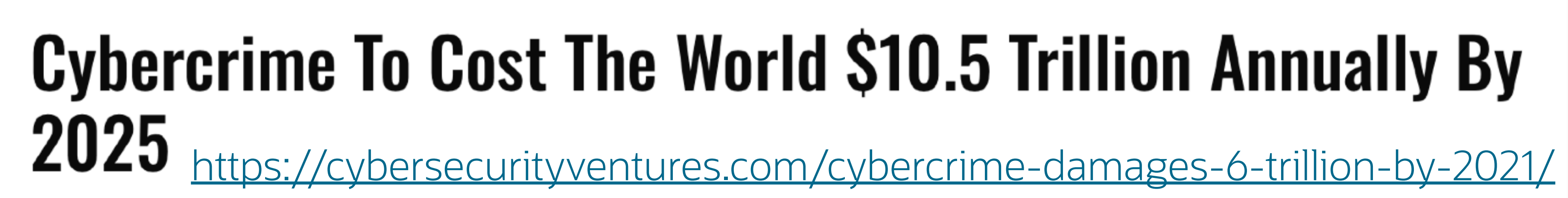
Integrity by Default

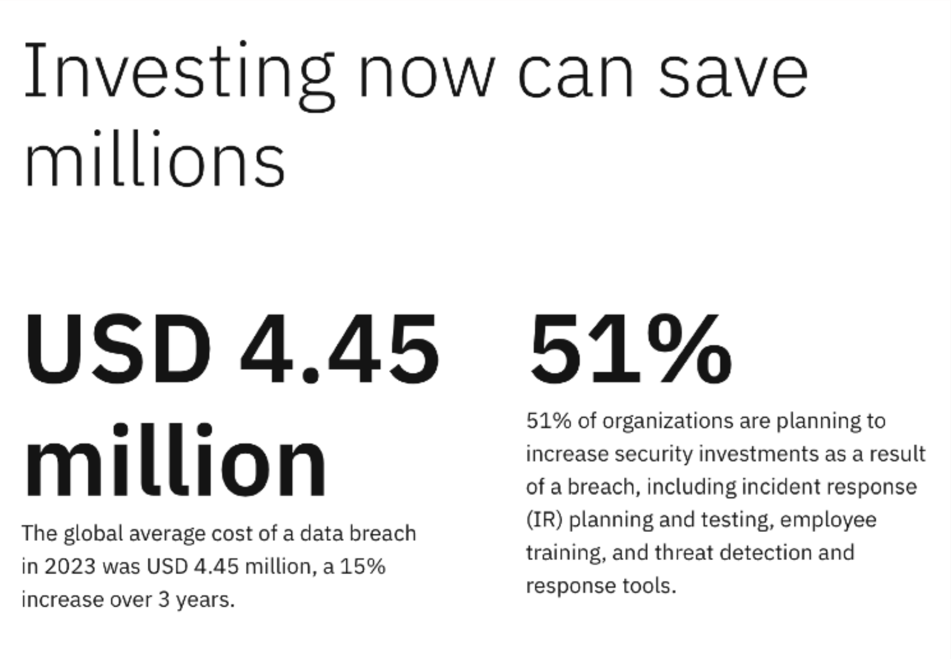
Ron Pressler

September 2023



Cybercrime To Cost The World $10.5 Trillion Annually By 2025

https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/



Investing now can save millions

USD 4.45 million

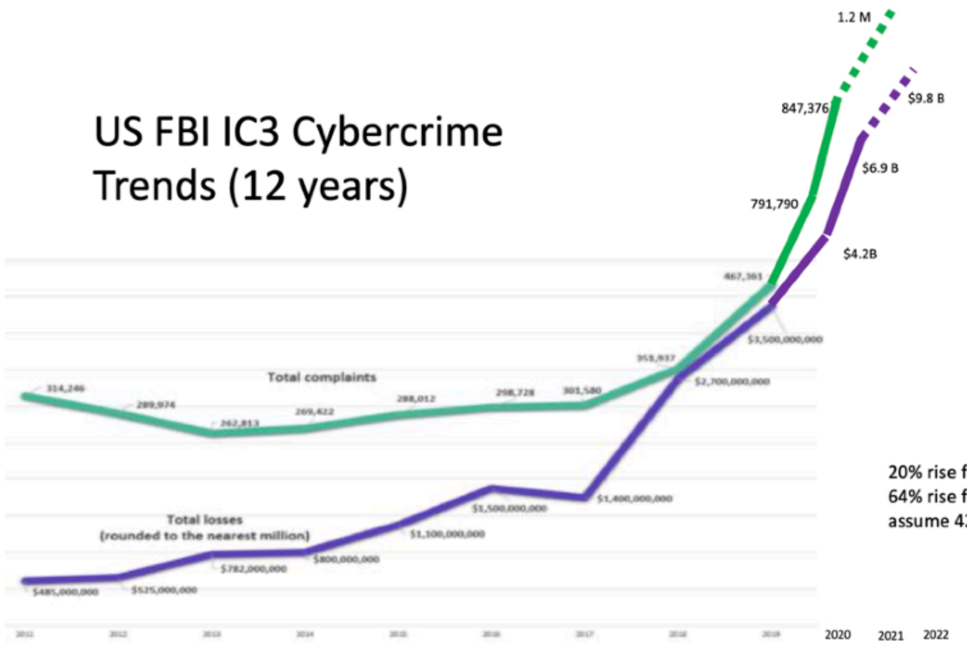
The global average cost of a data breach in 2023 was USD 4.45 million, a 15% increase over 3 years.

