Apache Log4j Audit

v. 1.0.0-SNAPSHOT

Project Documentation

Apache Software Foundation **2018-02-11**

Table Of Content

1 Getting Started

1.1 Getting Started with Log4j Audit

This guide provides an overview of how to define events to be audited, generate the Java interfaces for those events and then use those interfaces to generate the audit events.

1.1.1 What you will build

You will build a project that consist of two modules. One module generates a jar that contains the audit catalog along with the Java interfaces that were created from the catalog. The second module generates a war that provides the service endpoints to perform remote audit logging and manage dynamic catalogs. You will install and use the catalog editor. Finally, you will also build a project that uses the audit event interfaces and generates audit events.

1.1.2 What you will need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Apache Maven 3.0+
- Apache Tomcat 7.0+ (Any servlet 2.5+ compatible container will work)

1.1.3 How to complete this guide

Create a directory for this guide:

```
mkdir log4j-audit-guide
cd log4j-audit-guide
```

Download and unzip the sample source repository, or clone it using Git:

```
git clone https://github.com/apache/logging-log4j-audit-sample
```

Change to the root directory of the project and build it using Maven:

```
cd logging-log4j-audit-sample
mvn clean install
```

Two artifacts will have been created and installed into your local Maven repository:

1. org.apache.logging.log4j:audit-service-api:1.0.0-SNAPSHOT:jar
2. org.apache.logging.log4j:audit-service-war:1.0.0-SNAPSHOT:war

The sample catalog can be found at audit-service-api/src/main/resources/catalog.json.

1.1.4 Inspect the build results

List the contents of audit-service-api/target/generated-sources/log4j-audit directory. The event interfaces generated from the catalog will be located in this directory.

1.1.5 Run an application that performs auditing

1. Change to the sample-app diretory.

```
cd sample-app
```

2. Run the sample app and view the logs

```
./sample-app.sh
vi target/logs/audit.log
```

1.1.6 Deploy the Audit Service WAR

1. Download Apache Tomcat from the Tomcat website. and install Apache Tomcat. The simplest way to install Tomcat consists of just unzipping the Tomcat distribution into the appropriate directory. For more complete instructions follow the Tomcat setup page.

```
cd ..
unzip ~/Downloads/apache-tomcat-7.0.82.zip
```

2. Copy the war file generated by the pervious build into Tomcat's webapp directory:

```
cp logging-log4j-audit-sample/audit-service-war/target/AuditService.war apache-tomcat-7.0.82/web
```

1.1.7 Deploy the Audit Catalog WAR

1. Download the Log4j audit binary zip.

```
wget http://www.apache.org/dist/logging/log4j-audit/${Log4jAuditVersion}/apache-log4j-audit-${Lo
```

2. Unzip the contents.

```
unzip apache-log4j-audit-${Log4jAuditVersion}-bin.zip
```

3. Copy the Log4j Catalog Editor war to the tomcat webapps directory.

```
cp apache-log4j-audit-${Log4jAuditVersion}-bin/log4j-catalog-war-${Log4jAuditVersion}.war apache
```

4. Use an editor to create a file named catalog-config.properties in Tomcat's classpath. You can create the file in the apache-tomcat-7.0.82/lib directory.
5. Copy the following lines into the file. The value for remoteRepoUrl should the Git repo where your version of catalog.json should be stored. remoteRepoCatalogPath is the location within that Git repository where the catalog.json file resides. gitPassPhrase is the pass phrase needed to access the repository when SSH is used. gitUserName and gitPassPhrase are the credentials required to access the Git repository when using HTTP or HTTPS. If the credentials or pass phrase are not provided typically you will be able to view the catalog but not update it.

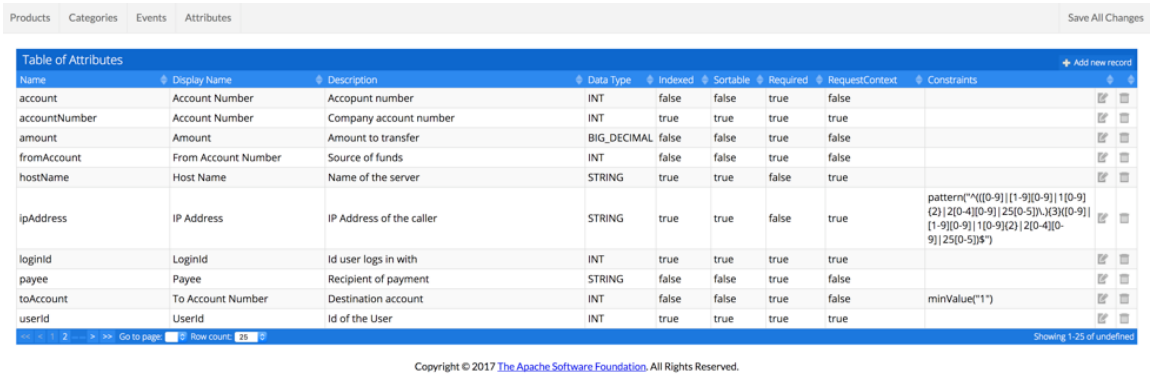
```
remoteRepoUrl=https://git-wip-us.apache.org/repos/asf/logging-log4j-audit-sample.git
remoteRepoCatalogPath=audit-service-api/src/main/resources/catalog.json
gitUserName=
gitPassword=
gitPassPhrase=
```

