Uncovering Secrets of the Maven Repository: Java Build Aspects

An empirical analysis

G.J.T. Bot[1]

Supervisor(s): Dr. S. Proksch[1]**, M. Keshani**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: G.J.T. Bot
Final project course: CSE3000 Research Project
Thesis committee: Dr. S. Proksch, M. Keshani, S. Chakraborty

Abstract

The *Maven Central Repository*[1] hosts over 11 million packages[2]. As *Maven*[3] itself is a build tool for *Java*, the majority of these packages are Java archives.

This research aims to analyze these packages and look into various build aspects of these projects (the research questions): are Java modules used, what Java versions are used and how is the compiler configured? This is done by downloading a subset of these packages and looking at both the final artifact and the attached *Project Object Model* (Maven configuration). Specifically, one version is randomly selected from each distinct package hosted on Maven Central and then analyzed.

This research helps inform Java language developers and library maintainers. It captures parts of the build system evolution over a span of more than a decade. Precisely, version adoption, reproducibility and desirable build features are all important metrics for project maintainers in any software ecosystem.

In the end, 473352 packages were analyzed. Firstly, it was found that Java modules are rare: only 6919 (1.69% of the artifacts with an archive) of the packages use Java modules. Secondly, the most common Java version used is Java SE 8 (182476 packages) representing almost half of all analyzed packages. All *long-term support* versions (Java SE 8, 11, 17 and 21) make up roughly half of the available packages. Thirdly, only 54.98% of artifacts configure the *Maven compiler plugin* with the most used parameters being the source and target Java versions. The most common additional compiler flags are verbosity settings, providing more feedback and code analysis from both the linter and compiler.

Keywords: Java, Maven, Repository, Build Cycle, Configuration

1 Introduction

Java is currently one of the most widely used programming languages. It enables the creation of applications across diverse domains. As Java has grown increasingly popular over time, understanding the trends and practices used by developers becomes increasingly important. Java is a programming language designed to not be platform dependent. Famously, the term *write once, run anywhere* coined by *Sun Microsystems* - the creators of the Java programming language - describes this design philosophy. Furthermore, the language is intended to be backwards compatible. This means that a modern *Java Virtual Machine* (JVM) can run Java code compiled to an older version. Concretely, this means developers have virtually complete freedom in picking what version to use. This flexibility poses an interesting question: how do developers build their projects?

In this research, various aspects of building projects with *Maven*, a build tool for Java-based software projects, are examined. This is done in the form of an empirical analysis of projects hosted on the largest publicly available Maven repository: *Maven Central*. This repository hosts millions of packages released over the past decade. Specifically, the research questions are:

1. How commonly used are Java modules?
2. What are the popular Java versions used by libraries?
3. How is the compiler configured in the *POM*?

These research questions aim to provide an understanding of some common practices of Java developers in the ecosystem. This insight is valuable to Java language designers and library maintainers, as it will allow them to make informed decisions regarding feature adoption and version migration.

In this thesis, first, the related work is discussed. Next, the approach for the analysis is explained. Then, the ethical considerations are described. After that, the results from the research are covered followed by reflection on these results. Finally, the thesis is concluded with a swift reiteration of the research and results.

2 Related work

Version control is an essential part of software development, providing tools such as git to aid in single-project development. Maven, apart from being a build tool, also provides a repository specification to allow Java projects to build against dependencies available on such repositories. The largest publicly hosted repository is Maven Central, maintained by Sonatype.

Kula et al. [1] have analyzed the adoption (or absence thereof) of new releases for dependencies. On top of that the latency between new releases and dependencies adopting said releases is investigated. It was found that for existing projects, only 59.63% of their dependencies are adopted using their latest versions. In contrast, when maintainers add a new dependency to an existing project, in 81.16% of the cases the latest version is adopted. An important reason initial dependencies are not adopted with their latest version as often is because of incompatibility and general unawareness of updates, and adopting such versions takes time as a result. Note that these causes are not exhaustive. Over time, adopting the latest releases for dependencies is becoming more common.

Similarly, there has been research on the life cycles of libraries themselves. Kula et al. [2] have monitored library usage for various popular Maven packages to study library ageing. Library ageing was observed by looking at how many packages use/depend on a given library over time. Almost all libraries have a time of growth, a peak and then slowly decay. It was observed that most (93.87%) monitored packages have similar usage trends. More specifically, the trends can fit a first-, second- or higher-order module, the latter one being the most common. Moreover, the emergence or absence

[1] https://repo1.maven.org/maven2/

[2] https://mvnrepository.com/repos/central/

[3] https://maven.apache.org/

of rival libraries has an effect on the usage curves of existing libraries. The emergence of rival libraries often marks the peak of existing libraries, starting their decay. Note, however, that there is no significant reason for library migration at this point, so the decay may be very slow.

The Java Apache ecosystem is a large group of libraries provided by the *Apache Software Foundation*[4]. These libraries provide utilities in various areas such as math, logging, collections and I/O. In this prior research [3], changes in the ecosystem were analyzed. This includes (but is not limited to) dependency graphs, project size, releases, and active developers. Additionally, the behaviour of clients is also looked at. Specifically regarding their willingness to migrate to different versions and how much a client's codebase changes accompanying dependency version bump.

Unfortunately, not much research has been done on the adoption of new features. Moreover, the adoption of new (major) versions of a programming language is arguably more important but researched less. Newly introduced language features can aid software development and general productivity, let alone quality-of-life changes. On the other hand, since Java runs on a virtual machine, it is possible for new Java versions to improve run-time performance.

