Detecting non-local violations of API contracts in large software systems
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}
One month later...
“Feature X is not working”
“Could you have a look at the log file, please?”
“The log file is empty!”
“Customer says, the log is empty...”

“Oh no! How come?”
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}

static void copyAll(Reader r, Writer w) {

    int buffer = 0;
    try {
        do {
            buffer = r.read();
            w.write(buffer);
        } while (buffer != -1);
    }
}
One hour later...
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}
java.io

Class PrintWriter

java.lang.Object
   java.io.Writer
      java.io.PrintWriter

public class PrintWriter
   extends Writer

Print formatted representations of objects to a text-output stream. This class implements all of the print methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the PrintStream class, if automatic flushing is enabled it will be done only when one of the println() methods is invoked, rather than whenever a newline character happens to be output. The println() methods use the platform's own notion of line separator rather than the newline character.

**Methods in this class never throw I/O exceptions. The client may inquire as to whether any errors have occurred by invoking checkError().**

Since:
   JDK1.1
A better world...
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }

}
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }

}
Detecting non-local violations of API contracts in large software systems
Non-local violations involve aliasing through long call chains
API contracts:
Client contracts that come along with an API.
Large software systems: Complete Java programs with up to hundreds of thousands lines of code
Pure runtime monitoring

How does it work today?
public class Logging {
    private static Writer w;
    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }
    public static void log(String s) {
        w.write(s);
    }
    public static void logAll(Reader r) {
        StreamUtil.copyAll(r,w);
    }
}

“no write after close”
How to specify API contracts?
Current approach: **Tracematches**

```java
tracematch(Writer w) {
    sym close after returning:
    call(* Writer.close()) && target(w);
    sym write before:
    call(* Writer.write(...) && target(w);

close write
{
    System.out.println(
        "Writer " +w+" was closed!";
    }
}
```
```java
{  
    w1.close();
    System.out.println(
        w1.write("""foo"""");
        "Writer +w+ was 'closed!'"));
}

w1.write("foo");
```
Problem 1:
Potentially large runtime overhead
Problem 2: No static guarantees
public class Logging {
    private static Writer w;
    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }
    public static void log(String s) {
        w.write(s);
    }
    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}

“no write after close”

4 novel static program analyses
First static analysis:

“Quick Check”
△ close

△ write

\[
\begin{align*}
(\triangleleft) &= 0 \\
(\triangleright) &= 2
\end{align*}
\]
Second static analysis: “Consistent-shadows analysis”
[close, write]
[close, write]

\{c_1\} \times \{w_1, w_2\}

```java
new PrintWriter();
```
Third static analysis:

“Flow-sensitive unnecessary shadows analysis”
true

true

\( w = \text{i(false)} \)
\( w \neq \text{i(true)} \)
\( w = \text{i(w1)} \)

\text{v}

\text{w1.close();} \quad // s1

\text{w1.write("foo");} \quad // s2

\text{w1.write("foo");} \quad // s3
4th static analysis: “run-once loop optimization”
What if you had a universal remote... that controlled your universe?

ADAM SANDLER

CLICK
Results

102 program/tracematch combinations static guarantees in 77 cases
Found 5 programs (out of 12) with bugs or questionable code.
less than 10% overhead in 9 cases
private final void FillBuff() {
    ...
    try {
        if (i = inputStream.read(...) == -1) {
            inputStream.close();
            throw new java.io.IOException();
        } else {
            maxNextCharInd += i;
            return;
        }
    } catch (...) {
        ...
    }
}
static String getLine(BufferedReader reader, int line) {
    if (reader == null)
        return "";
    try {
        String text=null;
        for(int i=0; i < line; i++) {
            text = reader.readLine();
        }
        return text;
    } catch (IOException ioe) {
        return null;
    }
}
public Iterator iterator() {
    return new Iterator() {
        Iterator i = list.iterator();
        public void remove() {
            throw new UnsupportedOperationException();
        }
        public boolean hasNext() {
            return i.hasNext();
        }
        public Object next() {
            return i.next();
        }
    };
}
private List markUsages(IDataFlowNode inode) {
    ...
    for (Iterator k = ((List)entry.getValue()).iterator(); k.hasNext()) {
        addAccess(k, inode);
    }
    ...
}

private void addAccess(Iterator k, IDataFlowNode inode) {
    NameOccurrence occurrence = (NameOccurrence) k.next();
    ...
}
private List markUsages(IDataFlowNode inode) {
    ...
    for (NameOccurrence occurrence : entry.getValue()) {
        addAccess(occurrence, inode);
    }
    ...
}

private void addAccess(NameOccurrence occurrence, IDataFlowNode inode) {
    ...
}
protected NameDeclaration
    findVariableHere(NameOccurrence occurrence) {
        if (occurrence.isThisOrSuper() ||
            occurrence.getImage().equals(className)) {
            if (variableNames.isEmpty() &&
                methodNames.isEmpty()) {
                return null;
            }
            if (!variableNames.isEmpty()) {
                return variableNames.keySet().iterator().next();
            }
        }
        if (!variableNames.isEmpty()) {
            return variableNames.keySet().iterator().next();
        }
        return methodNames.keySet().iterator().next();
    }
...