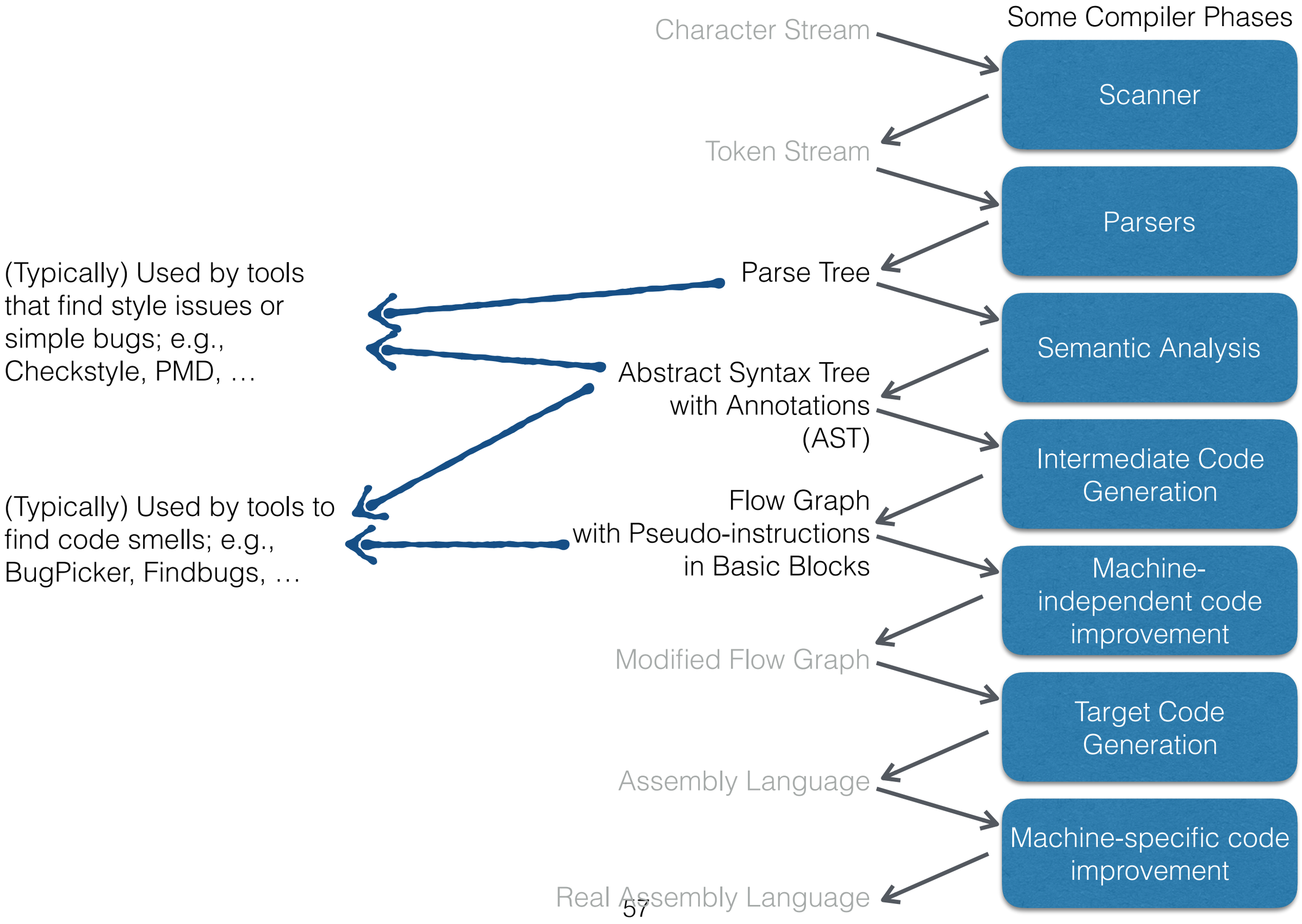
Are easily falsely perceived as false positives

