OpenJDK™

Java for Education

and “Small Java”

Ron Pressler

April 2024

ORACLE

- Java excels for writing large, “serious” applications; it is wildly popular at that
- Java must remain a “first language” to remain popular and grow
- Accommodating beginners is the essence of investment in sustained growth

What Educators Tell Us (about the Java experience and perception)

- “Activation Energy” is too high (the effort to get something up and running).
- “Hello World” is too complex.
- Many high school students use Chromebooks and can’t install Java.
- The ecosystem of libraries and tools is complex to navigate but necessary to use.
- Examples of creative, relevant, fun, modern Java projects are not easily discovered.
- Java is perceived as the language of legacy software, not of newer trends (AI, VisRec, Data).

- Python's technical challenge scaling up
- Disruption comes from below: easier languages that grab developers *early*
- Java's technical challenge is scaling *down*

Scaling Down: Small Java

- Scale down to less experienced users *and* to smaller projects/early stages
- Affordances for beginners serve veterans for tinkering and smaller programs
- Spans both language and tooling
- No separate mode, language dialect, or toolchain,
  rather an “**on-ramp**” for bigger, more advanced things

Small Java

- JEP 222: `jshell`: The Java Shell (Read-Eval-Print Loop) — JDK 9
- JEP 330: Launch Single-File Source-Code Programs — JDK 11

Hello, World?

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```


JEP 463: Implicit Classes

```
void main() {
    System.out.println("Hello, World!");
}
```


Upcoming:

```
void main() {
    println("Hello, World!");
}
```


Upcoming:

```
java.io.Console.Basic:

public static void println(Object obj) ...
public static void print(Object obj) ...
public static String input(String prompt) ...
```


Upcoming:

```
void main() {
    var authors = List.of("James", "Bill", "Guy", "Alex", "Dan", "Gavin");
    for (var name : authors) {
        println(name + ": " + name.length());
    }
}
```


Upcoming:

```
import module java.base;
```


Growing from tinkering, exploration, & small programs to large ones

0 files: jshell

1 file: source-code launcher

2 files: 1. Pick a build tool. 2. Learn the build tool. 3. Use the build tool.

JEP 458: Launch Multi-File Source-Code Programs (22)

```
stage1/
  Main.java
```

```
stage2/
  Main.java
  ComponentA.java
  ComponentB.java
```

```
stage3/
  Main.java
  component.a/
    :
  component.b/
    :
```

Run ’em all with:

```
$ java Main.java
```

- Makes the transition from small programs to large ones gradual
- Developer chooses whether and when to configure a build tool.

OpenJDK™

Integrity by Default

Ron Pressler

April 2024

ORACLE

The Importance of Integrity to Java

- Evolution/Migration
- Security
- Performance

JEPs

- Integrity by Default
- JEP 403: Strongly Encapsulate JDK Internals (17)
- JEP 451: Prepare to Disallow the Dynamic Loading of Agents (21)
- JEP 454: Foreign Function & Memory API (22)
- Prepare to Restrict the Use of JNI
- Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal
- Final means Final

Prepare to Restrict the Use of JNI

- Parity with FFM
- The first time a module loads a native library (`System.loadLibrary` etc.) or a native method it defines is called — warning
- `--enable-native-access=M1,M2…` (`ALL-UNNAMED`)

Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal

- Vast majority of Unsafe methods deprecated for removal
- Followed by a process of runtime warnings, errors, removal

Final Means Final

- Require additional flag to allow setting finals with `setAccessible`
- Allows performance improvements even for code on the classpath

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 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”

Focus: Compatibility

- Java is (largely) backward-compatible — ***through the SE spec***
- Most compatibility issues due to libraries *bypassing* the spec
- Forward looking: Integrity makes applications aware of portability risks imposed by dependencies (avoid most future compatibility pain)
- Backward looking: What about libraries that are still not portable?

OpenJDK™

Tip & Tail
For library development

Ron Pressler

April 2024

ORACLE

The question(s)

- How to address the different needs of my library's consumers?

- ⇒ What JDK baseline to choose?

The old answer

- One-size-fits-all — a single release train for everyone

- ⇒ One codebase baselined on an old JDK

Why?

- Multiple release trains targeting different JDK versions costs more

But why?

- Maintaining old release trains is costly

But *why*?