6. Start Tomcat.

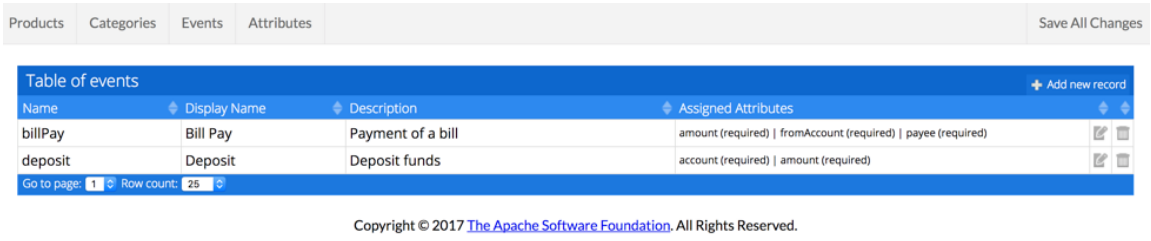
```
cd apache-tomcat-7.0.82/bin
./startup.sh
```

1.1.8 Use the Catalog Editor

1. Navigate to the edit attributes screen at http://localhost:8080/AuditCatalog/attributes. The screen should look like



2. Navigate to the edit events screen at http://localhost:8080/AuditCatalog/events. The screen should look like



2 Guidelines

2.1 Apache Log4j Project Guidelines

This document defines the guidelines for the Apache Log4j Project. It includes definitions of how conflict is resolved by voting, who is able to vote, the procedures to follow for proposing and making changes as well as guidelines for changing code.

The objective here is to avoid unnecessary conflict over changes and continue to produce a quality system in a timely manner. Not all conflict can be avoided, but at least we can agree on the procedures for conflict to be resolved.

2.1.1 People, Places, and Things

Apache Logging Project Management Committee

The group of volunteers who are responsible for managing the Apache Logging Projects, including Log4j. This includes deciding what is distributed as products of the Apache Logging Project, maintaining the Project's shared resources, speaking on behalf of the Project, resolving license disputes regarding Apache products, nominating new PMC members or committers, and establishing these guidelines.

Membership in the Apache Logging PMC is by invitation only and must be approved by consensus of the active Logging PMC members. A PMC member is considered inactive by their own declaration or by not contributing in any form to the project for over six months. An inactive member can become active again by reversing whichever condition made them inactive (*i.e.* , by reversing their earlier declaration or by once again contributing toward the project's work). Membership can be revoked by a unanimous vote of all the active PMC members other than the member in question.

Apache Logging Committers

The group of volunteers who are responsible for the technical aspects of the Apache Logging Projects. This group has write access to the appropriate source repositories and these volunteers may cast binding votes on any technical discussion. Although a committer usually joins due to their activity on one of the Logging projects, they will have commit access to all Logging projects.

Membership as a Committer is by invitation only and must be approved by consensus of the active Logging PMC members. A Committer is considered inactive by their own declaration or by not contributing in any form to the project for over six months. An inactive member can become active again by reversing whichever condition made them inactive (*i.e.* , by reversing their earlier declaration or by once again contributing toward the project's work). Membership can be revoked by a unanimous vote of all the active PMC members (except the member in question if they are a PMC member).

Log4j Developers

All of the volunteers who are contributing time, code, documentation, or resources to the Log4j Project. A developer that makes sustained, welcome contributions to the project for over six months is usually invited to become a Committer, though the exact timing of such invitations depends on many factors.

mailing list

The Log4j developers' primary mailing list for discussion of issues and changes related to the project (*dev@logging.apache.org*). Subscription to the list is open, but only subscribers can post directly to the list.

private list

The Logging PMC's private mailing list for discussion of issues that are inappropriate for public discussion, such as legal, personal, or security issues prior to a published fix. Subscription to the list is only open (actually: mandatory) to Apache Logging's Project Management Committee.