> […]the general trend holds; a not-understood bug report is commonly labeled a false positive, rather than spurring the programmer to delve deeper. The result? We have completely abandoned some analyses that might generate difficult to understand reports.
>
> Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53, 2 (February 2010), 66-75. DOI=http://dx.doi.org/10.1145/1646353.1646374

```
GeneralPath result = new GeneralPath(GeneralPath.WIND_NON_ZERO);
…
if (dx !
  Affin
  resul
}

publ
  if

  }
  re
}
```

```
class Path2D implements Shape, Cloneable

/*inner*/ class Double extends Path2D implements Serializable
```

Compilers and Static Analyses



Simple Example

Input/Given:

i = j +1;

TOKENS

Identifier(i) WS Identifier(j) WS Operator(+) WS Constant(1) SEMICOLON

AST with Annotations:

```
AssignmentStatement(
    target =      Var(name=i,type=Int),
    expression =  AddExpression(type=Int,
                      left =   Var(name=j,type=Int),
                      right = Const(1)
                  )
)
```

Intermediate Representations Generally Facilitate Static Analyses

- Nested Control-flow and complex expressions are unraveled. Intermediate values are given explicit names. The instruction set is usually limited.
- Examples
  - Java Bytecode
    (Stack Based)
  - 3Address Code
  - Static Single Assignment (Form)
  - Common Intermediate Language (CIL)
    (Stack Based)
  - LLVM IR
  - …

Java Bytecode

A HARDWARE- AND OPERATING SYSTEM-INDEPENDENT BINARY FORMAT, KNOWN AS THE CLASS FILE FORMAT.

The Java® Virtual Machine Specification
Java SE 8 Edition
Specification: JSR-000337 Java® SE 8 Release Contents Specification ("Specification") Version: 8
Status: Proposed Final Draft
Release: January 2014

Structure of the Java Virtual Machine (JVM)

- Data types:
  - Primitive Types:
    `boolean`, `byte`, `short`, `int`, `char`, *(computational type int)*
    `long`,
    `float`,
    `double`,
    `return address`
  - Reference Types:
    `class`,
    `array`,
    `interface` types

Structure of the Java Virtual Machine (JVM)

- Run-time Data Areas
  - the `pc` (program counter) register contains the address of the instruction that is currently executed by a thread; each thread has its own pc
  - each JVM thread has a `private stack` which holds local variables and partial results
  - the `heap` which is shared among all threads
  - `frames` are allocated from a JVM thread's private stack when a method is invoked; each frame has its own array of `local variables` and `operand stack`
    - local variables are indexed
      - *a single local* variable can hold a value of type `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference`, or `return address` (computational type category 1)
      - *a pair of local variables* can hold a value of type `long` or `double` (computational type category 2)
    - the `operand stack` is empty at creation time; an entry can hold any value
    - the `local variables` contains the parameters (including the implicit `this` parameter in local variable 0)

Structure of the Java Virtual Machine (JVM)

- Special Methods
  - the name of instance initialization methods (Java constructors) is "`<init>`"
  - the name of the class or interface initialization method (Java static initializer) is "`<clinit>`"

- Exceptions are instance of the class `Throwable` or one of its subclasses; exceptions are thrown if
  - an `athrow` instruction was executed
  - an abnormal execution condition

Structure of the Java Virtual Machine (JVM)

- Instruction Set Summary
  - an instruction consists of a one-byte opcode specifying the operation and zero or more operands (arguments to the operation)
  - most instructions encode type information in their name; in particular those operating on primitive types (e.g., `iadd`, `fadd`, `dadd`)
  - some are generic and are only restricted by the computational type category of the values (e.g. `swap`, `dup2`)

Structure of the Java Virtual Machine (JVM)

