Docs

[User Manual](http://docs.google.com/userguide/userguide.html)

[Guides and Tutorials](https://guides.gradle.org)

[DSL Reference](http://docs.google.com/dsl/)

[Javadoc](http://docs.google.com/javadoc/)

[Release Notes](http://docs.google.com/release-notes.html)

[Forums](https://discuss.gradle.org/)

[Training](https://gradle.org/training/)

[Try Gradle Enterprise](https://gradle.com/enterprise)

[PDF](http://docs.google.com/userguide/userguide.pdf)

* [User Manual Home](http://docs.google.com/userguide/userguide.html)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Installing Gradle](http://docs.google.com/userguide/installation.html)
* [Tutorials](https://guides.gradle.org/)

### Reference

* [Groovy DSL Reference](http://docs.google.com/dsl/)
* [Gradle API Javadoc](http://docs.google.com/javadoc/)
* [Core Plugins](http://docs.google.com/userguide/plugin_reference.html)
* [Gradle & Third-party Tools](http://docs.google.com/userguide/third_party_integration.html)

### Getting Started

* [Creating New Gradle Builds](https://guides.gradle.org/creating-new-gradle-builds/)
* [Creating Build Scans](https://guides.gradle.org/creating-build-scans/)
* [Migrating From Maven](https://guides.gradle.org/migrating-from-maven/)

### Running Gradle Builds

* [Command-Line Interface](http://docs.google.com/userguide/command_line_interface.html)
* [Customizing Execution](#gjdgxs)
  + [Configuring the Build Environment](http://docs.google.com/userguide/build_environment.html)
  + [Configuring the Gradle Daemon](http://docs.google.com/userguide/gradle_daemon.html)
  + [Initialization Scripts](http://docs.google.com/userguide/init_scripts.html)
* [Directory Layout](http://docs.google.com/userguide/directory_layout.html)
* [Executing Multi-Project Builds](http://docs.google.com/userguide/intro_multi_project_builds.html)
* [Gradle Wrapper](http://docs.google.com/userguide/gradle_wrapper.html)
* [Troubleshooting](http://docs.google.com/userguide/troubleshooting.html)
* [Using Build Scans](https://docs.gradle.com/build-scan-plugin)
* [Enabling and Configuring the Build Cache](http://docs.google.com/userguide/build_cache.html)
* [Integrating Separate Gradle Builds (Composite Builds)](http://docs.google.com/userguide/composite_builds.html)