Git

All of the Apache products are maintained in information repositories using either Subversion or Git; Log4j uses Git. Only some of the Apache developers have write access to the Apache Logging repositories; everyone has read access.

2.1.2 Issue Management

The Log4j project uses the Jira bug tracking system hosted and maintained by the Apache Software Foundation for tracking bugs and enhancements. The project roadmap may be maintained in JIRA through its RoadMap feature and through the use of Story or Epic issues.

Many issues will be encountered by the project, each resulting in zero or more proposed action items. Issues should be raised on the mailing list as soon as they are identified. Action items **must** be raised on the mailing list and added to JIRA using the appropriate issue type. All action items may be voted on, but not all of them will require a formal vote.

2.1.3 Voting

Any of the Log4j Developers may vote on any issue or action item. However, the only binding votes are those cast by active members of the Apache Logging PMC; if the vote is about a change to source code or documentation, the primary author of what is being changed may also cast a binding vote on that issue. All other votes are non-binding. All developers are encouraged to participate in decisions, but the decision itself is made by those who have been long-time contributors to the project. In other words, the Apache Log4j Project is a minimum-threshold meritocracy.

The act of voting carries certain obligations -- voting members are not only stating their opinion, they are agreeing to help do the work of the Log4j Project. Since we are all volunteers, members often become inactive for periods of time in order to take care of their "real jobs" or devote more time to other projects. It is therefore unlikely that the entire group membership will vote on every issue. To account for this, all voting decisions are based on a minimum quorum.

Each vote can be made in one of three flavors:

+1

Yes, agree, or the action should be performed. On some issues, this vote is only binding if the voter has tested the action on their own system(s).

±0

Abstain, no opinion, or I am happy to let the other group members decide this issue. An abstention may have detrimental effects if too many people abstain.

-1

No. On issues where consensus is required, this vote counts as a **veto**. All vetoes must include an explanation of why the veto is appropriate. A veto with no explanation is void. No veto can be overruled. If you disagree with the veto, you should lobby the person who cast the veto. Voters intending to veto an action item should make their opinions known to the group immediately, so that the problem can be remedied as early as possible.

An action item requiring *consensus approval* must receive at least **3 binding +1** votes and **no vetoes**. An action item requiring *majority approval* must receive at least **3 binding +1** votes and more **+1** votes than **-1** votes (*i.e.* , a majority with a minimum quorum of three positive votes). All other action items are considered to have *lazy approval* until someone votes **-1** , after which point they are decided by either consensus or a majority vote, depending upon the type of action item.

When appropriate, votes should be tallied in the JIRA issue. All votes must be either sent to the mailing list or added directly to the JIRA issue.

2.1.4 Types of Action Items

Long Term Plans

Long term plans are simply announcements that group members are working on particular issues related to the Log4j software. These are not voted on, but group members who do not agree with a particular plan, or think an alternate plan would be better, are obligated to inform the group of their feelings. In general, it is always better to hear about alternate plans **prior** to spending time on less adequate solutions.

Short Term Plans

Short term plans are announcements that a developer is working on a particular set of documentation or code files, with the implication that other developers should avoid them or try to coordinate their changes. This is a good way to proactively avoid conflict and possible duplication of work.

Release Plan

A release plan is used to keep all the developers aware of when a release is desired, who will be the release manager, when the repository will be frozen in order to create the release, and assorted other trivia to keep us from tripping over ourselves during the final moments. Lazy majority (at least 3 x +1 and more +1 than -1) decides each issue in the release plan.

Release Testing

After a new release is built it must be tested before being released to the public. Majority approval is required before the distribution can be publicly released.

Showstoppers/Blockers

Showstoppers are issues that require a fix be in place before the next public release. They are listed in Jira in order to focus special attention on the problem. An issue becomes a showstopper when it is listed as such in Jira and remains so by lazy consensus.

All product changes to the currently active repository are subject to lazy consensus. All product changes to a prior-branch (old version) repository require consensus before the change is committed.

2.1.5 When to Commit a Change

Ideas must be review-then-commit; patches can be commit-then-review. With a commit-then-review process, we trust that the developer doing the commit has a high degree of confidence in the change. Doubtful changes, new features, and large-scale overhauls need to be discussed before being committed to a repository. Any change that affects the semantics of arguments to configurable directives, significantly adds to the runtime size of the program, or changes the semantics of an existing API function must receive consensus approval on the mailing list before being committed.