- Categories of Instructions
  - Load and store instructions (e.g., `aload_0`, `istore`(x))
    (except of the load and store instructions the only other instruction that manipulates a local variable is `iinc`)
  - Arithmetic instructions (e.g., `add`, `ushr`)
  - Type conversion instructions (e.g., `i2d`)
  - Object/Array creation and manipulation (e.g., `new`, `checkcast`)
  - (Generic) Operand Stack Management Instructions (e.g., `dup`)
  - Control Transfer Instructions (e.g., `itlt`, `if_icmplt, goto, jsr, ret`)
  - Method Invocation and Return instructions (e.g., `invokespecial`, `return`)
  - Throwing Exceptions (`athrow`)
  - Synchronization (`monitorenter`, `monitorexit`)

Java Bytecode

```
static int max(int i, int j) {
    if (i > j)
        return i;
    else
        return j;
}
```

PC	Instruction
0	iload_0
1	iload_1
2	if_icmple 7
5	iload_0
6	ireturn
7	iload_1
8	ireturn

Java Bytecode

```
static int factorial(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

PC	Instruction
0	iconst_1
1	istore_1
2	goto 12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc 0, -1
12	iload_0
13	ifgt 5
16	iload_1
17	ireturn

Java Bytecode

```
static int numberOfDigits(int i) {
     assert (i > 0);
     return ((int) Math.floor(Math.log10(i))) + 1;
 }
```

PC	Instruction	Code
0	getstatic TACDemo { boolean $assertionsDisabled }	assert(i > 0)
3	ifne 18	
6	iload_0	
7	ifgt 18	
10	new java.lang.AssertionError	
13	dup	
14	invokespecial java.lang.AssertionError { void <init> () }	
17	athrow	
18	iload_0	`Math.log10(i)`
19	i2d	
20	invokestatic java.lang.Math { double log10 (double) }	
23	invokestatic java.lang.Math { double floor (double) }	Math.floor(…)
26	d2i	(int) *“Typecast”*
27	iconst_1	1
28	iadd	+
29	ireturn	return

Java Bytecode

```
public class TACDemo {

    private static volatile TACDemo instance;

    static TACDemo getInstance() {
        TACDemo instance = TACDemo.instance;
        // thread-safe double checked locking
        if (instance == null) {
            synchronized (TACDemo.class) {
                instance = TACDemo.instance;
                if (instance == null) {
                    instance = new TACDemo();
                    TACDemo.instance = instance;
                }
            }
        }
        return instance;
    }
}
```

PC	Instruction	Exceptions	
0	getstatic TACDemo { TACDemo instance }		
3	astore_0		
4	aload_0		
5	ifnonnull 41		
8	ldc TACDemo.class		
10	dup		
11	astore_1		
12	monitorenter		
13	getstatic TACDemo { TACDemo instance }	1: Any	
16	astore_0		
17	aload_0		
18	ifnonnull 33		
21	new TACDemo		
24	dup		
25	invokespecial TACDemo { void <init> () }		
28	astore_0		
29	aload_0		
30	putstatic TACDemo { TACDemo instance }		
33	aload_1		
34	monitorexit		
35	goto 41		
38	aload_1		2: Any
39	monitorexit		
40	athrow		
41	aload_0		
42	areturn		

Analyzing Java Bytecode works (very) well, because **Java compilers deliberately don't optimize.**

```
static long optimizableExpression(double d, int i, long l) {
    return (long)((d * d) + i) * 0l;
}
```

PC	Instruction
0	dload_0
1	dload_0
2	dmul
3	iload_2
4	i2d
5	dadd
6	d2l
7	lconst_0
8	lmul
9	lreturn

Example

Analyzing Java Bytecode works (very) well, because **Java compilers deliberately don't optimize.**