Not all versions add new features and tweak existing ones. An important group of updates are vulnerability patches. Java is also not free of vulnerabilities. The adoption of security patches for Java is very important as such vulnerabilities potentially affect a way larger group than vulnerabilities in single packages, after all virtually all Java programs run on a JVM.

Düsing and Hermann [4] have analyzed the adoption of library updates specifically for patching vulnerabilities. This was done for multiple repositories, including repositories for programming languages other than Java. It was found that for Maven, roughly 50% of all vulnerabilities are patched before their date of disclosure. Furthermore, on average vulnerabilities are patched 188.2 days before the vulnerability is published. Additionally, the majority of unpatched vulnerabilities are low and medium severity. Interestingly enough, 15.46% of artefacts released on Maven include dependencies for which public security advisories are available.

There has also been (limited) research into compiler configuration, but such research often prioritizes compiler optimization rather than analysing non-performance-related configuration flags.

Zhang et al. [5] propose a tool to detect and repair configurations for both continuous integration (CI) infrastructure and Java build tools (i.e. Maven and *Gradle*[5]). This tool is called *BuildSonic* and it focuses on various (common) performance-related configuration smells aiming to significantly decrease build times. One such configuration smell is not allowing Maven to fork the compiler plugin. Enabling this reduces the performance overhead of garbage collection when compiling a project with many source files. Others include not building separate Maven modules in parallel, providing not enough heap space for the build tool's JVM and declaring repositories in non-optimal order for dependency resolution. Out of the analyzed Java projects, it was found that 99.0% of these projects had configuration smells. Furthermore, the projects that implemented the proposed configuration changes had, on average, a 12.4% build-time improvement. Note that BuildSonic does not detect (and repair) all identified configuration smells as some require heavyweight analysis, which is against its design philosophy: BuildSonic is a linter.

[4]https://apache.org/

[5]https://gradle.org/

3 Methodology

This research is an empirical analysis of build aspects for projects in the Java ecosystem. For this analysis, a large data set is needed. Maven Central is the largest publicly hosted repository of Maven projects and will function as a foundation. Maven itself is widely used and has been around for a long time, making it an adequate data set representing the entire Java ecosystem. Furthermore, it has a policy that published artifacts cannot be updated or deleted. This provides solid historical data.

3.1 Data Selection

Due to the size of the central Maven repository, it is unpractical to analyze everything uploaded to the repository, so instead, a subset of the available artifacts is used.

First, the set of available artifacts is fetched. The Maven Central repository is indexed[6], i.e. there is a record resembling a logbook available with all additions to the repository. Unfortunately, there are no index records available for artifacts produced before 2011. The set of artifacts is stored in a database.

Second, a subset is randomly selected from this set of available packages. To avoid skewing data in favour of packages with more versions released, only one version per unique group id and artifact id is used[6]. This is done because different releases for the same package have a large overlap. The date of publishing is also important, as it places releases in a historical context, most noticeably the available Java versions at the time. The sample must have a similar trend of packages released per year. To achieve this, a subset per year is used. This is stratified sampling[7] and it ensures every year is represented. Importantly, note that the data selection process is made reproducible by using a set seed for the random number generation.

The resulting distribution can be seen in table 1.

3.2 Data Extraction

With the artifacts to analyze selected, every package is downloaded and opened. Additionally, every Maven project has an associated *pom.xml* file. This is the *Project Object Model* (POM): an *XML* file describing the Maven project itself. For the purposes of this research, the POM is used to fetch the compiler configuration. While most artifacts are Java code, it should be noted that this is not a requirement for artifacts to be hosted (or even built) by Maven. Furthermore, not all packaging types are *ZIP* archives. The data extraction is based

[6]https://repo1.maven.org/maven2/.index/

Year	Packages	Sample Count
2011	278326	24796 (8.9%)
2012	136552	13711 (10.0%)
2013	178376	15385 (8.6%)
2014	237086	19536 (8.2%)
2015	345994	28037 (8.1%)
2016	514152	34919 (6.8%)
2017	728262	39897 (5.5%)
2018	920570	44539 (4.8%)
2019	1218375	53072 (4.4%)
2020	1404258	51530 (3.7%)
2021	1773855	58590 (3.3%)
2022	1791565	59908 (3.3%)
2023	805670	35995(4.5%)
Total	10333041	479915 (4.6%)

Table 1: The artifact distribution per year. (sample size: 100%, seed: 0.5, index date: June 6, 2023)

on the assumption that the artifact does, in fact, contain Java (byte)code, and can be opened as a ZIP archive. Common artifacts extensions (`.jar`, `.war`, `.ear`) are all formatted as ZIP files. If the artifact is not openable as a ZIP file it is most likely not a Java artifact and will be ignored, but the POM will still be analyzed. If it is a ZIP file, determining whether it is related to Java is out of the scope of this research; only the presence of `.class` files (the file extension for files containing Java bytecode) is checked.

Each research question (see section 1) has its own set of data that needs to be extracted, described in the following subsections for each research question. To provide context, the following data is also available for each artifact:

1. The year of publishing (from the index file(s))
2. If the artifact could be opened (i.e. if it is a ZIP file)
3. The packaging type (from the index file(s))

Java Modules (RQ 1)

Java modules are defined by their respective module descriptors. These descriptors are put in a single file (per module), identifiable by its name: *module-info.class*. To check if a given archive uses Java modules, iterating over all its entries and checking if entries with this name exist is sufficient.

Java Versions (RQ 2)

There are two approaches to check what Java version a given archive uses.

For the first approach, one can look at `.class` files inside the archive. These are files containing Java bytecode. In their headers, there is a two-byte major version number and a two-byte minor version number. This indicates the *Java Virtual Machine* specification the class files complies with. The major version number corresponds directly to Java version releases, e.g. Java SE 1.8 for major version $0x0034$. Because archives can have multiple classes, it is possible to have files with different version numbers. Per artifact, a list of class versions together with their number of occurrences is stored if there is not just a singular Java class version.