Each developer is responsible for notifying the mailing list and adding an action item to Jira when they have an idea for a new feature or major change to propose for the product. The distributed nature of the Log4j project requires an advance notice of 48 hours in order to properly review a major change -- consensus approval of either the concept or a specific patch is required before the change can be committed. Note that a member might veto the concept (with an adequate explanation), but later rescind that veto if a specific patch satisfies their objections. No advance notice is required to commit singular bug fixes.

Related changes should be committed as a group, or very closely together. Half-completed projects should not be committed unless doing so is necessary to pass the baton to another developer who has agreed to complete the project in short order. All code changes must be successfully compiled and unit tests pass on the developer's platform before being committed.

The current source code tree should be capable of complete compilation at all times. However, it is sometimes impossible for a developer on one platform to avoid breaking some other platform when a change is committed, particularly when completing the change requires access to a special development tool on that other platform. If it is anticipated that a given change will break some other platform, the committer must indicate that in the commit log.

The committer is responsible for the quality of any third-party code or documentation they commit to the repository. All software committed to the repository must be covered by the Apache LICENSE or contain a copyright and license that allows redistribution under the same conditions as the Apache LICENSE.

A committed change must be reversed if it is vetoed by one of the voting members and the veto conditions cannot be immediately satisfied by the equivalent of a "bug fix" commit. The veto must be rescinded before the change can be included in any public release.

2.1.6 changes.xml and Git logs

Many code changes should be noted in the changes.xml file, and all should be documented in Git commit messages. Often the text of the Git log and the changes.xml entry are the same, but the distinct requirements sometimes result in different information.

2.1.7 Git log

The Git commit log message contains any information needed by

- fellow developers or other people researching source code changes/fixes
- end users (at least point out what the implications are for end users; it doesn't have to be in the most user friendly wording)

If the code change was provided by a non-committer, attribute it using Submitted-by. If the change was committed verbatim, identify the committer(s) who reviewed it with Reviewed-by. If the change was committed with modifications, use the appropriate wording to document that, perhaps "committed with changes" if the person making the commit made the changes, or "committed with contributions from xxxx" if others made contributions to the code committed.

Example log message:

```
LOG4J2-9999
Check the return code from parsing the content length, to avoid a
crash if requests contain an invalid content length.
Submitted by: Jane Doe <janedoe example.com>
Reviewed by: susiecommitter
```

2.1.8 changes.xml

changes.xml is the subset of the information that end users need to see when they upgrade from one release to the next:

- what can I now do that I couldn't do before
- what problems that we anticipate a user could have suffered from are now fixed
- all security fixes included, with CVE number. (If not available at the time of the commit, add later.)

All entries in changes.xml should include the appropriate Jira issue number and should credit contributions made by non-committers by referencing them in the due-to attribute even if modifications needed to be made to the contribution.

The attribution for the change is anyone responsible for the code changes.

2.1.9 Committing Security Fixes

Open source projects, ASF or otherwise, have varying procedures for commits of vulnerability fixes. One important aspect of these procedures is whether or not fixes to vulnerabilities can be committed to a repository with commit logs and possibly CHANGES entries which purposefully obscure the vulnerability and omit any available vulnerability tracking information. The Apache HTTP Server project has decided that it is in the best interest of our users that the initial commit of such code changes to any branch will provide the best description available at that time as well as any available tracking information such as CVE number. Committing of the fix will be delayed until the project determines that all of the information about the issue can be shared.

In some cases there are very real benefits to sharing code early even if full information about the issue cannot, including the potential for broader review, testing, and distribution of the fix. This is outweighed by the concern that sharing only the code changes allows skilled analysts to determine the impact and exploit mechanisms but does not allow the general user community to determine if preventative measures should be taken.

If a vulnerability is partially disclosed by committing a fix before the bug is determined to be exploitable, the httpd security team will decide on a case by case basis when to document the security implications and tracking number.

2.1.10 Patch Format

When a specific change to the software is proposed for discussion or voting on the mailing list, it should be presented in the form of input to the patch command. When sent to the mailing list, the message should contain a Subject beginning with `[PATCH]` and a distinctive one-line summary corresponding to the action item for that patch. Afterwords, the patch summary in the STATUS file should be updated to point to the Message-ID of that message.

The patch should be created by using the diff -u command from the original software file(s) to the modified software file(s). E.g., `diff -u http_main.c.orig http_main.c >> patchfile.txt` or `svn diff http_main.c >> patchfile.txt` All patches necessary to address an action item should be concatenated within a single patch message. If later modification of the patch proves necessary, the entire new patch should be posted and not just the difference between two patches. The STATUS file entry should then be updated to point to the new patch message.

The completed patchfile should produce no errors or prompts when the command, `patch -s < patchfile` is issued in the target repository.