- We need to backport a lot — costly

The Two Classes of Consumers

- **Legacy**:
  - Application (or a component of an application) isn't changing much
  - Value stability
- **Evolving**:
  - Application (or a component of an application) is adapting to new requirements
  - Value added value: new features/capabilities/performance

The dichotomy

- Applications that want to avoid new JDKs do so because they want stability
- They usually want to avoid new library features, too
- Giving them new features hurts *everyone*:

The cost of one-size-fits-all

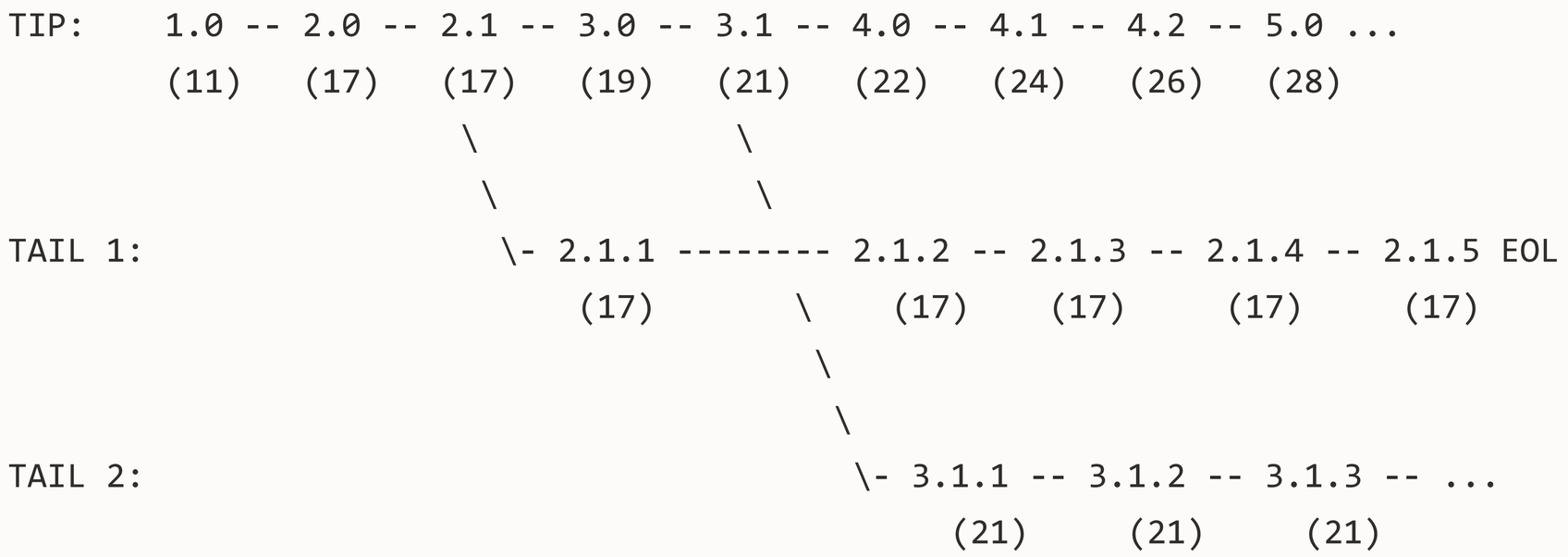
- Legacy users get less stability
- Evolving users can’t get all the features they need
  - At the very least, exploiting new JDK capabilities based on runtime queries increases development cost (and still doesn’t allow them in the APIs)
- Baselining on an old JDK is less fun for the library developer

Tip & Tail

- New features and enhancements (incl. performance) go into a **tip** release train
  - Can target a recent JDK
  - The lion share of effort is made easier because it's easy to add capabilities when targeting a new JDK without resorting to clever tricks
- Security patches + fixes to catastrophic bugs are backported to **tail** release trains
  - Otherwise they are largely left alone
  - Their cost is very low

Tip & Tail

- In the tip train, baseline each tip release on the JDK version that best supports the library's new features and enhancements.
- In a tail train, keep the JDK baseline as constant as possible over the life of the train.



Tip & Tail is the rare free lunch

- Costs less *for everyone*
  - Added process cost but lower cost overall (most effort in enhancements)
- Delivers more *to everyone*
  - Evolving users get more features (added value) quicker
  - Legacy users get more stability
  - Library maintainers get more fun — more motivation
- BUT — requires a *shift* in mindset and resources

A real conversation

- Please backport no-pinning-on-synchronized to 21
- Why won’t you upgrade to a new JDK?
- Because we certified the application on 21
- But if we backport such a change to 21 it won’t be the same 21
- (… but as long as you give the version the same name the process can remain)

The Choice

- Stability and new features are contradictory requirements
- Consumers *must choose one or the other* — not both
- Consumers must honestly accept this and change their process
- This requires a shift in mindset but doesn't introduce a new dilemma
  - Merely makes us honest about an inescapable tradeoff

A concrete example

- Problem: FFM was made permanent in 22 and Unsafe is being removed — can't be used in the same codebase
- Solution: Start new release train baselined on a new JDK and migrate to FFM. Keep old release train baselined on an old JDK and *leave it alone*.