The second approach uses the archive's metadata file (if present): *META-INF/MANIFEST.MF*. This text file containing metadata stores all sorts of information about the archive, including version information. The relevant entries are: `Created-By`, `Build-Jdk` and `Build-Jdk-Spec`. This approach comes with drawbacks, however. Firstly, none of these fields are 'official' (i.e. part of the Java specification), they are just commonly included. As a result, it is expected that a significant portion of archives does not have these entries or put in unusable/irrelevant information. Secondly, having built an archive using a given development kit version does guarantee the artifact will have bytecode for that Java version. The native *Javac* compiler can produce bytecode compliant with different Java versions. Nevertheless, both approaches are used for this research.

Additionally, the `Multi-Release` entry from the manifest is stored, which identifies multi-release archives. This is a language feature introduced in Java 9 designed to allow different class implementations for different Java versions to allow developers to use newer features without dropping support for older versions. These classes are located in their own version-specific directory inside the JAR, unlike non-multi-release archives. For these archives, the developer must ensure the *Java virtual machine* will never try to load class files compiled in a newer version as this will cause errors.

Compiler Configuration (RQ 3)

Maven takes care of compiling source code, meaning the compiler is configured in the project's POM. The Apache Maven ecosystem provides a compiler plugin: the *Apache Maven Compiler Plugin*. This plugin allows the user to override the default compilation settings and add manual configuration. Although it is possible for third-party plugins to handle compilation, for the purposes of this research the focus is on the Apache Maven Compiler Plugin.

In this research, the following parameters are considered:

- `compilerArgs`: additional command line parameters to pass to the compiler.
- `compilerId`: the ID of the compiler, e.g. *Javac*.
- `encoding`: the character encoding for source files.
- `source`: the Java specification the source code should comply with.
- `target`: the Java specification the bytecode should be suitable for.

An example of the plugin configuration can be seen in listing 1. Note that not all fields are necessarily available.

Listing 1: Example plugin configuration.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.11.0</version>
  <configuration>
    <encoding>UTF-8</encoding>
    <source>17</source>
    <target>17</target>
  </configuration>
</plugin>
```

With this approach, it is important to realize that projects without actual Java code can still have compiler configurations. A common example of this is a POM with packaging type `pom`; these projects are essentially containers for sub-modules which can inherit this configuration. Furthermore, since configurations are inheritable, both the `plugins` and `pluginManagement`, and project parent configurations are considered. Additionally, placeholders (entries of the form `${x}`) and the properties `maven.compiler.source` and `maven.compiler.target` (the default values of the `source` and `target` fields respectively) are resolved.

3.3 Implementation

The program is written in Java, using a *PostgreSQL* database for storage. The implementation is available on *GitHub*[7].

First, the `IndexerReader` reads repository index files and stores a list of artifacts in the database. Then, a `Selector` picks a sample from this list of artifacts to use for analysis: one random version per group id and artifact id (listing 2), and then taking a subset of the list of unique packages (listing 3). The program ended up being efficient enough, that all unique packages were analyzed, rather than a subset (i.e. the sample size is 100%). The seed used for the sample selection is 0.5. Next, the selected artifacts are downloaded from Maven Central by the `Resolver` and passed to various `Extractors`. An `Extractor` is a separate component that looks at the artifact and collects data. The data collected by the extractors are then stored in the database. The program diagram is seen in figure 1. Note that, to increase performance, multiple runners were run in parallel.

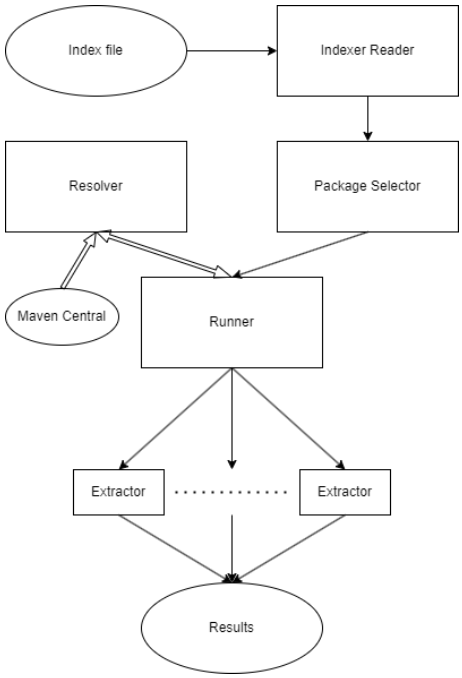

Figure 1: The application diagram.

Finally, a separate program[8], takes the data and performs analysis, providing statistics and distributions.

In this program, the manifest entries `Created-By`, `Build-Jdk` and `Build-Jdk-Spec` were parsed using the grammar in listing 4. These entries were merged since, for the purposes of this research, they serve the same purpose. For example, via this grammar, inputs like `10` get matched with Java SE 10, and inputs like `1.8.0_253 (Oracle)` get matched with Java SE 8 (with minor version 253). If multiple entries are specified containing different versions, the most recent version is picked.

In addition, the compiler arguments were sanitized as well:

- Strings containing whitespace characters likely contain multiple arguments, so they were split into separate arguments.
- `@<filename>` arguments are grouped under `@`.
- `-A<key>[=<value>]` arguments are grouped under `-A`.
- `-J<option>` arguments are grouped under `-J`.
- Arguments containing values (`-<key>=<value>` and `-<key>:<value>`) are grouped under `-<key>`.
- Strings not starting with `-` or `@` are assumed to be flag parameters, so they are excluded.

Listing 2: Distinct selection query.