2.1.11 Teamwork

Open source projects function best when everyone is aware of the "rules of the road" and abide by them.

1. Error on the side of caution. If you don't understand it, don't touch it and ask on the list. If you think you understand it read it again or ask until you are sure you do. Nobody will blame you for asking questions.
2. Don't break the build - if there is the slightest chance the change you are making could cause unit test failures, run all unit tests. Better yet, get in the habit of always running the unit tests before doing the commit.
3. If the build breaks and you have made recent changes then assume you broke it and try to fix it. Although it might not have been something you did it will make others feel a lot better than having to fix the mistake for you. Everyone makes mistakes. Taking responsibility for them is a good thing.
4. Don't change things to match your personal preference - the project has style guidelines that are validated with checkstyle, PMD, and other tools. If you aren't fixing a bug, fixing a problem identified by the tools, or fixing something specifically called out in these guidelines then start a discussion to see if the change is something the project wants before starting to work on it. We try to discuss things first and then implement the consensus reached in the discussion.
5. Along the same lines, do not commit automatic changes made by your IDE without reviewing them. There are a few places in the code that cannot conform to style guidelines without causing errors in some environments. These are clearly marked and must be left as is.

3 Style Guide

3.1 Apache Log4j Code Style Guidelines

3.1.1 Introduction

This document serves as the **complete** definition of the Log4j project's coding standards for source code in the Java™ Programming Language. It originated from the Google coding standards but incorporates modifications that reflect the desires of the Log4j community.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the **hard-and-fast rules** that we follow universally, and avoids giving *advice* that isn't clearly enforceable (whether by human or tool).

3.1.2 Terminology notes

In this document, unless otherwise clarified:

1. The term *class* is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (`@interface`).
2. The term *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

Other "terminology notes" will appear occasionally throughout the document.

3.1.3 Guide notes

Example code in this document is **non-normative**. That is, while the examples are in Log4j Style, they may not illustrate the *only* stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

3.1.4 Source File Basics

3.1.5 File name

The source file name consists of the case-sensitive name of the top-level class it contains, plus the `.java` extension.

3.1.6 2.2 File encoding: UTF-8

Source files are encoded in **UTF-8**.

3.1.7 Special characters

3.1.7.1 Whitespace characters

Aside from the line terminator sequence, the **ASCII horizontal space character** (**0x20**) is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped.
2. Tab characters are **not** used for indentation.

3.1.7.2 Special escape sequences

For any character that has a special escape sequence (`\b`, `\t`, `\n`, `\f`, `\r`, `\"`, `\'` and `\\`), that sequence is used rather than the corresponding octal (e.g. `\012`) or Unicode (e.g. `\u000a`) escape.

3.1.7.3 Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g. ∞) or the equivalent Unicode escape (e.g. `\u221e`) is used, depending only on which makes the code **easier to read and understand**.

Tip: In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

Examples:

Example	Discussion
`String unitAbbrev = "μs";`	Best: perfectly clear even without a comment.
`String unitAbbrev = "\u03bcs"; // "μs"`	Allowed, but there's no reason to do this.
`String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"`	Allowed, but awkward and prone to mistakes.
`String unitAbbrev = "\u03bcs";`	Poor: the reader has no idea what this is.
`return '\ufeff' + content; // byte order mark`	Good: use escapes for non-printable characters, and comment if necessary.

Tip: Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are **broken** and they must be **fixed**.

3.1.8 Source file structure

A source file consists of, **in order**:

1. Apache license
2. Package statement
3. Import statements
4. Exactly one top-level class

Exactly one blank line separates each section that is present.

3.1.9 Apache License

The Apache license belongs here. No other license should appear. Other licenses that apply should be referenced in a NOTICE file

3.1.10 Package statement

The package statement is **not line-wrapped**. The column limit (Column limit: 120) does not apply to package statements.

3.1.11 Import statements

3.1.11.1 No wildcard imports in the main tree

Wildcard imports, static or otherwise, **are not used**.

3.1.11.2 Static wildcard imports in the test tree

Wildcard static imports are encouraged for test imports like JUnit, EasyMock, and Hamcrest.

3.1.11.3 No line-wrapping

Import statements are **not line-wrapped**. The column limit (Column limit: 120) does not apply to import statements.

3.1.11.4 Ordering and spacing

Import statements are divided into the following groups, in this order, with each group separated by a single blank line:

1. java
2. javax
3. org
4. com
5. All static imports in a single group

Within a group there are no blank lines, and the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order; the presence of semicolons warps the result.)

