Growing an ecosystem on the Java platform

Iulian Dragos
Typesafe Inc

The Premise

Ecosystems flourish on stable grounds

The Promise

- Software composes well
- Pick off-the-shelf (open source) components
- Glue them together
- Profit

The reality

It is **hard** to pick the right libraries that work **together**.

Abstraction

- Stable interfaces

- Pluggable implementations

Stable interfaces

- An interface specifies a contract
  - names, method signatures, etc.
- It can be automatically *checked* in a typed language

```
class TextFile {
  def readText(file: File): String
}
```

Binary Compatibility: One can simply swap one compiled binary library for another

Stable Interfaces

```
class TextFile {
  def readText(file: File): String
}
```

Stable Interfaces

```
class TextFile {
  def readText(file: File, encoding: String): String
}
```

- They *evolve* (so they are **not** stable)

Stable Interfaces

- They a

- The pr *ıration* of libraries and their versions that work together"
  - DLL Hell, Jar Hell

Dependency resolution

(more software)

Dependency resolution

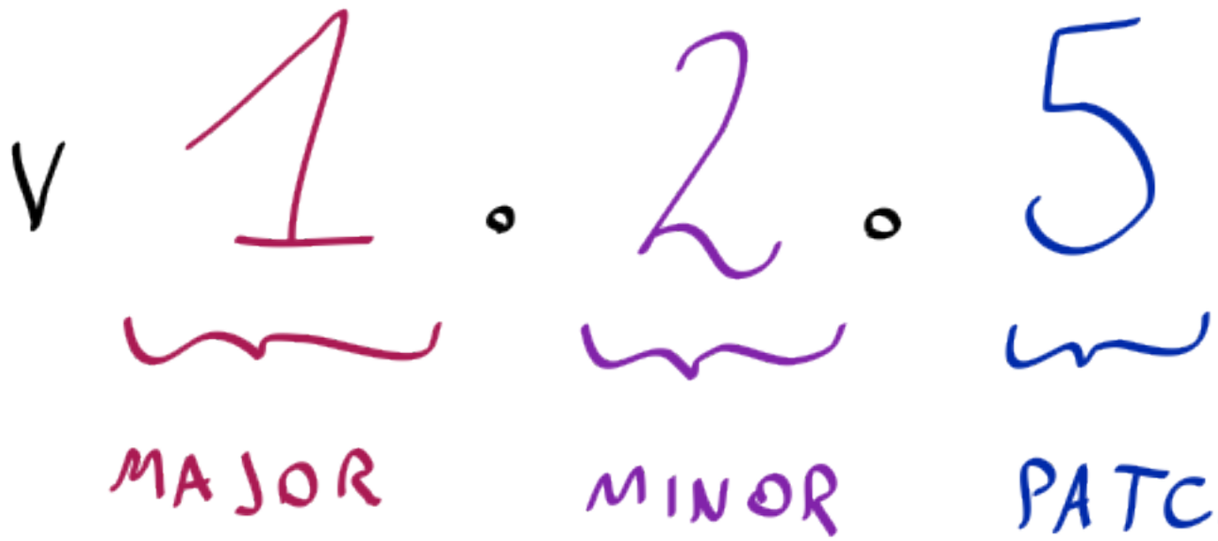
- Linux: apt-get, rpm (user-level) or autotools (dev side)
- Mac OS (homebrew, fink, etc.)
- Maven, Ivy for Java developers
- OSGi

Dependency resolution

- Allow upgrades/downgrades of individual libraries without ripple effects (i.e. human intervention)

- Use versions (or version ranges) to derive constraints

Semantic Versioning



- patch: drop-in replacement (binary compatible)
- minor: additional APIs (binary compatible)
- major: breaking changes

What is versioned?

	Maven/Ivy	OSGi
Granularity	Artifact (Jar)	Bundle (Jar) Package
Namespace	GroupID +ArtifactID	Fully Qualified Name