### Authoring Gradle Builds

* [Fundamentals](#30j0zll)
  + [Introducing the Basics of Build Scripts](http://docs.google.com/userguide/tutorial_using_tasks.html)
  + [Working with Tasks](http://docs.google.com/userguide/more_about_tasks.html)
  + [Learning More About Build Scripts](http://docs.google.com/userguide/writing_build_scripts.html)
  + [Working with Files](http://docs.google.com/userguide/working_with_files.html)
  + [Creating Custom Task Types](http://docs.google.com/userguide/custom_tasks.html)
  + [Using Gradle Plugins](http://docs.google.com/userguide/plugins.html)
  + [The Standard Gradle Plugins](http://docs.google.com/userguide/standard_plugins.html)
  + [Understanding the Build Lifecycle](http://docs.google.com/userguide/build_lifecycle.html)
  + [Working with Logging](http://docs.google.com/userguide/logging.html)
  + [Configuring Multi-Project Builds](http://docs.google.com/userguide/multi_project_builds.html)
* [Best Practices](#1fob9te)
  + [Authoring Maintainable Build Scripts](http://docs.google.com/userguide/authoring_maintainable_build_scripts.html)
  + [Organizing Gradle Projects](http://docs.google.com/userguide/organizing_gradle_projects.html)
  + [Optimizing Build Performance](https://guides.gradle.org/performance/)
  + [Using the Build Cache](https://guides.gradle.org/using-build-cache/)
* [Dependency Management](#3znysh7)
  + [Introduction to Dependency Management](http://docs.google.com/userguide/introduction_dependency_management.html)
  + [Dependency Management Terminology](http://docs.google.com/userguide/dependency_management_terminology.html)
  + [Dependency Types](http://docs.google.com/userguide/dependency_types.html)
  + [Repository Types](http://docs.google.com/userguide/repository_types.html)
  + [Declaring Dependencies](http://docs.google.com/userguide/declaring_dependencies.html)
  + [Declaring Repositories](http://docs.google.com/userguide/declaring_repositories.html)
  + [Inspecting Dependencies](http://docs.google.com/userguide/inspecting_dependencies.html)
  + [Managing Dependency Configurations](http://docs.google.com/userguide/managing_dependency_configurations.html)
  + [Managing Transitive Dependencies](http://docs.google.com/userguide/managing_transitive_dependencies.html)
  + [Dependency Locking](http://docs.google.com/userguide/dependency_locking.html)
  + [Troubleshooting Dependency Resolution](http://docs.google.com/userguide/troubleshooting_dependency_resolution.html)
  + [Customizing Dependency Resolution Behavior](http://docs.google.com/userguide/customizing_dependency_resolution_behavior.html)
  + [Dependency Cache Internals](http://docs.google.com/userguide/dependency_cache.html)
  + [Working with Dependencies](http://docs.google.com/userguide/working_with_dependencies.html)
* [Publishing Artifacts](http://docs.google.com/userguide/artifact_management.html)
* [C++ Projects](#2et92p0)
  + [Building Native Software](http://docs.google.com/userguide/native_software.html)
  + [Software Model Concepts](http://docs.google.com/userguide/software_model_concepts.html)
  + [Rule-based Model Configuration](http://docs.google.com/userguide/software_model.html)
  + [Implementing Model Rules in a Plugin](http://docs.google.com/userguide/rule_source.html)
  + [Extending the Software Model](http://docs.google.com/userguide/software_model_extend.html)
* [Java Projects](#tyjcwt)
  + [Building Java & JVM projects](http://docs.google.com/userguide/building_java_projects.html)
  + [Testing Java & JVM projects](http://docs.google.com/userguide/java_testing.html)
* [Advanced Techniques](#3dy6vkm)
  + [Configuring Tasks Lazily](http://docs.google.com/userguide/lazy_configuration.html)
  + [Developing Parallel Tasks](https://guides.gradle.org/using-the-worker-api/)
  + [Testing Your Build with TestKit](http://docs.google.com/userguide/test_kit.html)
  + [Using Ant from Gradle](http://docs.google.com/userguide/ant.html)
* [Sample Gradle builds](#1t3h5sf)
  + [Groovy DSL Samples](https://github.com/gradle/gradle/tree/master/subprojects/docs/src/samples)
  + [Kotlin DSL Samples](https://github.com/gradle/kotlin-dsl/tree/master/samples)

### Extending Gradle

* [Writing Custom Plugins](http://docs.google.com/userguide/custom_plugins.html)
* [Plugin Development Guides](https://gradle.org/guides/?q=Plugin+Development)

[Edit this page](https://github.com/gradle/gradle/edit/master/subprojects/docs/src/docs/userguide/)

# The Java Library Plugin

Contents

[Usage](#4d34og8)

[API and implementation separation](#2s8eyo1)

[Recognizing API and implementation dependencies](#17dp8vu)

[The Java Library plugin configurations](#3rdcrjn)

[Known issues](#26in1rg)

The Java Library plugin expands the capabilities of the Java plugin by providing specific knowledge about Java libraries. In particular, a Java library exposes an API to consumers (i.e., other projects using the Java or the Java Library plugin). All the source sets, tasks and configurations exposed by the Java plugin are implicitly available when using this plugin.