51%

51% of organizations are planning to increase security investments as a result of a breach, including incident response (IR) planning and testing, employee training, and threat detection and response tools.

https://www.ibm.com/reports/data-breach

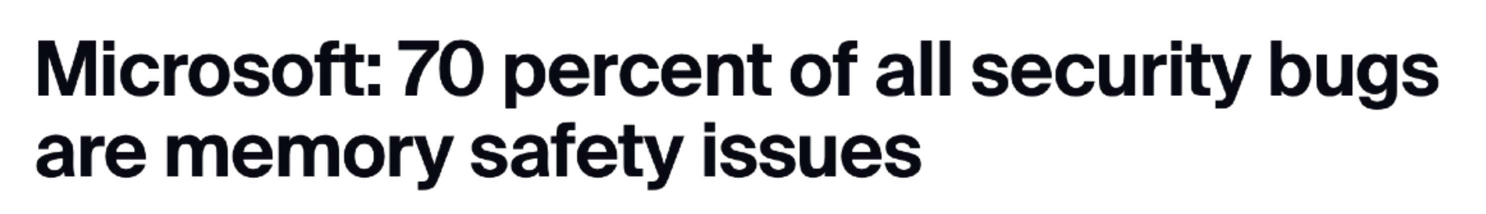
Figure 4-1 Cybercrime Trends in the US: Last 12 years



US FBI IC3 Cybercrime Trends (12 years)

20% rise from 2019 to 2020
64% rise from 2020 to 2021
assume 42% for 2022

IC3 – Internet Crime Complaint Center



Microsoft: 70 percent of all security bugs are memory safety issues

CISQ

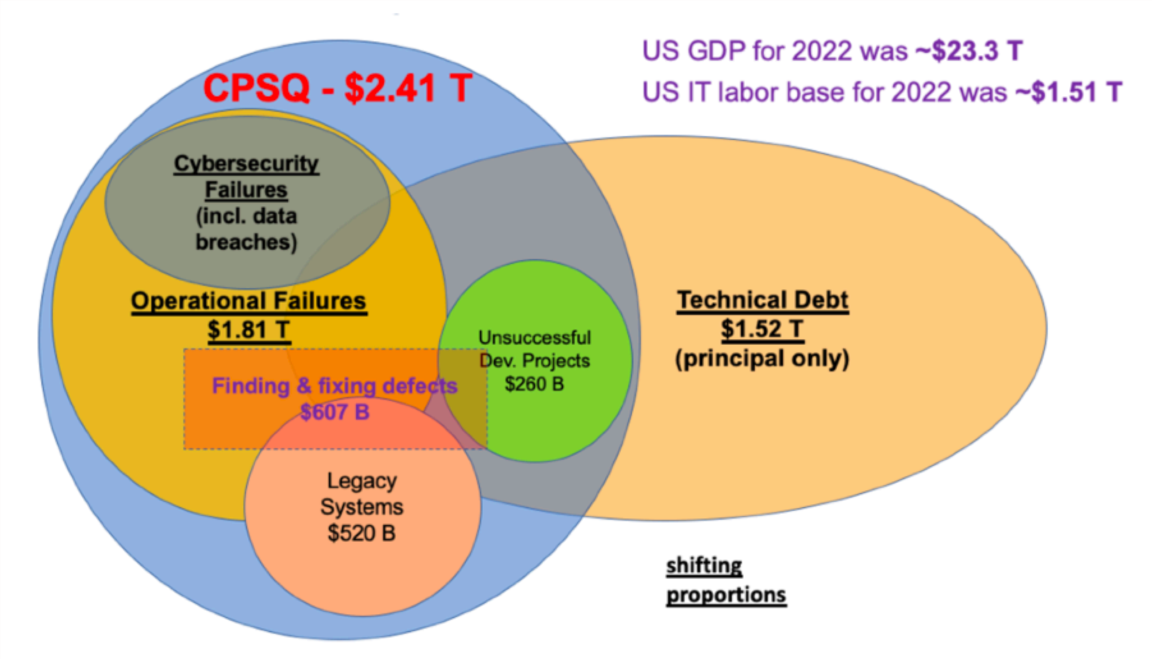
Consortium for Information & Software Quality™

The Cost of Poor Software Quality in the US: A 2022 Report

From Problem to Solutions

HERB KRASNER
MEMBER, ADVISORY BOARD
CONSORTIUM FOR INFORMATION & SOFTWARE QUALITY (CISQ)
WWW.IT-CISQ.ORG
HKRASNER@UTEXAS.EDU

DATE: DECEMBER 15, 2022



US GDP for 2022 was ~$23.3 T
US IT labor base for 2022 was ~$1.51 T

CPSQ - $2.41 T

Cybersecurity Failures (incl. data breaches)

Operational Failures $1.81 T

Finding & fixing defects $607 B

Unsuccessful Dev. Projects $260 B

Legacy Systems $520 B

Technical Debt $1.52 T (principal only)

shifting proportions

https://www.synopsys.com/software-integrity/resources/analyst-reports/cost-poor-quality-software.html

Integrity is an advanced concept, but it’s the answer to the most serious, costly problems that software developers face.

PART I

How things have changed

Java’s Backward Compatibility

- A remarkable success considering age, size of ecosystem, depth of dependency graphs
- Achieved through the Java SE Specification, but also applies to supported JDK APIs
- Standard APIs are only removed by a deprecation process spanning multiple releases (or by Maintenance Reviews)
  - Even then, only done for APIs that are not widely used or have good alternatives
- Platform components may be restricted in a gradual process spanning multiple releases and involving warnings
- Changes reviewed through the CSR process

Java’s Backward Compatibility

- Not just a principle, but **one of Java’s greatest strengths!**
- When companies invest in expensive software development, they want to preserve their investment:
  - Existing code continues working
  - Platform evolves to offer better performance and new functionality as requirements and environments change

… At Least in Theory

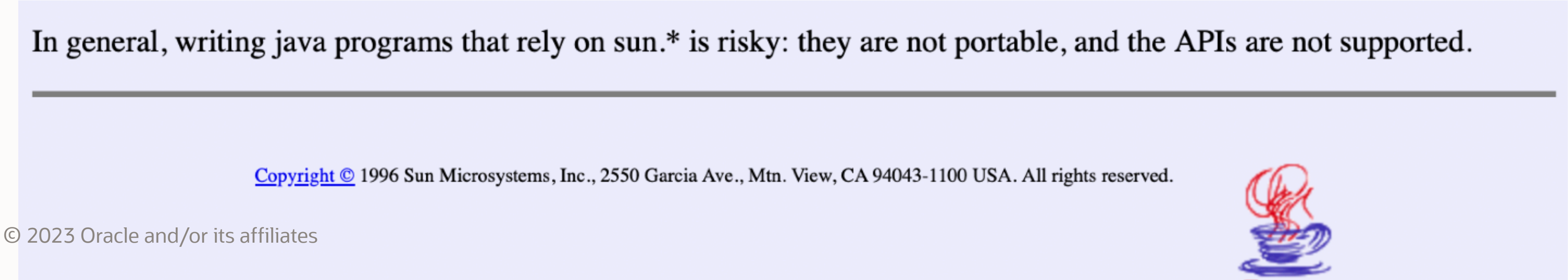
- In Java’s first decade, things were only added, rarely removed — started small
- Then Java experienced some years of slow evolution