```
static void optimizableIndexInc(int[] is, int i) {
    is[i++] = 0;
    is[i++] = 1;
}
```

PC	Instruction
0	aload_0
1	iload_1
2	iinc 11
5	iconst_0
6	iastore
7	aload_0
8	iload_1
9	iinc 11
12	iconst_1
13	iastore
14	return

Example

Three-Address Code

Three-Address Code

- Three-address code is a sequence of statements with the general form:
  `x = y op z`

- where `x`,`y` and `z` are (local variable) names, constants (in case of `y` and `z`) or compiler-generated temporaries

- The name was chosen, because “most” statements use three addresses: two for the *operators* and *one to store the result*

General Types of Three-Address Statements

- **Assignment** statements `x = y bin_op z` or `x = unary_op z`
- **Copy** statements `x = y`
- **Unconditional jumps**: `goto l` (and `jsr l`, `ret` in case of Java bytecode)
- **Conditional jumps**: `if (x rel_op y) goto l` (else fall through), `switch`
- **Method call and return**: `invoke(m, params)`, `return x`
- **Array access**: `a[i]` or `a[i] = x`
- *More IR specific types.*

Converting Java Bytecode to Three-Address Code

- Core Idea:
  - Compute for each instruction the current stack layout; i.e., the types of values found on the stack before the instruction is evaluated; do this by following the control flow
    (The JVM specification guarantees that the operand stack always has the same layout independent of the taken path.)
  - Assign each local variable to a variable where the name is based on the local variable index. E.g., if the stack is empty then an `aload0` instruction is transformed into the three address code:
    `op_0 = r_0`

Converting Java Bytecode to Three-Address Code

```
static int numberOfDigits(int i) {
    return ((int) Math.floor(Math.log10(i))) + 1;
}
```

PC	Instruction	Stack Layout (before execution)	Three-Address Code
	initialization		`r_0 = i;`
0	iload_0		`op_0 = r_0;`
1	i2d	0: Integer Value	`op_0 = (double) op_0;`
2	invokestatic java.lang.Math { double log10 (double) }	0: Double Value	`op_0 = Math.log10(op_0);`
5	invokestatic java.lang.Math { double floor (double) }	0: Double Value	`op_0 = Math.floor(op_0);`
8	d2i	0: Double Value	`op_0 = (int) op_0;`
9	iconst_1	0: Integer Value	`op_1 = 1;`
10	iadd	1: Integer Value 0: Integer Value	`op_0 = op_0 + op_1;`
11	ireturn	0: Integer Value	`return op_0;`

A Three-Address Code for Java Bytecode

Basic Statements

```
Assignment(
        pc:             PC,
        targetVar:      Var,
        expr:           Expr
)

Goto(pc: PC, target: Int)

JumpToSubroutine(pc: PC, target: Int)
Ret(pc: PC, returnAddressVar: Var)

Nop(pc: PC)

If(
        pc:             PC,
        left:           ValExpr,
        condition:      RelationalOperator,
        right:          ValExpr,
        target:         Int
)
Switch(
        pc:             PC,
        defaultTarget:  PC,
        index:          ValExpr,
        npairs:         IndexedSeq[(Int, PC)]
)
```

```
ReturnValue(pc: PC, expr: ValExpr)
Return(pc: PC)

ArrayStore(
    pc:         PC,
    arrayRef:   Var,
    index:      ValExpr,
    value:      ValExpr
)

Throw(pc: PC, exception: Var)

MonitorEnter(pc: PC, objRef: Var)
MonitorExit(pc: PC, objRef: Var)
```

A Three-Address Code for Java Bytecode

Field Access and Method Call Statements

```
PutStatic(
    pc:             PC,
    declaringClass: ObjectType,
    name:           String,
    value:          ValExpr
)
PutField(
    pc:             PC,
    declaringClass: ObjectType,
    name:           String,
    objRef:         Var,
    value:          ValExpr
)
```

