



Home News & Analysis EE Life Design Products Education & Training Events Sign In Join

#### Design

DesignLines

Audio

Automotive

CommsDesign

EDA

Embedded.com

Embedded Internet

Graphical System

Design

Industrial Control

MCU

Medical

Memory

Microwave & RF

Military & Aerospace

Planet Analog

Power Management

Programmable Logic

Signal Processing

Smart Energy

Design Tools

Reference Designs

Evaluation Kit

Source Code



### Guest Editor

#### Learn the Top 5 Java coding mistakes

Alen Zukich

10/2/2009 10:24 AM EDT

While C dominates as the programming language of choice for embedded development, the use of Java is definitely on the rise. In fact, according to a recent VDC survey, 12.3% of respondents currently use Java in the embedded space, and 17.9% expect to be using Java in the next two years.

For those transitioning from embedded development using C, you might find yourself falling into the hype that Java is a "safe" language. For example, Java developers face no requirement for managing memory associated with objects. However, this is where the trap may be laid. Even though there's no need for memory management, developers may need to keep track of specific resources the object allocates. This is especially true in an embedded context where resources are often constrained. Even for experienced developers, these traps pop up time and again and can easily jeopardize your code quality and security.

Here's a round-up of the top five programming issues developers should be aware of in embedded Java development.

##### 1. Resource leaks

Resource leak issues are reported when resources are allocated but not properly disposed after use.

```
1 public void printLines(String fName, Collection lines) {
2 try {
3 File file = new File(fName);
4 PrintWriter pstr = new PrintWriter(
5 new BufferedOutputStream(
6 new FileOutputStream(file)));
7 for (final String line : lines) {
8 pstr.println(line);
9 }
10 } catch (IOException e) {
11 e.printStackTrace();
12 }
13 }
```

Here we can see that 'pstr' is never closed on exit. These types of resource leaks are surprising. Occasionally, these errors are caused by a developer's misunderstanding, but more often than not they're simply overlooked while coding.

##### 2. Null pointer exceptions

A NullPointerException is thrown in the case of an attempt to dereference a null value.

```
15 Reader getReader(String configurationPath) throws
IOException {
16 File file = new File(configurationPath);
17 if (file.exists()) {
18 return new BufferedReader(new FileReader(file));
19 }
20 return null;
21 }
22
23
24 Reader getDefaultReader() throws IOException {
25 return getReader("conf");
26 }
27
28 public void init() throws IOException {
29 load(getDefaultReader());
30 }
31
32 private String load(Reader reader) throws IOException {
33 StringBuffer sb = new StringBuffer();
34
```

### Navigate to related information

**Design:** Java Essentials for Embedded Networked Devices - Part 3: Errors, exceptions and exception handling

**Design:** Expressive vs. permissive languages: Is that the question?

**Discussion:** Deallocating objects vs. deallocating storage

**Technical Paper:** Formal Specifications

**Discussion:** References and const



### Most Popular

- 1 USB Explained: An Introduction to USB and Its Future
- 2 Digital Signal Processing: A Practical Guide (Part 1)
- 3 Fundamentals of Wireless
- 4 The Inefficiency of C++, Fact or Fiction?
- 5 Digital Signal Processing: A Practical Guide (Part 2)
- 6 Digital Signal Processing: A Practical Guide (Part 3)
- 7 Comment: Andy Grove, startups and job creation
- 8 Digital Signal Processing: A Practical Guide (Part 4)
- 9 BLDC motor for Electronic Bike Application
- 10 Digital Signal Processing: A Practical Guide (Part 5)

### Product Parts Search

Enter part number or keyword

SEARCH

### Get Involved

Start a Forum

Subscribe to a Newsletter

**Main Entry:**  
**des·ti·na·tion:**  
 1 : a place  
 worthy of travel  
 or an extended  
 visit

```

35 char[] buffer = new char[1024];
36 int length;
37 while ((length = reader.read(buffer)) > 0) {
38 sb.append(buffer, 0, length);
39 }
40 return sb.toString();
41 }

```

Because the value returned by 'getDefaultReader()' call at line 29 can be null and it's passed into the 'load(Reader reader)' method as a parameter, the value is dereferenced. This case spans multiple methods which gives it an extra level of complexity.

### 3. Null pointer dereference

This one is similar to the previous Java issue, but with a twist. This is a situation of finding statistical null pointer dereferences.

```

22 public void printAbsolutePath() {
23 final File parent = f.getParentFile();
24 if (parent != null) {
25 String absolutePath = parent.getAbsolutePath();
26 System.out.println("absolute path " + absolutePath);
27 }
28 }
29
30 public void printCanonicalParentPath() throws IOException
31 {
31 final File parent = f.getParentFile();
32 if (parent != null) {
33 String canonicalPath = parent.getCanonicalPath();
34 System.out.println("canonical path: " + canonicalPath);
35 }
36 }
37
38 public void printParentPath() throws IOException {
39 String path = f.getParentFile().getPath();
40 System.out.println("path " + path);
41 }

```

In this case, there are a number of instances where you check for null, but for some reason in one (or more) particular cases you don't. So in this example, since the value returned by 'getParentFile()' is usually checked for null before dereferencing, there's a possibility of a null pointer dereference here. These statistical null pointer dereferences don't necessarily mean a real issue is present in your code, but certainly deserves a look.