IDE settings for ordering imports automatically can be found in the source distributions under `src/ide`. For example:

- Eclipse: `src/ide/eclipse/4.3.2/organize-imports.importorder`
- IntelliJ: `src/ide/Intellij/13/IntellijSettings.jar`

3.1.12 Class declaration

3.1.12.1 Exactly one top-level class declaration

Each top-level class resides in a source file of its own.

3.1.12.2 Class member ordering

Class members should be grouped in the following order>.

1. static variables grouped in the order shown below. Within a group variables may appear in any order.
2. 1. public
   2. protected
   3. package
   4. private
3. instance variables grouped in the order shown below. Within a group variables may appear in any order
4. 1. public
   2. protected
   3. package
   4. private
5. constructors
6. methods may be specified in the following order but may appear in another order if it improves the clarity of the program.
7. 1. public
   2. protected
   3. package
   4. private

3.Overloads: never split

When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no intervening members.

3.1.13 Formatting

Terminology Note: *block-like construct* refers to the body of a class, method or constructor. Note that, by array initializers, any array initializer *may* optionally be treated as if it were a block-like construct.

3.1.14 Braces

3.1.14.1 Braces are used where optional

Braces are used with `if`, `else`, `for`, `do` and `while` statements, even when the body is empty or contains only a single statement.

3.1.14.2 Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style (" Egyptian brackets") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace *if* that brace terminates a statement or the body of a method, constructor or *named* class. For example, there is *no* line break after the brace if it is followed by `else` or a comma.

Example:

```
return new MyClass() {
    @Override public void method() {
        if (condition()) {
            try {
                something();
            } catch (ProblemException e) {
                recover();
            }
        }
    }
};
```

A few exceptions for enum classes are given in Section 4.8.1, Enum classes.

3.1.14.3 Empty blocks: may be concise

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters or line break in between (`{}`), **unless** it is part of a *multi-block statement* (one that directly contains multiple blocks: `if/else-if/else` or `try/catch/finally`).

Example:

```
void doNothing() {}
```

3.1.15 Block indentation: +4 spaces

Each time a new block or block-like construct is opened, the indent increases by four spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in Section 4.1.2, Nonempty blocks: K & R Style.)

3.1.16 One statement per line

Each statement is followed by a line-break.

3.1.17 Column limit: 120

The column limit for Log4j is 120 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in Line-wrapping.

Exceptions:

1. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).
2. `package` and `import` statements (see Package statement and Import statements).
3. Command lines in a comment that may be cut-and-pasted into a shell.

3.1.18 Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, typically to avoid overflowing the column limit, this activity is called *line-wrapping*.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

Tip: Extracting a method or local variable may solve the problem without the need to line-wrap.

3.1.18.1 Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

1. When a line is broken at a *non-assignment* operator the break comes *before* the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
    2. This also applies to the following "operator-like" symbols: the dot separator (`.`), the ampersand in type bounds (`<T extends Foo & Bar>`), and the pipe in catch blocks (`catch (FooException | BarException e)`).
3. When a line is broken at an *assignment* operator the break typically comes *after* the symbol, but either way is acceptable.
    4. This also applies to the "assignment-operator-like" colon in an enhanced `for` ("foreach") statement.
5. A method or constructor name stays attached to the open parenthesis (`(`) that follows it.
6. A comma (`,`) stays attached to the token that precedes it.

3.1.18.2 Indent continuation lines at least +8 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +8 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +8 as desired.
In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

The section on Horizontal alignment addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

3.1.19 Whitespace

3.1.19.1 Vertical Whitespace

A single blank line appears:

1. *Between* consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.
2. **Exception:** A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.
3. Within method bodies, as needed to create *logical groupings* of statements.
4. *Optionally* before the first member or after the last member of the class (neither encouraged nor discouraged).
5. As required by other sections of this document (such as Import statements).

Multiple consecutive blank lines are permitted, but never required (or encouraged).

3.1.19.2 Horizontal whitespace

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only**.

1. Separating any reserved word, such as `if`, `for` or `catch`, from an open parenthesis (`(`) that follows it on that line
2. Separating any reserved word, such as `else` or `catch`, from a closing curly brace (`}`) that precedes it on that line
3. Before any open curly brace (`{`), with two exceptions:
4. `String[][] x = {{"foo"}};` (no space is required between `{{`, by item 8 below)
5. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:
6. the ampersand in a conjunctive type bound: `<T extends Foo & Bar>`
7. the pipe for a catch block that handles multiple exceptions: `catch (FooException | BarException e)`
8. the colon (`:`) in an enhanced `for` ("foreach") statement
9. After `,` `:` `;` or the closing parenthesis (`)`) of a cast
10. On both sides of the double slash (`//`) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
11. Between the type and variable of a declaration: `List<String> list`
12. *Optional* just inside both braces of an array initializer
13. `new int[] {5, 6}` and `new int[] { 5, 6 }` are both valid

Note: This rule never requires or forbids additional space at the start or end of a line, only *interior* space.