[Usage](#4d34og8)

To use the Java Library plugin, include the following in your build script:

[Example: Using the Java Library plugin](#lnxbz9)

**build.gradle**

apply plugin: 'java-library'

[API and implementation separation](#2s8eyo1)

The key difference between the standard Java plugin and the Java Library plugin is that the latter introduces the concept of an *API* exposed to consumers. A library is a Java component meant to be consumed by other components. It’s a very common use case in multi-project builds, but also as soon as you have external dependencies.

The plugin exposes two [configurations](http://docs.google.com/managing_dependency_configurations.html#managing_dependency_configurations) that can be used to declare dependencies: api and implementation. The api configuration should be used to declare dependencies which are exported by the library API, whereas the implementation configuration should be used to declare dependencies which are internal to the component.

[Example: Declaring API and implementation dependencies](#35nkun2)

**build.gradle**

dependencies {  
 api 'commons-httpclient:commons-httpclient:3.1'  
 implementation 'org.apache.commons:commons-lang3:3.5'  
}

Dependencies appearing in the api configurations will be transitively exposed to consumers of the library, and as such will appear on the compile classpath of consumers. Dependencies found in the implementation configuration will, on the other hand, not be exposed to consumers, and therefore not leak into the consumers' compile classpath. This comes with several benefits:

* dependencies do not leak into the compile classpath of consumers anymore, so you will never accidentally depend on a transitive dependency
* faster compilation thanks to reduced classpath size
* less recompilations when implementation dependencies change: consumers would not need to be recompiled
* cleaner publishing: when used in conjunction with the new maven-publish plugin, Java libraries produce POM files that distinguish exactly between what is required to compile against the library and what is required to use the library at runtime (in other words, don’t mix what is needed to compile the library itself and what is needed to compile against the library).

| **✨** | The compile configuration still exists but should not be used as it will not offer the guarantees that the api and implementation configurations provide. |
| --- | --- |

If your build consumes a published module with POM metadata, the Java and Java Library plugins both honor api and implementation separation through the scopes used in the pom. Meaning that the compile classpath only includes compile scoped dependencies, while the runtime classpath adds the runtime scoped dependencies as well.

This often does not have an effect on modules published with Maven, where the POM that defines the project is directly published as metadata. There, the compile scope includes both dependencies that were required to compile the project (i.e. implementation dependencies) and dependencies required to compile against the published library (i.e. API dependencies). For most published libraries, this means that all dependencies belong to the compile scope. However, as mentioned above, if the library is published with Gradle, the produced POM file only puts api dependencies into the compile scope and the remaining implementation dependencies into the runtime scope.

| **✨** | Separating compile and runtime scope of modules is active by default in Gradle 5.0+. In Gradle 4.6+, you need to activate it by adding enableFeaturePreview('IMPROVED\_POM\_SUPPORT') in *settings.gradle*. |
| --- | --- |

[Recognizing API and implementation dependencies](#17dp8vu)

This section will help you identify API and Implementation dependencies in your code using simple rules of thumb. The first of these is:

* Prefer the implementation configuration over api when possible

This keeps the dependencies off of the consumer’s compilation classpath. In addition, the consumers will immediately fail to compile if any implementation types accidentally leak into the public API.

So when should you use the api configuration? An API dependency is one that contains at least one type that is exposed in the library binary interface, often referred to as its ABI (Application Binary Interface). This includes, but is not limited to:

* types used in super classes or interfaces
* types used in public method parameters, including generic parameter types (where *public* is something that is visible to compilers. I.e. , *public*, *protected* and *package private* members in the Java world)
* types used in public fields
* public annotation types

By contrast, any type that is used in the following list is irrelevant to the ABI, and therefore should be declared as an implementation dependency:

* types exclusively used in method bodies
* types exclusively used in private members
* types exclusively found in internal classes (future versions of Gradle will let you declare which packages belong to the public API)