```
SELECT setseed(seed);
CREATE TEMP TABLE package_list_distinct AS
(SELECT DISTINCT ON (groupid, artifactid) *
FROM package_list
ORDER BY groupid, artifactid, random());
```

Listing 3: Sample query.

```
INSERT INTO selected_packages
SELECT * FROM package_list_distinct
TABLESAMPLE BERNOULLI(sample_size)
REPEATABLE (seed)
WHERE DATE_PART('year', lastmodified) = year;
```

Listing 4: Java version grammar.

```
java_version := digit+
    [ period digit+ ]
    [ period digit+ ]
    [ underscore digit+ ]
    [ space par_l (not par_r)+ par_r ]
    ; major.minor.micro_patch (text)
digit := U+0030 | U+0031 | U+0032
    | U+0033 | U+0034 | U+0035
    | U+0036 | U+0037 | U+0038
    | U+0039 ; '0' - '9'
period := U+002E ; '.'
underscore := U+005F ; '_'
space := U+0020 ; ' ' (space)
par_l := U+0028 ; '('
par_r := U+0029 ; ')'
```

4 Responsible Research

This section discusses the main ethical concerns with empirical studies and how they are handled. Specifically, obtaining the data and reproducing the results.

[7] https://github.com/ashkboos/MavenSecrets/

[8] Module `visualization-build-aspects`

Data Usage
All data for this research is downloaded from Maven Central. To avoid disrupting their services, a large chunk (>2Tb) of the repository had been downloaded over the period of multiple years[9]. This local copy is used as a cache to reduce the number of download requests made. The downloaded artifacts are not outdated, as Sonatype's (the maintainer of the central repository) policy is that published resources cannot be deleted or changed to remain consistent[10]. All downloaded data is publicly available and does not contain sensitive information. The method of data selection used is designed to attempt to minimize bias to the best of the author's abilities.

Reproducibility
The entire research is reproducible by design. All code is open source[11]. By design, the data selection has a configurable seed (used for pseudo-random data selection) and sample size. For this research, seed 0.5 and sample size 100% were used. The required extractors are `ArtifactExistsExctractor`, `CompilerConfigExtractor`, `JavaModuleExtractor` and `JavaVersionExtractor`. The extracted and processed data are available at Zenodo (DOI: 10.5281/zenodo.807712), in addition to the index file used (retrieved on June 6, 2023).

5 Results

This section provides a comprehensive overview of the results obtained. It begins by dissecting the data selection. Then, each research question is addressed individually. For each research question, the relevant data obtained are discussed.

Data Sample
From the index file, 1033341 different packages were read. 4.64% of the packages in this list are distinct (i.e. they have a unique group id and artifact id). This means that one random version was picked per package. All these packages were selected for analysis. Most of the selected packages were downloaded successfully, with only 1.37% of them failing to be resolved. This is solely because of parent POMs not being resolved. Upon manual inspection, this happens when a project depends on a parent project that is not hosted on Maven Central[12]. Out of the resolved artifacts, 86.64% have ZIP formatted file. This is relevant for the research questions looking at Java code. The sample is broken down in table 2.

Set	Packages
Indexed packages	10333041 (100%)
Distinct	479915 (4.64%)
Sample size	479915 (4.64%)
Resolution fail	6563 (1.37%)
Resolution success	473352 (98.63%)
With archive	410102 (86.64%)

Table 2: Selection breakdown (sample size: 100%, seed: 0.5, index date: June 6, 2023)

How commonly used are Java modules? (RQ 1)
Out of all the analyzed packages with archive (410102 packages), 6919 (1.69%) use Java modules. Interestingly enough, 2021 (29.2%) of the artifacts using Java modules contain a majority of class files compiled in a Java version prior Java 9, the version Java modules are introduced in. In these cases, the module files are either generated during the build cycle or compiled in a different Java version and included in the final artifact. This is not an issue as the *Java Virtual Machine* (JVM) prior to Java 9 will never try to load these files (which would cause errors), but can still load the normal class files. Newer JVMs, however, will load and make use of the modules files.

What are the popular Java versions used by libraries? (RQ 2)
The most common Java version is Java SE 8, with the *long-term support* (LTS) versions (Java SE 8, 11, 17 and 21) making up 51.2% of the archives (only packages with archive are considered, 410102 packages). 48 (0.01%) of the artifacts use the latest Java version (Java SE 20 as of June 25, 2023). Additionally, note that old non-LTS versions (Java SE 5, 6 and 7) have a way more significant presence compared to the other non-LTS versions. The full distribution per major version is seen in table 3. Note that this table lists the most common Java version based on class file versions, and is grouped by major release.

Furthermore, there is one artifact compiled in Java SE 21[13]. As this Java version is not released yet, the published artifact must have been built with an early-access version, and as such is not guaranteed to be able to run on Java SE 21 once it releases.