3.1.19.3 Horizontal alignment: never required

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required** by Google Style. It is not even required to *maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment:

```
private int x; // this is fine
private Color color; // this too

private int   x;      // permitted, but future edits
private Color color;  // may leave it unaligned
```

Tip: Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformattings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

3.1.20 Grouping parentheses: recommended

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is *not* reasonable to assume that every reader has the entire Java operator precedence table memorized.

3.1.21 Specific constructs

3.1.21.1 Enum classes

After each comma that follows an enum constant, a line-break is optional.

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see array initializers).

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

Since enum classes *are classes*, all other rules for formatting classes apply.

3.1.21.2 Variable declarations

3.One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

3.Declared when needed, initialized as soon as possible

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

3.1.21.3 Arrays

3.Array initializers: can be "block-like"

Any array initializer may *optionally* be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

```
new int[] {           new int[] {
  0, 1, 2, 3            0,
}                       1,
                        2,
new int[] {             3,
  0, 1,                 }
  2, 3
}                       new int[]
                            {0, 1, 2, 3}
```

3.No C-style array declarations

The square brackets form a part of the *type*, not the variable: `String[] args`, not `String args[]`.

3.1.21.4 Switch statements

Terminology Note: Inside the braces of a *switch block* are one or more *statement groups*. Each statement group consists of one or more *switch labels* (either `case FOO:` or `default:`), followed by one or more statements.

3.Indentation

As with any other block, the contents of a switch block are indented +2.

After a switch label, a newline appears, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

3.Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a `break`, `continue`, `return` or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block. Example:

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
    // fall through
  case 3:
    handleOneTwoOrThree();
    break;
default:
    handleLargeNumber(input);
}
```

3.The default case is present

Each switch statement includes a `default` statement group, even if it contains no code.

3.1.21.5 Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section 4.5, Line-wrapping), so the indentation level is not increased. Example:

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

Exception: A *single* parameterless annotation *may* instead appear together with the first line of the signature, for example:

```
@Override public int hashCode() { ... }
```

Annotations applying to a field also appear immediately after the documentation block, but in this case, *multiple* annotations (possibly parameterized) may be listed on the same line; for example:

```
@Partial @Mock DataLoader loader;
```

There are no specific rules for formatting parameter and local variable annotations.

3.1.21.6 Comments

3.Block comment style

Block comments are indented at the same level as the surrounding code. They may be in `/* ... */` style or `// ...` style. For multi-line `/* ... */` comments, subsequent lines must start with `*` aligned with the `*` on the previous line.

```
/*
 * This is          // And so           /* Or you can
 * okay.            // is this.          * even do this. */
 */
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

Tip: When writing multi-line comments, use the `/* ... */` style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in `// ...` style comment blocks.

3.1.21.7 Modifiers

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

```
public protected private abstract static final transient volatile synchronized native strictfp
```

3.1.21.8 Numeric Literals

`long`-valued integer literals use an uppercase `L` suffix, never lowercase (to avoid confusion with the digit `1`). For example, `3000000000L` rather than `3000000000l`.

3.1.22 Naming

3.1.23 Rules common to all identifiers

Identifiers use only ASCII letters and digits, and in two cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+` .

In Google Style special prefixes or suffixes, like those seen in the examples `name_`, `mName`, `s_name` and `kName`, are **not** used.

3.1.24 Rules by identifier type

3.1.24.1 Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, `com.example.deepspace`, not `com.example.deepSpace` or `com.example.deep_space`.

3.1.24.2 Class names

Class names are written in UpperCamelCase.

Class names are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

There are no specific rules or even well-established conventions for naming annotation types.

Test classes are named starting with the name of the class they are testing, and ending with `Test`. For example, `HashTest` or `HashIntegrationTest`.

3.1.24.3 Method names

Method names are written in lowerCamelCase.

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop`.

Underscores may appear in JUnit *test* method names to separate logical components of the name. One typical pattern is `test <MethodUnderTest>_ <state>`, for example `testPop_emptyStack`. There is no One Correct Way to name test methods.

3.1.24.4 Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly?

Every constant is a static final field, but not all static final fields are constants. Before choosing constant case, consider whether the field really *feels like* a constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely *intending* to never mutate the object is generally not enough. Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final Joiner COMMA_JOINER = Joiner.on(',');  // because Joiner is immutable
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);
static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

These names are typically nouns or noun phrases.