The following class makes use of a couple of third-party libraries, one of which is exposed in the class’s public API and the other is only used internally. The import statements don’t help us determine which is which, so we have to look at the fields, constructors and methods instead:

[Example: Making the difference between API and implementation](#1ksv4uv)

**src/main/java/org/gradle/HttpClientWrapper.java**

// The following types can appear anywhere in the code  
// but say nothing about API or implementation usage  
import org.apache.commons.httpclient.\*;  
import org.apache.commons.httpclient.methods.\*;  
import org.apache.commons.lang3.exception.ExceptionUtils;  
import java.io.IOException;  
import java.io.UnsupportedEncodingException;  
  
public class HttpClientWrapper {  
  
 private final HttpClient client; // private member: implementation details  
  
 // HttpClient is used as a parameter of a public method  
 // so "leaks" into the public API of this component  
 public HttpClientWrapper(HttpClient client) {  
 this.client = client;  
 }  
  
 // public methods belongs to your API  
 public byte[] doRawGet(String url) {  
 GetMethod method = new GetMethod(url);  
 try {  
 int statusCode = doGet(method);  
 return method.getResponseBody();  
  
 } catch (Exception e) {  
 ExceptionUtils.rethrow(e); // this dependency is internal only  
 } finally {  
 method.releaseConnection();  
 }  
 return null;  
 }  
  
 // GetMethod is used in a private method, so doesn't belong to the API  
 private int doGet(GetMethod method) throws Exception {  
 int statusCode = client.executeMethod(method);  
 if (statusCode != HttpStatus.SC\_OK) {  
 System.err.println("Method failed: " + method.getStatusLine());  
 }  
 return statusCode;  
 }  
}

The *public* constructor of HttpClientWrapper uses HttpClient as a parameter, so it is exposed to consumers and therefore belongs to the API. Note that GetMethod is used in the signature of a *private* method, and so it doesn’t count towards making HttpClient an API dependency.

On the other hand, the ExceptionUtils type, coming from the commons-lang library, is only used in a method body (not in its signature), so it’s an implementation dependency.

Therefore, we can deduce that commons-httpclient is an API dependency, whereas commons-lang is an implementation dependency. This conclusion translates into the following declaration in the build script:

[Example: Declaring API and implementation dependencies](#44sinio)

**build.gradle**

dependencies {  
 api 'commons-httpclient:commons-httpclient:3.1'  
 implementation 'org.apache.commons:commons-lang3:3.5'  
}

[The Java Library plugin configurations](#3rdcrjn)

The following graph describes the main configurations setup when the Java Library plugin is in use.



* The configurations in *green* are the ones a user should use to declare dependencies
* The configurations in *pink* are the ones used when a component compiles, or runs against the library
* The configurations in *blue* are internal to the component, for its own use
* The configurations in *white* are configurations inherited from the Java plugin

And the next graph describes the test configurations setup:



| **✨** | The *compile*, *testCompile*, *runtime* and *testRuntime* configurations inherited from the Java plugin are still available but are deprecated. You should avoid using them, as they are only kept for backwards compatibility. |
| --- | --- |

The role of each configuration is described in the following tables:

Table 1. Java Library plugin - configurations used to declare dependencies

| **Configuration name** | **Role** | **Consumable?** | **Resolvable?** | **Description** |
| --- | --- | --- | --- | --- |
| api | Declaring API dependencies | no | no | This is where you should declare dependencies which are transitively exported to consumers, for compile. |
| implementation | Declaring implementation dependencies | no | no | This is where you should declare dependencies which are purely internal and not meant to be exposed to consumers. |
| compileOnly | Declaring compile only dependencies | yes | yes | This is where you should declare dependencies which are only required at compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime. |
| runtimeOnly | Declaring runtime dependencies | no | no | This is where you should declare dependencies which are only required at runtime, and not at compile time. |
| testImplementation | Test dependencies | no | no | This is where you should declare dependencies which are used to compile tests. |
| testCompileOnly | Declaring test compile only dependencies | yes | yes | This is where you should declare dependencies which are only required at test compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime. |
| testRuntimeOnly | Declaring test runtime dependencies | no | no | This is where you should declare dependencies which are only required at test runtime, and not at test compile time. |