Looking at the Java version distribution per year (see table 2), LTS versions catch on relatively fast. Within two years since the release of these versions, they become one of the more common versions. Even though other LTS versions are adopted significantly more than non-LTS versions, Java 8 is by far the most popular version. Showing the release trends of just a select few versions for the entire time period would get rid of smaller (but still significant) versions per year, so the full table is included in appendix A. Furthermore, the one artifact compiled in Java 21 is not unique: in the past, there have been other archives compiled in early-access Java versions, though uncommon. On the other end, there are still packages being released in Java versions that are no longer officially supported. This is the case for roughly 21000 packages. Note that this is an estimate; end-of-support dates for old versions are not known, and if known these dates can differ per Java vendor by multiple years. Most older versions contributing to a significant portion of releases have had their usage dropping over time.

It is possible to have multiple different Java class file versions in the same archive, this happens in 16575 (4.04%) archives. Out of these archives, only 700 (4.22%) are multi-release archives. Interestingly, 370 (34.58%) of all multi-release archives do not contain multiple class versions. By

[9] FASTEN project: https://fasten-project.eu/
[10] https://central.sonatype.org/publish/
[11] https://github.com/ashkboos/MavenSecrets/ (release: 1.0.0)
[12] `org.smarti18n:smarti18n-com:1.0-PRE4`
[13] `org.infinispan:infinispan-commons-jdk21:15.0.0.Dev01`

Java Version	Packages
J2SE 1.2	1539 (0.38%)
J2SE 1.3	912 (0.22%)
J2SE 1.4	2154 (0.53%)
Java SE 5	27231 (6.64%)
Java SE 6	56824 (13.86%)
Java SE 7	33612 (8.2%)
Java SE 8	182476 (44.5%)
Java SE 9	1073 (0.26%)
Java SE 10	461 (0.11%)
Java SE 11	22406 (5.46%)
Java SE 12	243 (0.06%)
Java SE 13	196 (0.05%)
Java SE 14	302 (0.07%)
Java SE 15	336 (0.08%)
Java SE 16	390 (0.1%)
Java SE 17	4560 (1.11%)
Java SE 18	59 (0.01%)
Java SE 19	163 (0.04%)
Java SE 20	48 (0.01%)
Java SE 21	1 (0.0%)
Classless	73430 (17.91%)
Unknown*	1686 (0.41%)

Table 3: Java class version distribution. * Older Java versions use the same major class number and cannot be identified.

manual inspection, it was found that there are multiple reasons for archives to contain multiple versions:

1. Third-party sources, which may have been compiled in a different version, can be included in the archive[14].
2. Module files are purposely included even though the rest of the code is compiled in an older version[15].
3. The project contains code for a different JVM language, which ends up being compiled into bytecode for a different Java version[16].

Note that in all these cases, there is no reason to set the multi-release flag, which largely contributes to why so few archives specify it. The multi-release feature is not intended to bundle different class versions, but to allow multiple definitions of the same class with different implementations per Java version. This does not apply here.

The inspected manifest entries specify the *Java Development Kit* (JDK) version used to compile the code. These are specified in 268609 packages (65.5% of archives). Out of the artifacts with multiple manifest entries defined (245653 packages), 65 artifacts have mismatching versions. In many cases, this happens when one of the entries specifies the bytecode version while another specifies the compiler version used to create the code[17]. Note that entries that are not parseable by the rules specified in section 3.3 are ignored (50.9%). In most cases, the most common class file version matches the JDK version used to compile the code (see table 4). In 37.26%

[14] `activesoap:jaxb-xalan:1.5`
[15] `ai.swim:swim-codec:3.10.0`
[16] `ai.h2o:xgboost4j:0.90.3`
[17] `com.liferay:javax.servlet.jsp:2.3.3-b02.LIFERAY-PATCHED-5`

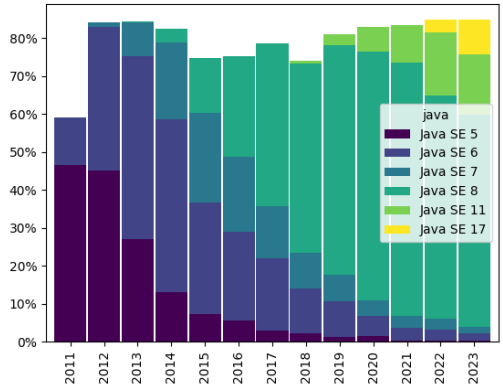

Figure 2: Java class version distribution (LTS versions and 3 other common versions). See appendix A for the full table.

cases, a more recent JDK version is used to compile the code, meaning the developers had the option to compile to a more modern version but chose not to. Curiously, in a few cases, the class files are compiled in a more modern version than the JDK version used to build the project. By manual inspection, it was found that this happens when the modern code is not generated by the JDK version specified in the manifest[18]. Note that for this analysis, only the packages with at least one of the manifest entries matching the grammar described in section 3 are considered (224526 packages).

Comparison Result	Packages
Same	140617 (62.63%)
Newer	83668 (37.26%)
Older	241 (0.11%)

Table 4: JDK version compared to the most common class file version.

How is the compiler configured in the POM? (RQ 3)

In many cases (213106 packages, 45.02%) the Maven compiler plugin is not configured. In 260246 packages it is.

The most common character set used is `UTF-8`, making out 25.0% of the configurations (see table 5). The overwhelming majority, however, does not define the character encoding used (74.5%), instead using the build platform's encoding. This technically makes the build platform-dependent. In 241 cases, the POM contained a placeholder that could not be resolved (i.e. they are of the form `${x}`).

If the Maven compiler plugin is used, both the source and target version are specified 95.89% of the time (249542 packages) and in 10202 cases (3.92%) neither is specified. Interestingly, even though by design it is possible to use different source and release versions, in practically all cases where both are specified, they are equal. A very small portion of artifacts specifies different versions (385 packages, 0.15%). This feature has its purposes but is rarely capitalized on. Only specifying one of the two is rare: only 376 packages (0.14%)