Using internals

- Not many new APIs were added and some bugs remained unfixed
- Ecosystem reached for JDK internals
  - New functionality
  - Work around bugs
  - Improve performance

Using internals

- Not many new APIs were added and some bugs remained unfixed
- Ecosystem reached for JDK internals
  - New functionality
  - Work around bugs
  - Improve performance
- Still, it was technical debt



In general, writing java programs that rely on sun.* is risky: they are not portable, and the APIs are not supported.

Copyright © 1996 Sun Microsystems, Inc., 2550 Garcia Ave., Mtn. View, CA 94043-1100 USA. All rights reserved.

Using internals

- Not many new APIs were added and some bugs remained unfixed
- Ecosystem reached for JDK internals
  - New functionality
  - Work around bugs
  - Improve performance
- Still, it was technical debt
- *Effectively* backward compatible because internals didn't change much

Using internals

- Not many new APIs were added and some bugs remained unfixed
- Ecosystem reached for JDK internals
  - New functionality
  - Work around bugs
  - Improve performance
- Still, it was technical debt
- *Effectively* backward compatible because internals didn't change much

OSSIFICATION

What Happened in JDK 9?

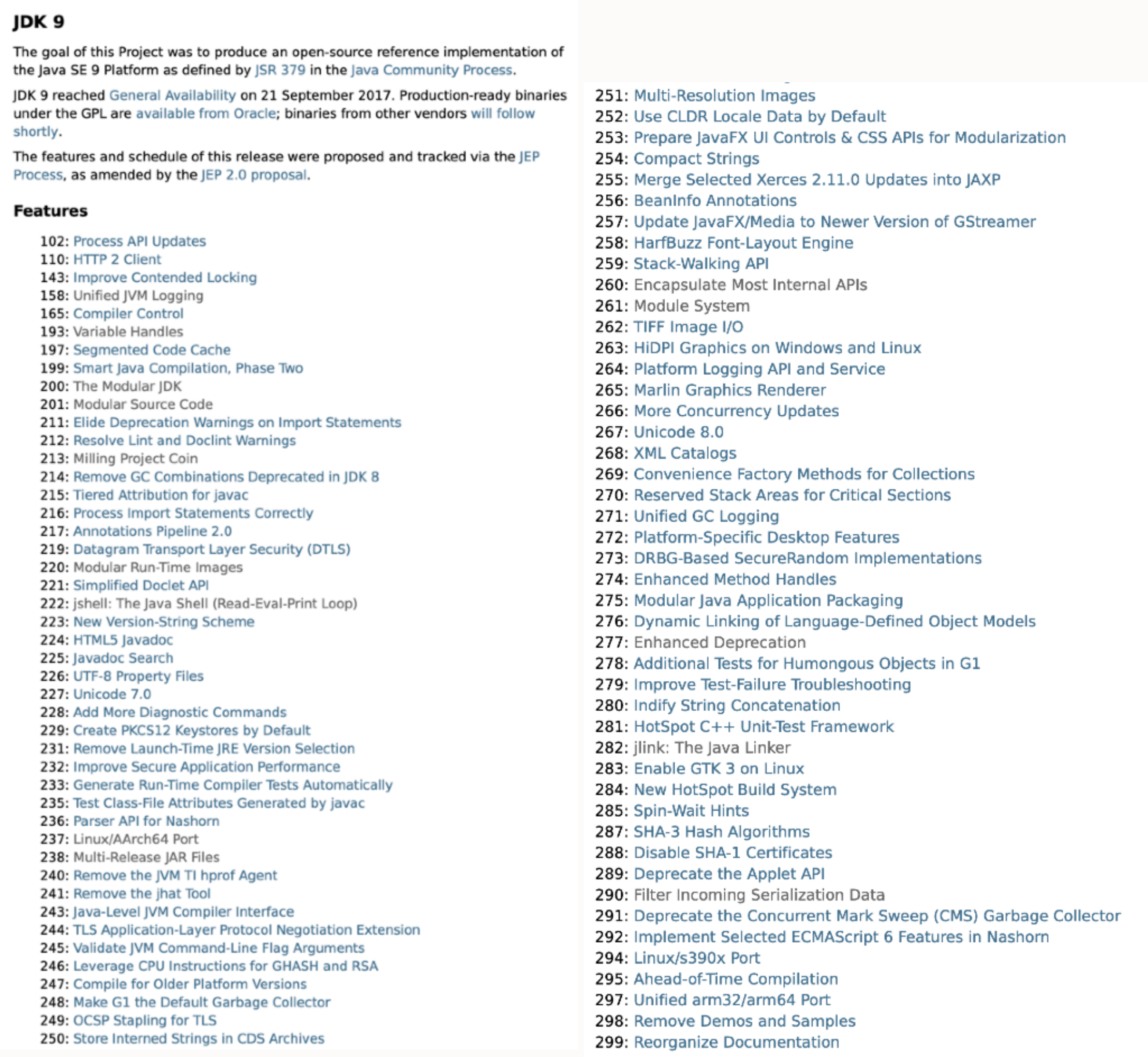
- Modules restricted access to internals breaking lots of libraries
- `sun.misc.Unsafe` was removed, breaking more libraries

Nah

- ~~Modules restricted access to internals breaking lots of libraries~~
  - Modules' strong encapsulation of internals wasn't turned on until JDK 16. All runtime access to internals remained as it was in JDK 8 until then
- ~~`sun.misc.Unsafe` was removed, breaking more libraries~~
  - `sun.misc.Unsafe` is still here, exactly as accessible as ever

What Really Happened?