3.1.24.5 Non-constant field names

Non-constant field names (static or otherwise) are written in lowerCamelCase.

These names are typically nouns or noun phrases. For example, `computedValues` or `index`.

3.1.24.6 Parameter names

Parameter names are written in lowerCamelCase.

One-character parameter names should be avoided.

3.1.24.7 Local variable names

Local variable names are written in lowerCamelCase, and can be abbreviated more liberally than other types of names.

However, one-character names should be avoided, except for temporary and looping variables.

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

3.1.24.8 Type variable names

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as `E`, `T`, `X`, `T2`)
- A name in the form used for classes (see Class names), followed by the capital letter `T` (examples: `RequestT`, `FooBarT`).

3.1.25 Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
    3. *Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case *per se*; it defies *any* convention, so this recommendation does not apply.
4. Now lowercase *everything* (including acronyms), then uppercase only the first character of:
    5. ... each word, to yield *upper camel case*, or

6. ... each word except the first, to yield *lower camel case*
7. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded. Examples:

Prose form	Correct	Incorrect
"XML HTTP request"	`XmlHttpRequest`	`XMLHTTPRequest`
"new customer ID"	`newCustomerId`	`newCustomerID`
"inner stopwatch"	`innerStopwatch`	`innerStopWatch`
"supports IPv6 on iOS?"	`supportsIpv6OnIos`	`supportsIPv6OnIOS`
"YouTube importer"	`YouTubeImporter` `YoutubeImporter*`	

*Acceptable, but not recommended.

Note: Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

3.1.26 Programming Practices

3.1.27 @Override: always used

A method is marked with the `@Override` annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

Exception: `@Override` may be omitted when the parent method is `@Deprecated`.

3.1.28 Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. (Typical responses are to log it, or if it is considered "impossible", rethrow it as an `AssertionError`.)

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

```
try {
    int i = Integer.parseInt(response);
    return handleNumericResponse(i);
} catch (NumberFormatException ok) {
    // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

Exception: In tests, a caught exception may be ignored without comment *if* it is named `expected`. The following is a very common idiom for ensuring that the method under test *does* throw an exception of the expected type, so a comment is unnecessary here.

```
try {
    emptyStack.pop();
    fail();
} catch (NoSuchElementException expected) {
}
```

3.1.29 Static members: qualified using class

When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

```
Foo aFoo = ...;
Foo.aStaticMethod(); // good
aFoo.aStaticMethod(); // bad
somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

3.1.30 Finalizers: not used

It is **extremely rare** to override `Object.finalize`.

Tip: Don't do it. If you absolutely must, first read and understand *Effective Java* Item 7, "Avoid Finalizers," very carefully, and *then* don't do it.

3.1.31 Javadoc

3.1.32 Formatting

3.1.32.1 General form

The *basic* formatting of Javadoc blocks is as seen in this example:

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

The basic form is always acceptable. The single-line form may be substituted when there are no at-clauses present, and the entirety of the Javadoc block (including comment markers) can fit on a single line.

3.1.32.2 Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (*)—appears between paragraphs, and before the group of "at-clauses" if present. Each paragraph but the first has `<p>` immediately before the first word, with no space after.

3.1.32.3 At-clauses

Any of the standard "at-clauses" that are used appear in the order `@param`, `@return`, `@throws`, `@deprecated`, and these four types never appear with an empty description. When an at-clause doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the `@`.

3.1.33 The summary fragment

The Javadoc for each class and member begins with a brief **summary fragment**. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does **not** begin with `A {@code Foo} is a...`, or `This method returns...`, nor does it form a complete imperative sentence like `Save the record.`. However, the fragment is capitalized and punctuated as if it were a complete sentence.

Tip: A common mistake is to write simple Javadoc in the form `/** @return the customer ID */`. This is incorrect, and should be changed to `/** Returns the customer ID. */`.

3.1.34 Where Javadoc is used

At the *minimum*, Javadoc is present for every `public` class, and every `public` or `protected` member of such a class, with a few exceptions noted below.

Other classes and members still have Javadoc *as needed*. Whenever an implementation comment would be used to define the overall purpose or behavior of a class, method or field, that comment is written as Javadoc instead. (It's more uniform, and more tool-friendly.)

3.1.34.1 Exception: self-explanatory methods

Javadoc is optional for "simple, obvious" methods like `getFoo`, in cases where there *really and truly* is nothing else worthwhile to say but "Returns the foo".

Important: it is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named `getCanonicalName`, don't omit its documentation (with the rationale that it would say only `/** Returns the canonical name. */`) if a typical reader may have no idea what the term "canonical name" means!

3.1.34.2 Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.