[18] `org.bytedeco:tensorrt-platform:8.2-1.5.7`

Character Encoding	Packages
UTF-8	65058 (25.0%)
ISO-8859-1	997 (0.38%)
ISO-8859-15	65 (0.02%)
WINDOWS-1252	63 (0.02%)
ASCII	27 (0.01%)
GBK	26 (0.01%)
WINDOWS-31J	1 (0.0%)
Undefined*	193760 (74.5%)
Unresolved**	241 (0.09%)

Table 5: The source encoding distribution. *If left undefined, the platform's encoding is used. **Unresolved Maven placeholders.

solely specify the source version and 126 packages (0.05%) specify just the target version. The source and target version distribution is listed in table 6. As expected, they roughly match the Java class version distribution seen earlier, but there are significant deviations.

Java Version	Source	Target
JDK 1.0	1 (0.0%)	6 (0.0%)
JDK 1.1	0 (0.0%)	42 (0.02%)
J2SE 1.2	42 (0.02%)	47 (0.02%)
J2SE 1.3	535 (0.21%)	525 (0.2%)
J2SE 1.4	1855 (0.71%)	1861 (0.72%)
Java SE 5	21833 (8.39%)	21729 (8.35%)
Java SE 6	39568 (15.2%)	39593 (15.21%)
Java SE 7	36347 (13.97%)	36368 (13.97%)
Java SE 8	124923 (48.0%)	124774 (47.94%)
Java SE 9	801 (0.31%)	831 (0.32%)
Java SE 10	264 (0.1%)	264 (0.1%)
Java SE 11	18338 (7.05%)	18269 (7.02%)
Java SE 12	201 (0.08%)	201 (0.08%)
Java SE 13	161 (0.06%)	154 (0.06%)
Java SE 14	199 (0.08%)	200 (0.08%)
Java SE 15	219 (0.08%)	213 (0.08%)
Java SE 16	297 (0.11%)	293 (0.11%)
Java SE 17	4055 (1.56%)	4036 (1.55%)
Java SE 18	53 (0.02%)	36 (0.01%)
Java SE 19	171 (0.07%)	171 (0.07%)
Java SE 20	55 (0.02%)	55 (0.02%)
Undefined	10328 (3.97%)	10578 (4.06%)

Table 6: Source and target Java versions.

Furthermore, only 3141 (1.21%) artifacts deviate from using the standard *Javac* compiler, bundled with every JDK (see table 7). One noteworthy compiler is a *Groovy* compiler, which is another JVM language. The other is `frgaal`, a retrofit compiler, meaning modern Java features can be used in source code, and the compiler makes it runnable on versions predating those features.

Only 32194 configurations specify additional compiler arguments (12.4%). The most common additional flag is linter configuration (`-Xlint`), which enables more verbose feedback from the linter, but additional compiler flags are rare in general (see table 8). `-Xlint` arguments are present in 22629 configurations (70.3%). In fact, the vast majority of extra

Compiler Id	Packages
groovy-eclipse-compiler	1189 (0.46%)
javac-with-errorprone	1173 (0.45%)
eclipse	483 (0.19%)
jdt	255 (0.1%)
javac	214 (0.08%)
tuscany-eclipse	32 (0.01%)
frgaal	9 (0.0%)
Undefined*	256891 (98.71%)

Table 7: The source encoding distribution. *If left undefined, `javac` is used.

flags provide more feedback when compiling. Another noteworthy flag is `-parameters` (7759 packages, 24.1%), which makes the compiler store method and constructor parameter names in the class file(s). Note that the same argument (key) may be specified multiple times in a single package.

Command Line Argument	Occurrences
`-Xlint`	32767
`-Xep`	18498
`-parameters`	7759
`-J`	3136
`-Werror`	1868

Table 8: Compiler flags (top 5). See appendix B for the full table.

6 Discussion

Java modules are uncommon, and even though they have been available to use for 6 years, they have not caught on. Presumably, this is because they are not required for Java development, and as such are often ignored. Another possibility is that similar solutions exist for project segregation, such as Maven modules or multi-project Gradle builds. These solutions predate Java modules. These patterns in feature adoption are not unique to Java modules. Hartveld[8] also showed a comparable pattern for asynchronous programming in *C#*.

With Java modules being so uncommon, one has to wonder what other Java features are rarely adopted. Furthermore, even if features are rarely used, it is worth investigating what kind of projects do adopt these features. As an example, according to the Java specification request JSR 376[19], Java modules are intended for libraries and large applications. It is worthwhile for Java language designers to research whether the target audience for this language feature actually adopts it. Such research is also applicable for library maintainers, as their newly introduced features are in turn adopted by projects depending on these libraries.

The most common Java version used is Java 8, and even though later LTS versions have been adopted too, a large portion still uses years-old versions. In software development, it is not uncommon for codebases to stick to a single version and delay migration for years[9]. Nevertheless, Java 9 and later (apart from LTS version) are barely present, especially compared to non-LTS versions prior to Java 9.

[19]https://openjdk.org/projects/jigsaw/spec/

Perhaps the absence of newer versions is because these versions are still relatively new, but it should be noted that since the release of Java 9, new Java versions have been released twice a year[20]. This faster release cycle may have resulted in Java releases of a way smaller scope than earlier releases. As an example, Java 8 introduced lambdas, which is a large feature that drastically changed the way programmers write Java code. It is also possible that the features included in more recent versions are generally less desirable. Looking at past releases, what kind of features they included is worth looking into. Specifically, Java language designers can use such research to revise the release cycle and to determine how long versions should be supported.