Java picked up its pace



JDK 9

The goal of this Project was to produce an open-source reference implementation of the Java SE 9 Platform as defined by JSR 379 in the Java Community Process.

JDK 9 reached General Availability on 21 September 2017. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

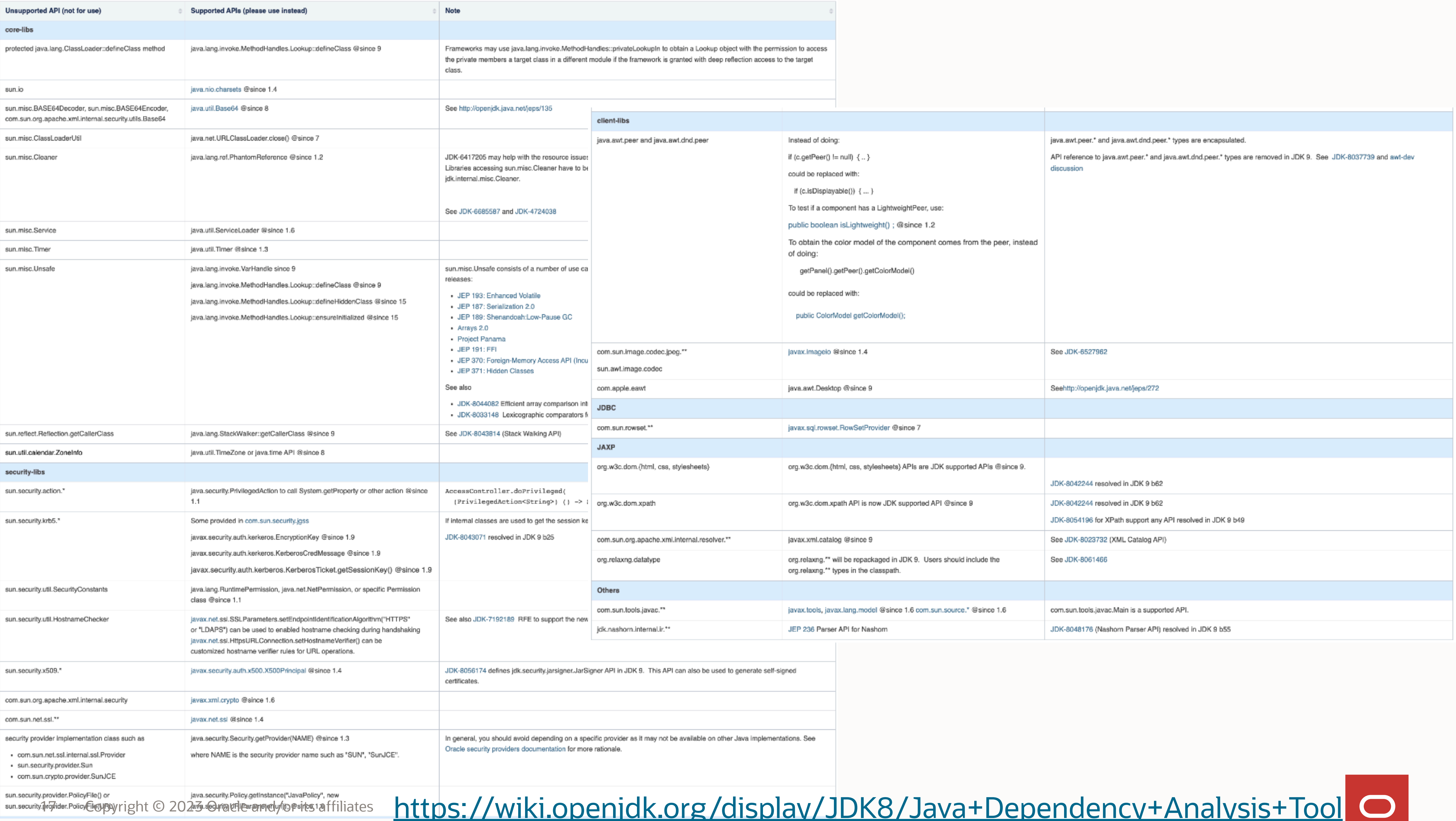
The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal.

Features

102: Process API Updates
110: HTTP 2 Client
143: Improve Contended Locking
158: Unified JVM Logging
165: Compiler Control
193: Variable Handles
197: Segmented Code Cache
199: Smart Java Compilation, Phase Two
200: The Modular JDK
201: Modular Source Code
211: Elide Deprecation Warnings on Import Statements
212: Resolve Lint and Doclint Warnings
213: Milling Project Coin
214: Remove GC Combinations Deprecated in JDK 8
215: Tiered Attribution for javac
216: Process Import Statements Correctly
217: Annotations Pipeline 2.0
219: Datagram Transport Layer Security (DTLS)
220: Modular Run-Time Images
221: Simplified Doclet API
222: jshell: The Java Shell (Read-Eval-Print Loop)
223: New Version-String Scheme
224: HTML5 Javadoc
225: Javadoc Search
226: UTF-8 Property Files
227: Unicode 7.0
228: Add More Diagnostic Commands
229: Create PKCS12 Keystores by Default
231: Remove Launch-Time JRE Version Selection
232: Improve Secure Application Performance
233: Generate Run-Time Compiler Tests Automatically
235: Test Class-File Attributes Generated by javac
236: Parser API for Nashorn
237: Linux/AArch64 Port
238: Multi-Release JAR Files
240: Remove the JVM TI hprof Agent
241: Remove the jhat Tool
243: Java-Level JVM Compiler Interface
244: TLS Application-Layer Protocol Negotiation Extension
245: Validate JVM Command-Line Flag Arguments
246: Leverage CPU Instructions for GHASH and RSA
247: Compile for Older Platform Versions
248: Make G1 the Default Garbage Collector
249: OCSP Stapling for TLS
250: Store Interned Strings in CDS Archives
251: Multi-Resolution Images
252: Use CLDR Locale Data by Default
253: Prepare JavaFX UI Controls & CSS APIs for Modularization
254: Compact Strings
255: Merge Selected Xerces 2.11.0 Updates into JAXP
256: BeanInfo Annotations
257: Update JavaFX/Media to Newer Version of GStreamer
258: HarfBuzz Font-Layout Engine
259: Stack-Walking API
260: Encapsulate Most Internal APIs
261: Module System
262: TIFF Image I/O
263: HiDPI Graphics on Windows and Linux
264: Platform Logging API and Service
265: Marlin Graphics Renderer
266: More Concurrency Updates
267: Unicode 8.0
268: XML Catalogs
269: Convenience Factory Methods for Collections
270: Reserved Stack Areas for Critical Sections
271: Unified GC Logging
272: Platform-Specific Desktop Features
273: DRBG-Based SecureRandom Implementations
274: Enhanced Method Handles
275: Modular Java Application Packaging
276: Dynamic Linking of Language-Defined Object Models
277: Enhanced Deprecation
278: Additional Tests for Humongous Objects in G1
279: Improve Test-Failure Troubleshooting
280: Indify String Concatenation
281: HotSpot C++ Unit-Test Framework
282: jlink: The Java Linker
283: Enable GTK 3 on Linux
284: New HotSpot Build System
285: Spin-Wait Hints
287: SHA-3 Hash Algorithms
288: Disable SHA-1 Certificates
289: Deprecate the Applet API
290: Filter Incoming Serialization Data
291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector
292: Implement Selected ECMAScript 6 Features in Nashorn
294: Linux/s390x Port
295: Ahead-of-Time Compilation
297: Unified arm32/arm64 Port
298: Remove Demos and Samples
299: Reorganize Documentation