Table 2. Java Library plugin — configurations used by consumers

| **Configuration name** | **Role** | **Consumable?** | **Resolvable?** | **Description** |
| --- | --- | --- | --- | --- |
| apiElements | For compiling against this library | yes | no | This configuration is meant to be used by consumers, to retrieve all the elements necessary to compile against this library. Unlike the default configuration, this doesn’t leak implementation or runtime dependencies. |
| runtimeElements | For executing this library | yes | no | This configuration is meant to be used by consumers, to retrieve all the elements necessary to run against this library. |

Table 3. Java Library plugin - configurations used by the library itself

| **Configuration name** | **Role** | **Consumable?** | **Resolvable?** | **Description** |
| --- | --- | --- | --- | --- |
| compileClasspath | For compiling this library | no | yes | This configuration contains the compile classpath of this library, and is therefore used when invoking the java compiler to compile it. |
| runtimeClasspath | For executing this library | no | yes | This configuration contains the runtime classpath of this library |
| testCompileClasspath | For compiling the tests of this library | no | yes | This configuration contains the test compile classpath of this library. |
| testRuntimeClasspath | For executing tests of this library | no | yes | This configuration contains the test runtime classpath of this library |

[Known issues](#26in1rg)

[Compatibility with other plugins](#2jxsxqh)

At the moment the Java Library plugin is only wired to behave correctly with the java plugin. Other plugins, such as the Groovy plugin, may not behave correctly. In particular, if the Groovy plugin is used in addition to the java-library plugin, then consumers may not get the Groovy classes when they consume the library. To workaround this, you need to explicitly wire the Groovy compile dependency, like this:

[Example: Configuring the Groovy plugin to work with Java Library](#z337ya)

**a/build.gradle**

configurations {  
 apiElements {  
 outgoing.variants.getByName('classes').artifact(  
 file: compileGroovy.destinationDir,  
 type: ArtifactTypeDefinition.JVM\_CLASS\_DIRECTORY,  
 builtBy: compileGroovy)  
 }  
}

[Increased memory usage for consumers](#3j2qqm3)

When a project uses the Java Library plugin, consumers will use the output classes directory of this project directly on their compile classpath, instead of the jar file if the project uses the Java plugin. An indirect consequence is that up-to-date checking will require more memory, because Gradle will snapshot individual class files instead of a single jar. This may lead to increased memory consumption for large projects.

Docs

* [User Manual](http://docs.google.com/userguide/userguide.html)
* [DSL Reference](http://docs.google.com/dsl/)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Javadoc](http://docs.google.com/javadoc/)

News

* [Blog](https://blog.gradle.org/)
* [Newsletter](https://newsletter.gradle.com/)
* [Twitter](https://twitter.com/gradle)

Products

* [Build Scans](https://gradle.com/build-scans)
* [Build Cache](https://gradle.com/build-cache)
* [Enterprise Docs](https://gradle.com/enterprise/resources)

Get Help

* [Forums](https://discuss.gradle.org/c/help-discuss)
* [GitHub](https://github.com/gradle/)
* [Training](https://gradle.org/training/)
* [Services](https://gradle.org/services/)

Subscribe for important Gradle updates and news

Subscribe

By entering your email, you agree to our [Terms](https://gradle.org/terms/) and [Privacy Policy](https://gradle.org/privacy/), including receipt of emails. You can unsubscribe at any time.

© [Gradle Inc.](https://gradle.com) 2018 All rights reserved.

[Careers](https://gradle.com/careers) | [Privacy](https://gradle.org/privacy) | [Terms of Service](https://gradle.org/terms) | [Contact](https://gradle.org/contact/)