To boot, there are packages released both for unsupported and unreleased Java versions. The former can be blamed on a large presence of systems running legacy software, but the latter stands out. Developing software on pre-release Java builds can have its purpose, but publishing such software on Maven Central is peculiar. Software built on development versions is not guaranteed to be working when the full Java version is released, and since packages on Maven Central cannot be modified or removed once published, this may result in an unusable artifact. Why this software is released is unclear, but worth looking into. Sonatype, the maintainer of the central repository, has set virtually no requirements on the contents of artifacts to be published[21]. This means these kinds of releases are perfectly valid.

The Java compiler does not need to be configured in the majority of cases. If it is, virtually all configurations specify source and target versions. Compiler flags, however, are not present as often. In most cases when flags are defined, they are to help developers improve their software quality. This is done purely because maintaining high code quality is good practice since Maven Central does not have any form of quality control. Investigating the quality of artifacts published on Maven Central can inform both consumers of the repository to help decide what projects to depend on, and repository maintainers, if some kind of submission review should be implemented.

As for source encoding, most source code does not have any special characters, just the Latin alphabet, digits, control characters, and characters like `!@#$%^&*()`. As a result, `ASCII` can be used in most cases. This implicitly means that more modern character encoding like Unicode transformation formats, Windows code pages, and the `ISO 8859` family are all valid as they extend from `ASCII`. In most cases, developers, do not need to worry about the source encoding even if it makes their build platform-dependent, likely the source will compile.

Limitations

Unfortunately, the approach is not without its limitations, besides the Maven Central index being incomplete.

Firstly, Maven build cycles are inherently version-dependent. For example, if a plugin is configured, but no plugin version is specified, Maven tries to find the latest version. This version depends on the Maven version used to build the project. Maven `3.x` just considers `RELEASE` versions while Maven `2.x` also considers `SNAPSHOT` versions. Furthermore, depending on plugin versions, there may be different parameters, defaults, and properties used to configure these parameters. Additionally, environment variables play a role and these are not reproducible.

Secondly, Maven builds can both generate sources to include and include third-party sources in the final artifact. Since it is the final artifact that is analyzed, these added sources are not distinguished from the actual source.

Thirdly, the configuration is complex. Different executions, phases, goals, profiles, and inheritance makes predicting the output difficult. All projects implicitly inherit from a *super-POM*, which is also version-dependent. For the purposes of this research, only the root configurations of the `plugin`, `pluginManagement` and their parents are considered.

Fourthly, non-ZIP artifacts, third-party plugins, and non-Java artifacts skew the results, however, these are not common as Maven is primarily used for Java programming (and other languages that compile to Java bytecode).

Lastly, projects hosted in Maven repositories are not required to be built by Maven. These projects are still required to upload a POM containing some information like group id, artifact id, version. This means that there is no way to tell if the POM was actually used to create the artifact hosted in the repository.

7 Conclusion

In conclusion, the analysis of Maven Central projects sheds light on the Java software ecosystem. Based on the analysis of projects hosted on Maven Central, these are the key findings regarding the usage of Java modules, popular Java versions, and the configuration of the compiler.

For the analysis, one random version for each project hosted on Maven Central was selected. This artifact was then downloaded followed by extracting relevant data. The intermediate data was then processed further to produce the results described.

Firstly, it was observed that Java modules are rarely used, accounting for less than 2% of the projects examined. The adoption of Java modules has been minimal. Interestingly, it was discovered that there are projects compiled in older Java versions that still use Java modules. This is done by generating the module-info files during the build process.

Secondly, Java SE 8 is the most widely utilized Java version. Furthermore, it was found that long-term support versions make up the majority of releases ($>50\%$). This implies a preference for stable and well-supported Java versions. Out of the other versions, the older versions (Java SE 7 and below) are used significantly more than the newer versions. Furthermore, there are projects released using early-access Java builds and Java versions that are no longer officially supported.