What Changed?

- The JDK is changing more quickly
  - Reaching for internals can no longer work (the tech debt collector has come)
  - But it is also no longer needed as new standard APIs are added



Unsupported API (not for use)	Supported APIs (please use instead)	Note
core-libs		
protected java.lang.ClassLoader::defineClass method	java.lang.invoke.MethodHandles.Lookup::defineClass @since 9	Frameworks may use java.lang.invoke.MethodHandles::privateLookupIn to obtain a Lookup object with the permission to access the private members a target class in a different module if the framework is granted with deep reflection access to the target class.
sun.io	java.nio.charsets @since 1.4	
sun.misc.BASE64Decoder, sun.misc.BASE64Encoder, com.sun.org.apache.xml.internal.security.utils.Base64	java.util.Base64 @since 8	See http://openjdk.java.net/jeps/135
sun.misc.ClassLoaderUtil	java.net.URLClassLoader.close() @since 7	
sun.misc.Cleaner	java.lang.ref.PhantomReference @since 1.2	JDK-6417205 may help with the resource issues Libraries accessing sun.misc.Cleaner have to be jdk.internal.misc.Cleaner. See JDK-6685587 and JDK-4724038
sun.misc.Service	java.util.ServiceLoader @since 1.6	
sun.misc.Timer	java.util.Timer @since 1.3	
sun.misc.Unsafe	java.lang.invoke.VarHandle since 9 java.lang.invoke.MethodHandles.Lookup::defineClass @since 9 java.lang.invoke.MethodHandles.Lookup::defineHiddenClass @since 15 java.lang.invoke.MethodHandles.Lookup::ensureInitialized @since 15	sun.misc.Unsafe consists of a number of use ca releases: • JEP 193: Enhanced Volatile • JEP 187: Serialization 2.0 • JEP 189: Shenandoah:Low-Pause GC • Arrays 2.0 • Project Panama • JEP 191: FFI • JEP 370: Foreign-Memory Access API (Incu • JEP 371: Hidden Classes See also • JDK-8044082 Efficient array comparison int • JDK-8033148 Lexicographic comparators f
sun.reflect.Reflection.getCallerClass	java.lang.StackWalker::getCallerClass @since 9	See JDK-8043814 (Stack Walking API)
sun.util.calendar.ZoneInfo	java.util.TimeZone or java.time API @since 8	
security-libs		

sun.security.action.*	java.security.PrivilegedAction to call System.getProperty or other action @since 1.1	`AccessController.doPrivileged(`
		`(PrivilegedAction<String>) () -> `

sun.security.krb5.*	Some provided in com.sun.security.jgss javax.security.auth.kerkeros.EncryptionKey @since 1.9 javax.security.auth.kerkeros.KerberosCredMessage @since 1.9 javax.security.auth.kerberos.KerberosTicket.getSessionKey() @since 1.9	If internal classes are used to get the session ke JDK-8043071 resolved in JDK 9 b25
sun.security.util.SecurityConstants	java.lang.RuntimePermission, java.net.NetPermission, or specific Permission class @since 1.1	
sun.security.util.HostnameChecker	javax.net.ssl.SSLParameters.setEndpointIdentificationAlgorithm("HTTPS" or "LDAPS") can be used to enabled hostname checking during handshaking javax.net.ssl.HttpsURLConnection.setHostnameVerifier() can be customized hostname verifier rules for URL operations.	See also JDK-7192189 RFE to support the new
sun.security.x509.*	javax.security.auth.x500.X500Principal @since 1.4	JDK-8056174 defines jdk.security.jarsigner.JarSigner API in JDK 9. This API can also be used to generate self-signed certificates.
com.sun.org.apache.xml.internal.security	javax.xml.crypto @since 1.6	
com.sun.net.ssl.**	javax.net.ssl @since 1.4	
security provider implementation class such as • com.sun.net.ssl.internal.ssl.Provider • sun.security.provider.Sun • com.sun.crypto.provider.SunJCE	java.security.Security.getProvider(NAME) @since 1.3 where NAME is the security provider name such as "SUN", "SunJCE".	In general, you should avoid depending on a specific provider as it may not be available on other Java implementations. See Oracle security providers documentation for more rationale.
sun.security.provider.PolicyFile() or sun.security.provider.PolicyFile(URL)	java.security.Policy.getInstance("JavaPolicy", new java.security.URIParameter(uri)) @since 1.4	