```
NonVirtualMethodCall(
    pc:             PC,
    declaringClass: ReferenceType,
    name:           String,
    descriptor:     MethodDescriptor,
    receiver:       Var,
    params:         List[ValExpr]
)
VirtualMethodCall(
    pc:             PC,
    declaringClass: ReferenceType,
    name:           String,
    descriptor:     MethodDescriptor,
    receiver:       Var,
    params:         List[ValExpr]
)
StaticMethodCall(
    pc:             PC,
    declaringClass: ReferenceType,
    name:           String,
    descriptor:     MethodDescriptor,
    params:         List[ValExpr]
)
```

A Three-Address Code for Java Bytecode

Expressions

```
InstanceOf(pc: PC, value: Var, cmpTpe:
ReferenceType)
Checkcast(pc: PC, value: Var, cmpTpe:
ReferenceType)

Compare(
        pc:         PC,
        left:       ValExpr,
        condition:  RelationalOperator,
        right:      ValExpr
)

BinaryExpr(
    pc:   PC,
    cTpe: ComputationalType,
    op:   BinaryArithmeticOperator,
    left: ValExpr, right: ValExpr
)

PrefixExpr(
    pc:       PC,
    cTpe:     ComputationalType,
    op:       UnaryArithmeticOperator,
    operand:  ValExpr
)

PrimitiveTypecastExpr(
        pc:         PC,
        targetTpe:  BaseType,
        operand:    ValExpr
)

New(pc: PC, tpe: ObjectType)

NewArray(
        pc:     PC,
        counts: List[ValExpr],
        tpe:    ArrayType
)
ArrayLoad(pc: PC, index: Var, arrayRef: Var)
ArrayLength(pc: PC, arrayRef: Var)
```

ValExpr

```
Param(cTpe: ComputationalType, name: String)
SimpleVar(id: Int, cTpe: ComputationalType)

IntConst(pc: PC, value: Int)
LongConst(pc: PC, value: Long)
FloatConst(pc: PC, value: Float)
DoubleConst(pc: PC, value: Double)
StringConst(pc: PC, value: String)
ClassConst(pc: PC, value: ReferenceType)

NullExpr(pc: PC)
```

A Three-Address Code for Java Bytecode

Expressions

```
GetField(
        pc:                 PC,
        declaringClass:     ObjectType,
        name:               String,
        objRef:             ValExpr
)
GetStatic(
        pc:                 PC,
        declaringClass:     ObjectType,
        name:               String
)

Invokedynamic(
        pc:                 PC,
        bootstrapMethod:    BootstrapMethod,
        name:               String,
        descriptor:         MethodDescriptor,
        params:             List[Expr]
)
MethodTypeConst(pc: PC, desc: MethodDescriptor)
MethodHandleConst(pc: PC, desc: MethodHandle)
```

```
NonVirtualFunctionCall(
    pc:                 PC,
    declaringClass:     ReferenceType,
    name:               String,
    descriptor:         MethodDescriptor,
    receiver:           Expr,
    params:             List[Expr]
)

VirtualFunctionCall(
    pc:                 PC,
    declaringClass:     ReferenceType,
    name:               String,
    descriptor:         MethodDescriptor,
    receiver:           Expr,
    params:             List[Expr]
)

StaticFunctionCall(
    pc:                 PC,
    declaringClass:     ReferenceType,
    name:               String,
    descriptor:         MethodDescriptor,
    params:             List[Expr]
)
```

Java Bytecode vs. Three-Address Code

Java

```
static int max(int i, int j) {
    if (i > j)
        return i;
    else
        return j;
}
```

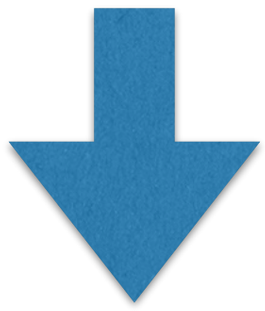
Bytecode

PC	Instruction
0	iload_0
1	iload_1
2	if_icmple 7
5	iload_0
6	ireturn
7	iload_1
8	ireturn

Three Address Code