Lastly, looking into the configuration of the Maven compiler plugin revealed that it is not used in roughly half of the cases (45%). However, when the plugin is configured, it is usually to specify the Java version to compile the code with.

[20] https://java.com/releases/

[21] https://central.sonatype.org/publish/requirements/

Additionally, the extra specified command line flags for the compiler suggest a focus on code quality and standards.

References

[1] R. Kula, D. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest maven release," 03 2015.

[2] R. G. Kula, D. M. German, T. Ishio, A. Ouni, and K. Inoue, "An exploratory study on library aging by monitoring client usage in a software ecosystem," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 407–411, Feb 2017.

[3] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: An evolutionary study," *Empirical Softw. Engg.*, vol. 20, p. 1275–1317, oct 2015.

[4] J. Düsing and B. Hermann, "Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories," *Digital Threats*, vol. 3, feb 2022.

[5] C. Zhang, B. Chen, J. Hu, X. Peng, and W. Zhao, "Buildsonic: Detecting and repairing performance-related configuration smells for continuous integration builds," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, (New York, NY, USA), Association for Computing Machinery, 2023.

[6] F. Dekking, *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer Texts in Statistics, Springer, 2005.

[7] V. L. Parsons, *Stratified Sampling*, pp. 1–11. John Wiley & Sons, Ltd, 2017.

[8] D. Hartveld, "An empirical evaluation of and toolkit for asynchronous programming in c# windows phone apps," Master's thesis, Delft University of Technology, August 2014.

[9] P. Capek, E. Kral, and R. Senkerik, "Towards an empirical analysis of .net framework and c# language features' adoption," in *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 865–866, Dec 2015.

A Java Version Distribution

	2011	2012	2013	2014	2015	2016	2017
J2SE 1.2	1176 (5.96%)	69 (0.6%)	43 (0.34%)	38 (0.22%)	67 (0.27%)	41 (0.13%)	28 (0.08%)
J2SE 1.3	721 (3.66%)	37 (0.32%)	30 (0.23%)	22 (0.13%)	31 (0.13%)	24 (0.08%)	9 (0.03%)
J2SE 1.4	1659 (8.41%)	151 (1.31%)	76 (0.59%)	51 (0.3%)	28 (0.11%)	28 (0.09%)	31 (0.09%)
Java SE 5	9175 (46.52%)	5227 (45.18%)	3452 (27.01%)	2193 (12.97%)	1789 (7.24%)	1755 (5.66%)	1037 (2.94%)
Java SE 6	2475 (12.55%)	4364 (37.72%)	6167 (48.25%)	7708 (45.58%)	7251 (29.36%)	7248 (23.36%)	6727 (19.05%)
Java SE 7	11 (0.06%)	133 (1.15%)	1149 (8.99%)	3436 (20.32%)	5870 (23.77%)	6107 (19.69%)	4855 (13.75%)
Java SE 8	-	-	10 (0.08%)	599 (3.54%)	3538 (14.33%)	8261 (26.63%)	15137 (42.87%)
Java SE 9	-	-	-	-	-	-	61 (0.17%)
Classless	3196 (16.2%)	1514 (13.09%)	1820 (14.24%)	2829 (16.73%)	6087 (24.65%)	7469 (24.08%)	7406 (20.98%)
Unknown*	1310 (6.64%)	75 (0.65%)	35 (0.27%)	35 (0.21%)	36 (0.15%)	88 (0.28%)	17 (0.05%)

Table 9: The Java class version distribution for packages released in 2011-2017. * Older Java versions use the same major class number and cannot be identified.

	2018	2019	2020	2021	2022	2023
J2SE 1.2	23 (0.06%)	13 (0.03%)	21 (0.05%)	9 (0.02%)	6 (0.01%)	5 (0.02%)
J2SE 1.3	9 (0.02%)	10 (0.02%)	8 (0.02%)	4 (0.01%)	5 (0.01%)	2 (0.01%)
J2SE 1.4	44 (0.11%)	20 (0.04%)	31 (0.07%)	19 (0.04%)	12 (0.02%)	4 (0.01%)
Java SE 5	863 (2.21%)	582 (1.28%)	689 (1.58%)	196 (0.4%)	191 (0.38%)	82 (0.27%)
Java SE 6	4660 (11.92%)	4272 (9.42%)	2299 (5.27%)	1661 (3.35%)	1415 (2.82%)	577 (1.91%)
Java SE 7	3664 (9.37%)	3118 (6.88%)	1764 (4.04%)	1539 (3.11%)	1430 (2.85%)	536 (1.77%)
Java SE 8	19508 (49.88%)	27449 (60.53%)	28567 (65.46%)	33062 (66.71%)	29446 (58.7%)	16899 (55.83%)
Java SE 9	207 (0.53%)	199 (0.44%)	171 (0.39%)	269 (0.54%)	121 (0.24%)	45 (0.15%)
Java SE 10	253 (0.65%)	97 (0.21%)	46 (0.11%)	43 (0.09%)	17 (0.03%)	5 (0.02%)
Java SE 11	241 (0.62%)	1319 (2.91%)	2873 (6.58%)	4833 (9.75%)	8356 (16.66%)	4784 (15.81%)
Java SE 12	-	98 (0.22%)	110 (0.25%)	30 (0.06%)	2 (0.0%)	3 (0.01%)
Java SE 13	-	10 (0.02%)	145 (0.33%)	33 (0.07%)	8 (0.02%)	0 (0.0%)
Java SE 14	-	-	149 (0.34%)	94 (0.19%)	45 (0.09%)	14 (0.05%)
Java SE 15	-	-	59 (0.14%)	121 (0.24%)	127 (0.25%)	29 (0.1%)
Java SE 16	-	-	2 (0.0%)	185 (0.37%)	112 (0.22%)	91 (0.3%)
Java SE 17	-	-	-	78 (0.16%)	1690 (3.37%)	2792 (9.22%)
Java SE 18	-	-	-	4 (0.01%)	46 (0.09%)	9 (0.03%)
Java SE 19	-	-	-	-	50 (0.1%)	113 (0.37%)
Java SE 20	-	-	-	-	-	48 (0.16%)
Java SE 21	-	-	-	-	-	1 (0.0%)
Classless	9598 (24.54%)	8154 (17.98%)	6676 (15.3%)	7374 (14.88%)	7084 (14.12%)	4223 (13.95%)
Unknown*	39 (0.1%)	8 (0.02%)	29 (0.07%)	7 (0.01%)	3 (0.01%)	4 (0.01%)

Table 10: The Java class version distribution for packages released in 2018-2023. * Older Java versions use the same major class number and cannot be identified.

B Maven Compiler Plugin: Compiler Flags

Command Line Argument	Occurrences
-Xlint	32767
-Xep	18498
-parameters	7759
-J	3136
-Werror	1868
-Xpkginfo	1709
-A	1114
-err	682
-XDignore.symbol.file	657
-Xplugin	581
-XDcompilePolicy	567
-proc	473
--add-exports	407
-XepExcludedPaths	267
-bootclasspath	247
-Xdoclint	208
-verbose	178
-warn	162
--enable-preview	124
-Xmaxwarns	121
-Xdiags	114
-h	112
-XepDisableWarningsInGeneratedCode	95
-implicit	87
-Xmaxerrs	65
-g	54
-XepOpt	51
--module-version	50
--add-modules	41
-XepAllErrorsAsWarnings	32
-XDenableSunApiLintControl	17
-profile	17
-Xbootclasspath/p	11
--add-opens	9
-extdirs	9
-XepAllDisabledChecksAsWarnings	9
-XepIgnoreUnknownCheckNames	6
-cp	5
-XepDisableAllChecks	4
-processor	4
--patch-module	4
-XX	4
-XprintProcessorInfo	3
-XprintRounds	3
-sourcepath	2
-deprecation	2
--add-reads	2
-XoutputDirectory	1
-O	1
-add-opens	1
-Xbootclasspath	1

Table 11: Compiler flags.