client-libs		
java.awt.peer and java.awt.dnd.peer	Instead of doing: if (c.getPeer() != null) { .. } could be replaced with: if (c.isDisplayable()) { ... } To test if a component has a LightweightPeer, use: public boolean isLightweight() ; @since 1.2 To obtain the color model of the component comes from the peer, instead of doing: getPanel().getPeer().getColorModel() could be replaced with: public ColorModel getColorModel();	java.awt.peer.* and java.awt.dnd.peer.* types are encapsulated. API reference to java.awt.peer.* and java.awt.dnd.peer.* types are removed in JDK 9. See JDK-8037739 and awt-dev discussion
com.sun.image.codec.jpeg.** sun.awt.image.codec	javax.imageio @since 1.4	See JDK-6527962
com.apple.eawt	java.awt.Desktop @since 9	Seehttp://openjdk.java.net/jeps/272
JDBC		
com.sun.rowset.**	javax.sql.rowset.RowSetProvider @since 7	
JAXP		
org.w3c.dom.{html, css, stylesheets}	org.w3c.dom.{html, css, stylesheets} APIs are JDK supported APIs @since 9.	JDK-8042244 resolved in JDK 9 b62
org.w3c.dom.xpath	org.w3c.dom.xpath API is now JDK supported API @since 9	JDK-8042244 resolved in JDK 9 b62 JDK-8054196 for XPath support any API resolved in JDK 9 b49
com.sun.org.apache.xml.internal.resolver.**	javax.xml.catalog @since 9	See JDK-8023732 (XML Catalog API)
org.relaxng.datatype	org.relaxng.** will be repackaged in JDK 9. Users should include the org.relaxng.** types in the classpath.	See JDK-8061466
Others		
com.sun.tools.javac.**	javax.tools, javax.lang.model @since 1.6 com.sun.source.* @since 1.6	com.sun.tools.javac.Main is a supported API.
jdk.nashorn.internal.ir.**	JEP 236 Parser API for Nashorn	JDK-8048176 (Nashorn Parser API) resolved in JDK 9 b55

https://wiki.openjdk.org/display/JDK8/Java+Dependency+Analysis+Tool

What Changed? (Internal)

- The JDK is changing more quickly
  - Reaching for internals can no longer work (the tech debt collector has come)
  - But it is also no longer needed as new standard APIs are added
- More of the runtime is written in Java

What Changed? (External)

- Java applications primarily run on the server with a wide and deep dependency trees.
  - Security focus has shifted from defending against malicious code to the greater challenge of defending against vulnerabilities in benevolent code
  - One notable exception: Supply-chain attacks
- Server applications run in containers; want to “scale to zero”

PART II

What is integrity?

Integrity: The Ability to Promise

- **Invariant**:
  A property that's true everywhere in a section of code (entire program)
- **Integrity Invariant**:
  An invariant that is *guaranteed* to hold by the language/runtime

Example:

- No out-of-bounds access to an array may or may not be (but *should* be) an invariant in a C program; requires a full-code analysis
- `int[] a = new int[10]` establishes an *integrity invariant* in Java that no out-of-bound access can take place; guaranteed by the runtime

Integrity Invariants in Java

- No out-of-bounds array access
- No use-after-free
- No process crash

No undefined behavior

- No uninitialized data
- Runtime type-safety (`String` can’t be cast to `Socket`)
- Relative file paths are stable (no `chdir` operation)

Integrity invariants are *safety properties*: something “bad” never happens

Encapsulation: The Mother of Java Integrity?

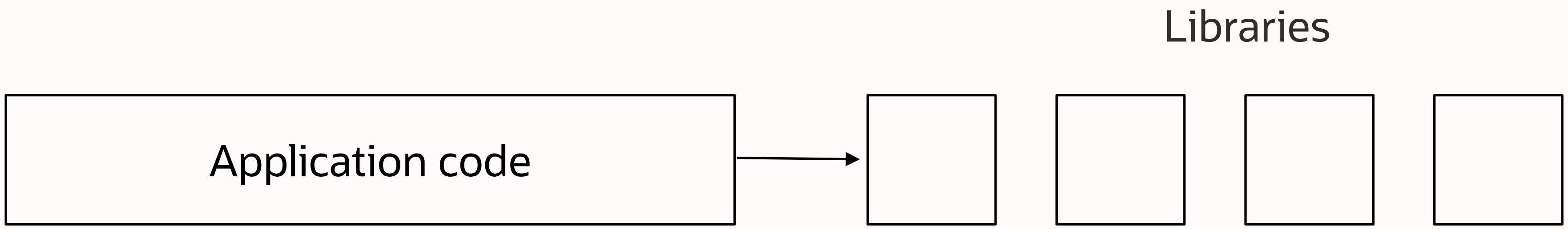
```
public final class Even {
    private int x = 0;
    public int value() { return x; }
    public void incrementByTwo() { x += 2; }
    public void decrementByTwo() { x -= 2; }
}
```

- New invariants can be created from an encapsulation invariant (no access rule violations)
- All integrity invariants depend on encapsulation; those on previous slides depend on native VM code being encapsulated from Java code.