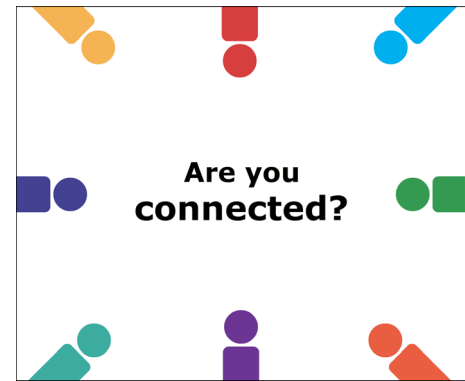
### 4. Use of freed

Use of freed issues are reported when there is an attempt to use resources after they were released.

```

18 public boolean checkMeta(InputStream stream) {
19 byte[] b = new byte[8];
20 try {
21 final long l = stream.available();
22 if (l < 8) {
23 return false;
24 }
25 stream.read(b);
26 } catch (IOException e) {
27 return false;
28 } finally {
29 try {
30 stream.close();
31 } catch (IOException e) {
32 // do nothing
33 }
34 }
35
36 // Cast unsigned character constants prior to comparison
37 return (b[0] == (byte) 'm' && b[1] == (byte) 'y' &&
38 b[2] == (byte) 'f' && b[3] == (byte) 'o' &&
39 b[4] == (byte) 'r' && b[5] == (byte) 'm' &&
40 b[6] == (byte) 'a' && b[7] == (byte) 't');
41 }
42
43 public void printOut(final URL url, final PrintStream ps)
44 throws IOException {
44 final InputStream stream = url.openStream();
45 if (checkMeta(stream)) {
46 try {
47 ps.println("content:");

```



```

48 byte[] b = new byte[1024];
49 int i;
50 while ((i = stream.read(b)) > 0) {
51     ps.print(new String(b, 0, i));
52 }
53 ps.println();
54 } catch (IOException e) {
55     e.printStackTrace();
56 }
57 }
58 }

```

On line 50, the input stream 'stream' is used after it was closed on line 30. This is another instance of an inter-procedural issue spanning another method.

#### 5. Empty catch clause

This last common Java coding mistake is nowhere near as important as some of the issues discussed above. In fact, it's a pretty low level issue but it's by far the most common maintainability issue.

```

12 public void openFile(String name) {
13     try {
14         FileInputStream is = new FileInputStream(name);
15         // read file ...
16     } catch (FileNotFoundException e) {
17         // TODO
18     }
19 }

```

Obviously, in this instance you should process this exception instead of just ignoring it. Aside from being the most common maintainability issue, it's also the most commonly ignored one.

On their own, these are fairly straightforward issues for a developer to deal with, but given the increased size and complexity of Java code, even in embedded applications, trying to identify these types of issues can be daunting. Source code analysis technology is one of the tools gaining in popularity as embedded development teams look for ways to continue building reliable, secure software.

These Java error types and many more can be found automatically with source code analysis. Through the use of these tools, the significant costs associated with software bugs both in terms of the productivity bottlenecks they cause during development and the havoc resulting from products released with defects can be greatly diminished.

#### About the author

**Alen Zukich** is Klocwork's director of product management and sets the company's product direction. With a technical and consulting background in the telecom equipment and software tools markets, Alen has helped hundreds of organizations successfully deploy source code analysis technology within some of the most demanding and complex development environments in the world. He can be reached at [alen.zukich@klocwork.com](mailto:alen.zukich@klocwork.com).

print email rss   0 Comments

#### Please sign in to post comment

Submit Comment
Follow Comments

#### More EE Times

Subscriptions	About Us
Newsletters	Privacy Policy
Reprints	EE Times Career Center
RSS Feeds	Contact Us
Media Kit	Email: <a href="mailto:feedback@eetimes.com">feedback@eetimes.com</a>
Sitemap	

#### EE Times Network

EE Times Asia	Electronic Supply & Manufacturing China
EE Times-China	Microwave Engineering Europe
EE Times-India	MCAD Online
EE Times Europe	TechOnline India
EE Times Japan	DeepChip.com
EE Times Korea	Design & Reuse
EE Times Taiwan	The RF Edge



All materials on this site  
copyright ©2010 EE Times Group.  
A UBM company  
All rights reserved