```
0: r_0 = i;
1: r_1 = j;
2: op_0 = r_0;
3: op_1 = r_1;
4: if(op_0 <= op_1) goto 7;
5: op_0 = r_0;
6: return op_0;
7: op_0 = r_1;
8: return op_0;
```

After Peephole Optimizations



```
0: if(i <= j) goto 2;
1: return i;
2: return j;
```

Java Bytecode vs. Three-Address Code

Java

```
static int factorial(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

Bytecode

PC	Instruction
0	iconst_1
1	istore_1
2	goto 12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc 0, -1
12	iload_0
13	ifgt 5
16	iload_1
17	ireturn

Three Address Code

```
 0: r_0 = n;
 1: op_0 = 1;
 2: r_1 = op_0;
 3: goto 9;
 4: op_0 = r_1;
 5: op_1 = r_0;
 6: op_0 = op_0 * op_1;
 7: r_1 = op_0;
 8: r_0 = r_0 + -1;
 9: op_0 = r_0;
10: if(op_0 > 0) goto 4;
11: op_0 = r_1;
12: return op_0;
```

After Peephole Optimizations

```
0: r_0 = n;
1: r_1 = 1;
2: goto 5;
3: r_1 = r_1 * r_0;
4: r_0 = r_0 + -1;
5: if(r_0 > 0) goto 3;
6: return r_1;
```

Control-Flow Graph

- The control-flow graph (CFG) represents the control flow of a single method.

- Each node represents a basic block. A basic block is a maximal-length sequence of statements without jumps in and out (and no exceptions).

- The arcs represent the inter-node control flow.

Control-Flow Graph

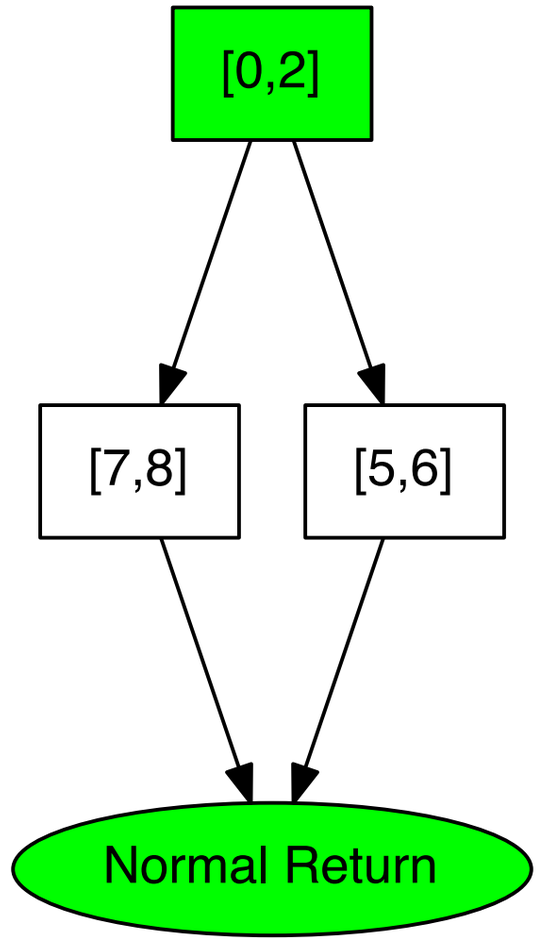
Java

```
static int max(int i, int j) {
    if (i > j)
        return i;
    else
        return j;
}
```

Bytecode

PC	Instruction
0	iload_0
1	iload_1
2	if_icmple 7
5	iload_0
6	ireturn
7	iload_1
8	ireturn