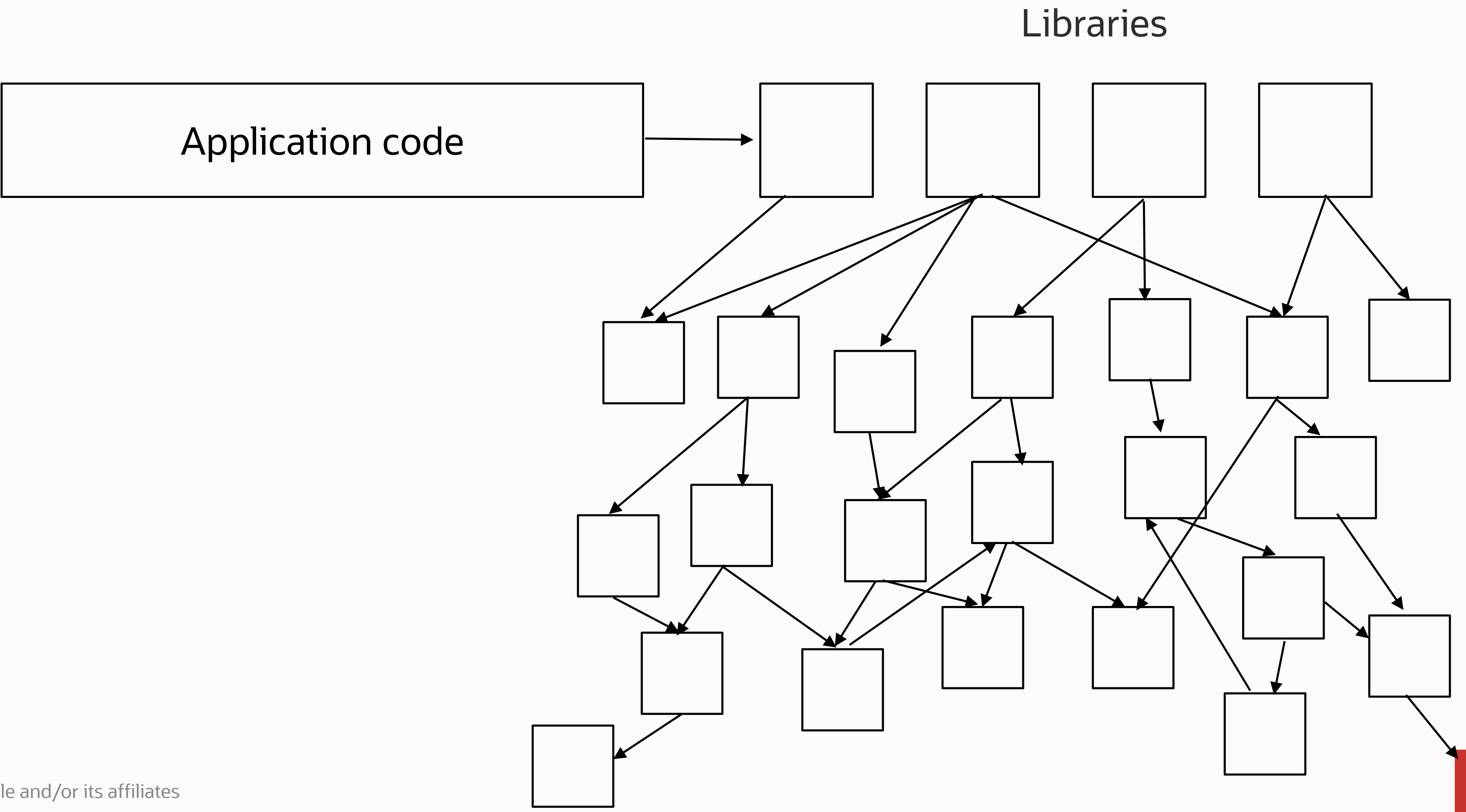
The Structure of a Modern Java Program

Application code

The Structure of a Modern Java Program



The Structure of a Modern Java Program



Encapsulation: The Mother of Java Integrity?

Any 4th-level dependency could violate the invariant:

- Deep reflection: `setAccessible`
- `sun.misc.Unsafe`
- `JNI`
- Dynamically load an agent and either redefine the methods (or, if class is not yet loaded, transform the field to `public`)

Impossible to establish *any* integrity invariant in Java if *any* of these is in play.
Invariance requires full-code analysis, same as buffer overflow in C

Encapsulation: The Mother of Java Integrity?

That’s why it matters that more of the runtime is being written in Java:

- JIT is written in Java: Java code could globally disable array bounds checking by encroaching on the JIT’s encapsulation
- Thread scheduling and monitors written in Java: Java code could globally disable the JMM by encroaching on the the implementation of the thread scheduler or the implementation of monitors

But surely they wouldn’t, would they?

PART III

The importance of integrity

The Importance of Integrity

- Evolution
- Security
- Performance

Integrity & Evolution

- Freedom to fearlessly change internals not subject to backward compatibility
- Why do libraries reach for internals
  - New functionality
  - Work around bugs
  - Improve performance
- No longer works in an age of faster-paced evolution
- No longer needed in an age of faster-paced evolution
- May be justified for a library individually, but over the wide and deep dependency ecosystem it leads to a *tragedy of the commons* that demands regulation
- Technical debt is secretly foisted on client applications

Integrity & Security

```
public final class Session {
    private boolean superuser
          = authorizeSuperuser();

    public void sensitiveOperation() {
        if (superuser) doSensitiveOperation();
        else throw new UnauthorizedException();
    }
    private void doSensitiveOperation() { ... }
    private boolean authorizeSuperuser() { ... }
}
```

Integrity & Security

```
public final class Session {
    private boolean superuser
          = authorizeSuperuser();

    public void sensitiveOperation() {
        if (superuser) doSensitiveOperation();
        else throw new UnauthorizedException();
    }
    private void doSensitiveOperation() { ... }
    private boolean authorizeSuperuser() { ... }
}
```

Could be set with deep reflection, JNI, Unsafe

Could be called with deep reflection/JNI

Could be redefined by an agent to always return `true`

Integrity & Security

- *Application*, uses `Session`. It also employs library *GoodSerializer* to deserialize JSON. *GoodSerializer* employs library *NeutralEncapsulationBreaker* to instantiate objects w/o constructor and assign private fields
- A bug in *GoodSerializer*'s input sanitation means an attacker could send an input to get it to set the private field `superuser`.
- Who's at fault?
  - *Application* is innocent and doing its best
  - *GoodSerializer* is well-intentioned, but bugs happen
  - *NeutralEncapsulationBreaker* has no vulnerability
- Any library that can break encapsulation, and any library that uses that library, becomes part of the attack surface area of any code that relies on encapsulation for integrity; we're back to full-code analysis

Integrity & Security

- Integrity is not a security mechanism, but no robust security mechanism can be created without it
- Offers “bulkheads” that compartmentalize the blast radius of a vulnerability

Integrity & Security: Aren’t we doomed, anyway?