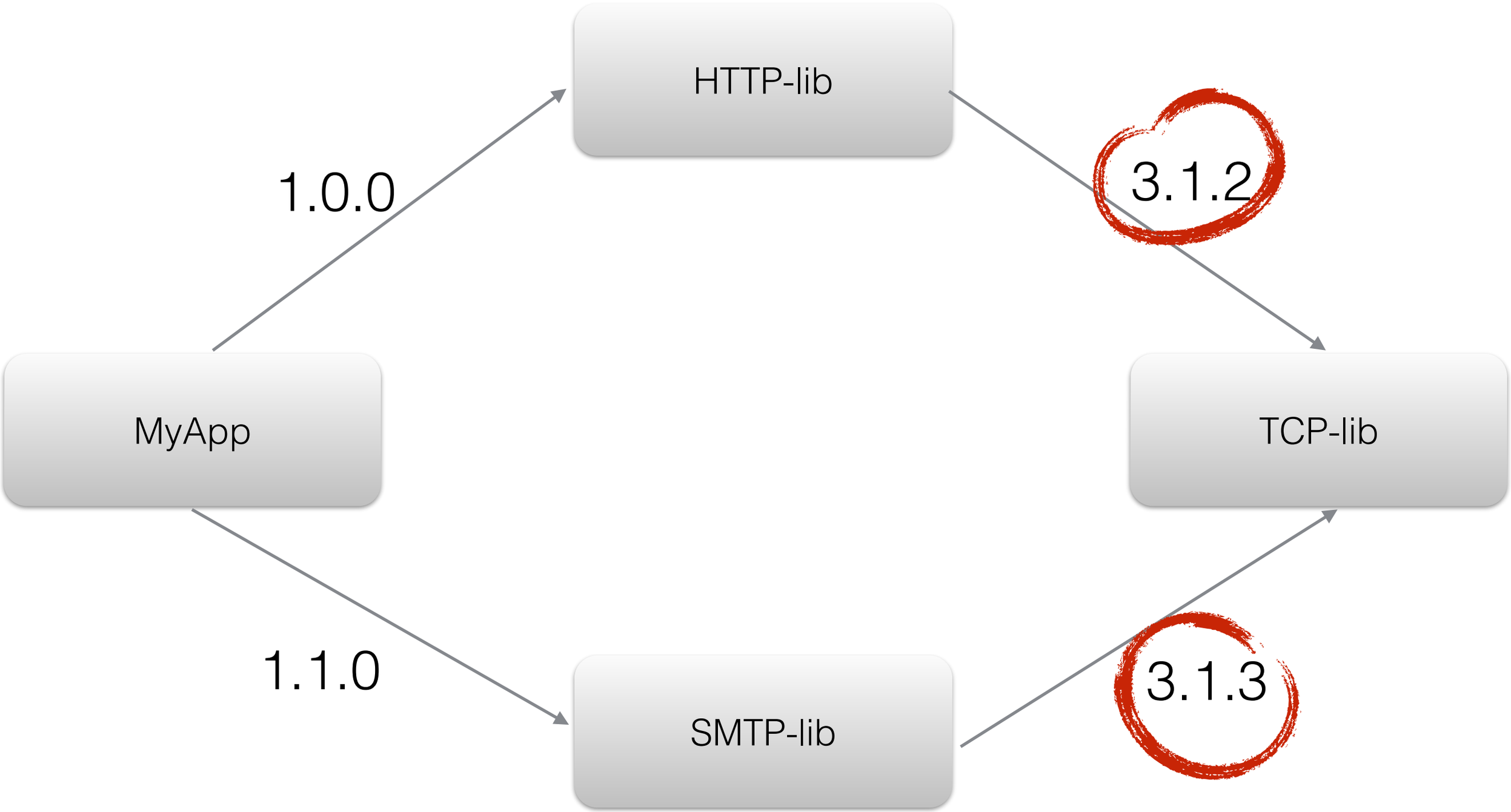
Example:

```
<dependency>
  <groupId>org.apache.lucene</groupId>
  <artifactId>lucene-core</artifactId>
    <version>4.1.0</version>
</dependency>
```

Require-Bundle:
org.eclipse.core.runtime,
org.scala-lang.scala-library;bundle-version="[2.11,2.12)"

Resolution

- The tool selects a version for each dependency
- May fail to find a workable configuration



A typical application has >200 libraries!

Multiple inheritance

- Libraries are used through different dependency chains
- Sometimes with different versions
  - resolution picks a compatible version based on
    - semantic versioning (OSGi)
    - distance — nearest-wins (Maven)

OSGi platform

- Resolution happens at runtime (**wiring**)
- Allows different versions of the same library (side-by-side)
  - avoids conflicts using classloader isolation
  - “communication” only through shared classes (for example, JDK objects)

Java Runtime

- **Everyone** depends on the JRE (standard library)
- JDK has **very strict** binary compatibility guarantees. That's why Java is still version 1.8!
  - ..and probably there won't be a Java 2.0 ever
- Ensures Java upgrades don't require a rebuild of the whole ecosystem
- (also, deprecated methods can **never** be removed)

The Scala Ecosystem

Scala ecosystem

- open-source
  - decentralized
- following Functional Programming principles
  - lots of small libraries
  - focus on composition

Binary compatibility

- Micro version is binary compatible
  - 2.11.0 —> 2.11.1 is a drop-in replacement
- Minor version **is not** binary compatible
  - 2.8.0 —> 2.9.0 requires rebuild of ecosystem
- Major version (epoch) is reserved for breaking *language* changes

Binary compatibility

- A given library can work with only **one** Scala major version

- ..therefore a Scala version (2.10) determines a *partition* of the ecosystem

- ..an organization has to **standardize** on such a version

More constraints

- Given a library, how do you know the Scala version it works with?
- 1st try: semantic versioning
    - version 1.0-1.99 works with 2.10
        - version 2.0-2.99 works with 2.11 etc.
- Downsides:
    - each library may have a different *base* version
    - a given version works with only *one* Scala version