CFG



Java Bytecode vs. Three-Address Code

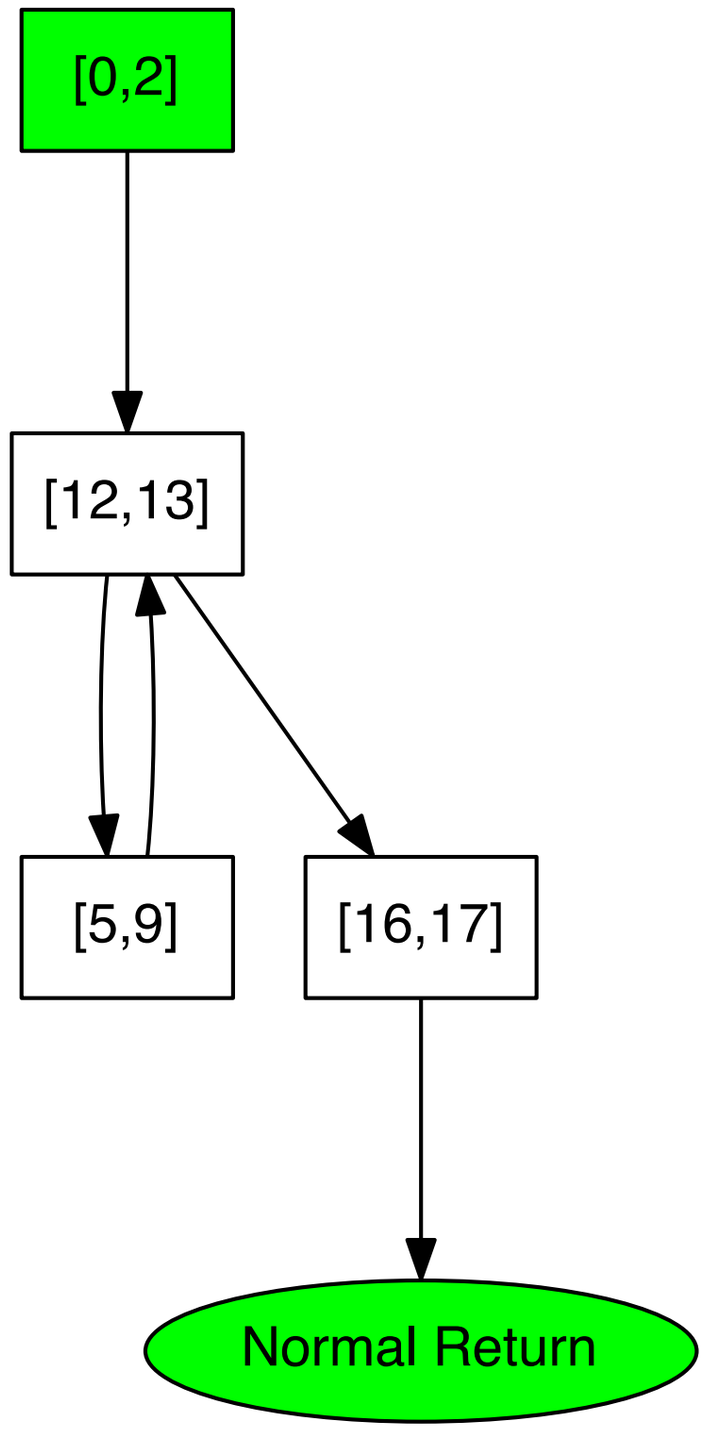
Java

```
static int factorial(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

Bytecode

PC	Instruction
0	iconst_1
1	istore_1
2	goto 12
5	iload_1
6	iload_0
7	imul
8	istore_1
9	iinc 0, -1
12	iload_0
13	ifgt 5
16	iload_1
17	ireturn

CFG



Java Bytecode

```
static int baseNumberOfDigits(int i) {
     return ((int) Math.floor(Math.log10(i))) + 1;
 }
```

PC	Instruction
0	iload_0
1	i2d
2	invokestatic java.lang.Math { double log10 (double) }
5	invokestatic java.lang.Math { double floor (double) }
8	d2i
9	iconst_1
10	iadd
11	ireturn

CFG