- A Java library could write to the class files in the file system
- Plus: Spectre, Rowhammer etc.
- **Integrity of components is best enforced by their owner**
  - File system: OS
  - CPU cache: CPU
- Layers can cooperate, but each is in charge of its own integrity
- Java can and must enforce the integrity of the things it owns — Java code and objects — but shouldn’t (and can’t reliably) do more

Integrity & Performance

- Constant folding: Can a `final` field be constant folded? No, may be reassigned with deep reflection, JNI, or `Unsafe`.
- “Tree shaking”: Can a private method unused in a class be removed by a Condenser (that must preserve program meaning)? No, may be invoked with deep reflection or JNI (yes, we could try relying on speculation, but it makes some things much more complicated)
- Mechanical, meaning preserving transformations require *absolute* certainty

But surely they wouldn’t, would they?

- *Tragedy of the Commons*: A library author may feel *individually* justified
- *Unintentionality*: A vulnerability in library X can unintentionally make library Y use its superpowers for bad
- *Certainty*: Mechanical transformations require absolute certainty

PART IV (and last)

Integrity by Default

Integrity by Default

- Disabling the integrity of an invariant has a global effect
- A library (4th-level dependency) or a framework must not make a global decision

Integrity by Default: *Every exception to integrity must be explicitly acknowledged by the* ***application*** *in a centralized program configuration*

A centralized application configuration is an auditable record of integrity exceptions and accepted risks

The final say on module boundaries and privileges is given to the application

Strong Encapsulation: The Mother of Java Integrity!

Strong encapsulation: The encapsulation offered by Java's access control cannot be broken by code in a different module *by any means* unless:

- The declaring module explicitly grants some other module the permission to do so in `module-info` or programmatically (`java.lang.Module`)
- Code passes on its privileges to other code with a `MethodHandles.Lookup` capability object
- The application redraws the map of encapsulation boundaries with `--add-opens`/`--add-exports` flags.
- The application grants "superpowers" (JNI, agent) to some/all code

Outdated Libraries

- `--add-opens/exports` are not a “JDK 8 compatibility mode”. A program with many such flags for technical-debt reasons is a program that’s about to break.
- They’re “landmine markers” — keep you alive while you clear the landmines
- They add no burden because landmines must be found to be cleared
- Remember: The program will break even if we required no flags. Short of stopping Java’s evolution, there’s nothing we can do about that
- Plus, ensure that no new uses of internals can be added unnoticed; there’s incompatibility pain only *once* more and only one fix: stop using internals
- Without strong encapsulation by default, 8->9 migration pains would have continued forever and ever and ever (and no other integrity benefits, either)
- There are worse fates than an exception
- If a library is not updated to not require flags that’s a red flag that it’s improperly maintained

Outdated Libraries

- In the real world, companies don’t have the resources to fix technical debt
- That’s absolutely true
- It’s also true that in the real world some countries don’t have resources to fix bridges and make sure buildings are up to code
- And in the real world bridges collapse and buildings burn down
- There are consequences to risk whether we must take it or not
- Ignoring risk doesn’t make it go away; best to know where it is
- Tip: **Add a comment/git message explaining why each flag is needed**

Supporting Old JDK Versions

- Our advice:
  - Develop at the tip, and only for some recent-enough JDK version
  - Largely freeze old library versions.
  - Backport only security patches and serious bugs — not much work
- That’s what we’ve done at Oracle with the JDK since we introduced the LTS service

Operating beyond encapsulation boundaries

- Unit tests
  - Build tools and testing frameworks should *automatically* emit `--add-exports`, `--add-opens`, and `--patch-module` for the module under test, as appropriate
  - For mocking, use agents loaded at startup
- Frameworks
  - Should not use `add-opens` flags; use `MethodHandles.Lookup`
  - `static { AcmeFramework.grantAccess(MethodHandles.lookup()); }`
- APM tools — Use agents loaded at startup
- Serialization (A common cause of vulnerabilities)
  - We have a vision for encapsulation-respecting serialization
  - Until then try serializing only records, collections and other classes with well-known construction.

Foreign Code, Foreign Memory

Even with a memory safe language, memory management is not entirely memory safe. Most memory safe languages recognize that software sometimes needs to perform an unsafe memory management function to accomplish certain tasks. As a result, classes or functions are available that are recognized as non-memory safe and allow the programmer to perform a potentially unsafe memory management task. Some languages require anything memory unsafe to be explicitly annotated as such to make the programmer and any reviewers of the program aware that it is unsafe. Memory safe languages can also use libraries written in non-memory safe languages and thus can contain unsafe memory functionality. Although these ways of including memory unsafe mechanisms subvert the inherent memory safety, they help to localize where memory problems could exist, allowing for extra scrutiny on those sections of code.

National Security Agency | Cybersecurity Information Sheet | Software Memory Safety
Nov. ’22 https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/

Where We Want to Be

The NSA information sheet continues:

> For languages with an extreme level of inherent protection, considerable work may be needed to simply get the program to compile due to the checks and protections.

Not in Java!

- Java should be the safest mainstream programming language in the world
- Exceptions to integrity are tracked and localized in an auditable configuration
- The tax on those who don’t care is not large esp. if they use the classpath (no “localization”)

Why not *opt-in* to integrity?

- Most programs can remain under full integrity due to recent work
  - Enjoy portability and other benefits *much* more easily than ever
  - The minority that don't *will be inconvenienced*, but not much
  - Simple enough to be on by default and reduce attack surface area
- Tighter regulation (integrity) = lower entropy (fewer possible programs)
  - For new programs it's easy (and better) to start with low entropy
  - Old program need to expend energy to reach low entropy once

A Gradual Yet Resolute Path Forward

- Deep reflection restricted since JDK 16
- Dynamically loaded agents will be restricted (non-SE/optional)
- JNI will be restricted (optional component)
- Unsafe will be removed (non-SE)
- FFM will be restricted (starts out restricted)

As always, we’ll emit warnings & give ecosystem time to adapt

Reminder 1: All command line options can be placed in shared, full or partial, configuration “@files”

Reminder 2: `jlink` is flexible and widely applicable (more than some seem to think)

A Gradual Yet Resolute Path Forward

- JEP draft: Integrity and Strong Encapsulation
- JEP 451: Prepare to Disallow the Dynamic Loading of Agents
- JEP draft: Prepare to Restrict The Use of JNI
- More to follow