Cross Compilation

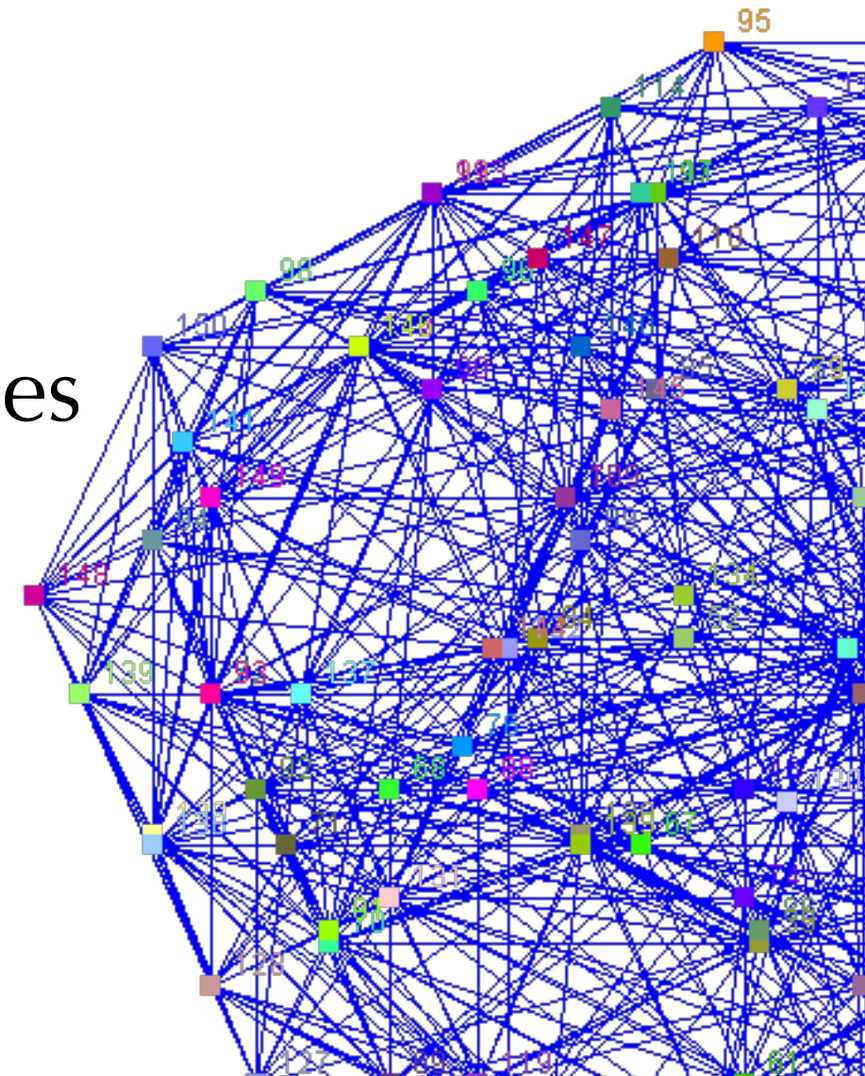
- Let's encode that in the name
  - scalatest_2.10 v. 1.0
  - scalatest_2.11 v. 1.0
- A library can be *cross-compiled* to many different Scala versions
- Uniform convention for all libraries

Major releases

- A Scala major release is a big thing
- happens every 1.5 years
- requires *everyone* to rebuild their code
- No cycles in the dependency graph!

Major releases

- What about unit testing frameworks?
- everyone depends on them
- they might depend on other libraries
  - hence a cycle is formed!



Dependency scoping

- Solution: Dependency *scope*
  - one set of dependencies for runtime
  - another set of dependencies for testing
- This way we can bootstrap the ecosystem in three steps
  - build/publish without test
  - build testing frameworks
  - rebuild/publish with tests

Upgrades

- Upgrading to a new major version (2.10—> 2.11) requires a *transaction*

- All projects that interoperate have to move together to the new version

Micro Services

There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

Anyone who doesn't do this will be fired. Thank you; have a nice day!

Jeff Bezos, Amazon

Micro services

- Decouple using lightweight HTTP servers
- Serialize to JSON (XML, proto buffers, etc)
- Services can evolve independently

Micro services

- There are also interfaces
  - So need versioning
  - but not statically checked :(
- Less coupling (no leaks of transitive dependencies)

Lessons learned

- Java has set a very high standard for BC
- People don’t like to rebuild their libraries
  - ..but also don’t like broken APIs
- Minimize breaking changes (deprecated methods stay in for 2 major releases — 3 years cycle)

Summary

- Dealing with dependencies is hard
- Version numbers are important
- Many solutions (Maven, apt, OSGi, microservices)