Table of Contents

Cabe 2

Cabe

TLDR

Add this to your Gradle build file to add automatic null-checks to the public API of your project and assertion based null checks to the private API:

Kotlin DSL

```
plugins {
  id("com.dua3.cabe") version "3.0-beta-11"
}
```

Read on for examples and more detailed configuration options.

Cabe is a Java byte code instrumentation tool that inserts checks based on JSpecify annotations into your class files.

According to the Fail-Fast Principle (https://www.martinfowler.com/ieeeSoftware/failFast.pdf), all invalid input should be detected and reported early. Cabe helps you doing this by automatically checking method and constructor parameters.

Cabe also helps you during development by checking return values of methods.

What are JSpecify Annotations?

JSpecify is a project that aims to enhance Java code by providing a set of annotations specifically designed to improve code quality and facilitate better type-checking. These annotations help developers specify nullability and (later) other type-related constraints more precisely, allowing for more robust and error-free code. This ultimately leads to improved documentation, better IDE support, and more reliable results during static analysis.

You can find more details about JSpecify on their official website (https://jspecify.dev).

What does Cabe do?

Cabe analyzes your classes byte code and injects code that checks for violations of the

nullability rules you added to your code by using JSpecify annotations.

> ⚠ The example projects here are contained within the "examples" subproject. To compile and run the examples, use the commmand `./gradlew -Dexamples examples:<name>:run` from the project's main folder.

> ⚠ Note that running the examples will usually result in a failure as the axamples demonstrate how using Cabe asserts your programs fail-fast when nullability violations are detected.

Let's look at an example (use `./gradlew :examples:hello:run` to execute):

```
import org.jspecify.annotations.NonNull;

public class hello.Hello {
    public static void main(String[] args) {
        sayHello(null);
    }

    public static void sayHello(@NonNull String name) {
        System.out.println("hello.Hello, " + name + "!");
    }
}
```

The contract of `sayHello(@NonNull String name)` is that `name` must have a non-null value when called. When you run this code through Cabe, using the standard configuration, an automatic null-check will be inserted, and when you run the program, you will see this:

```
Exception in thread "main" java.lang.NullPointerException: name is
null
at hello.Hello.sayHello(hello.Hello.java)
at hello.Hello.main(hello.Hello.java:5)
```

You can of course obtain the same result by using this code:

```
public void printGreeting(@NonNull String name) {
    Objects.requireNonNull(name, "name is null");
```

```
    System.out.println("hello.Hello, " + name);
}
```

Have a look at `hellofx` for another example. In this example, the program is trying to read a resource from the classpath that can not be found. When not using Cabe, the stack trace will look like this:

```
Exception in Application start method
java.lang.reflect.InvocationTargetException
    at
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Dir
ectMethodHandleAccessor.java:118)
    at java.base/java.lang.reflect.Method.invoke(Method.java:580)
    at
javafx.graphics/com.sun.javafx.application.LauncherImpl.launchApplic
ationWithArgs(LauncherImpl.java:464)
    at
javafx.graphics/com.sun.javafx.application.LauncherImpl.launchApplic
ation(LauncherImpl.java:364)
    at
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Dir
ectMethodHandleAccessor.java:103)
    at java.base/java.lang.reflect.Method.invoke(Method.java:580)
    at
java.base/sun.launcher.LauncherHelper$FXHelper.main(LauncherHelper.j
ava:1149)
Caused by: java.lang.RuntimeException: Exception in Application
start method
    at
javafx.graphics/com.sun.javafx.application.LauncherImpl.launchApplic
ation1(LauncherImpl.java:893)
    at
javafx.graphics/com.sun.javafx.application.LauncherImpl.lambda$launc
hApplication$2(LauncherImpl.java:196)
    at java.base/java.lang.Thread.run(Thread.java:1583)
Caused by: java.lang.NullPointerException: Input stream must not be
null
    at
```

```
javafx.graphics/javafx.scene.image.Image.validateInputStream(Image.j
ava:1140)
    at javafx.graphics/javafx.scene.image.Image.<init>
(Image.java:707)
    at hellofx.HelloFX.loadImage(HelloFX.java:21)
    at hellofx.HelloFX.start(HelloFX.java:28)
    at
javafx.graphics/com.sun.javafx.application.LauncherImpl.lambda$launc
hApplication1$9(LauncherImpl.java:839)
    at
javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runAn
dWait$12(PlatformImpl.java:483)
    at
javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runLa
ter$10(PlatformImpl.java:456)
    at
java.base/java.security.AccessController.doPrivileged(AccessControll
er.java:400)
    at
javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runLa
ter$11(PlatformImpl.java:455)
    at
javafx.graphics/com.sun.glass.ui.InvokeLaterDispatcher$Future.run(In
vokeLaterDispatcher.java:95)
Exception running application hellofx.HelloFX
```

The stacktrace indicates an error happened inside a JavaFX method, two methods down in the stack, our method `loadImage()` appears. When we now open our IDE and look at the actual code, it becomes apparent that the real problem lies in `getResourceAsStream()` returning `null` and not a valid streeam.

Now what happens if Cabe is used? We will get this stacktrace:

```
Exception in Application start method
java.lang.reflect.InvocationTargetException
        at
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Dir
ectMethodHandleAccessor.java:118)
        at
```

```
java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at
javafx.graphics/com.sun.javafx.application.LauncherImpl.launchApplic
ationWithArgs(LauncherImpl.java:464)
        at
javafx.graphics/com.sun.javafx.application.LauncherImpl.launchApplic
ation(LauncherImpl.java:364)
        at
java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(Dir
ectMethodHandleAccessor.java:103)
        at
java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at
java.base/sun.launcher.LauncherHelper$FXHelper.main(LauncherHelper.j
ava:1149)
Caused by: java.lang.RuntimeException: Exception in Application
start method
        at
javafx.graphics/com.sun.javafx.application.LauncherImpl.launchApplic
ation1(LauncherImpl.java:893)
        at
javafx.graphics/com.sun.javafx.application.LauncherImpl.lambda$launc
hApplication$2(LauncherImpl.java:196)
        at java.base/java.lang.Thread.run(Thread.java:1583)
Caused by: java.lang.AssertionError: invalid null return value
        at hellofx.HelloFX.getResourceAsStream(HelloFX.java:19)
        at hellofx.HelloFX.loadImage(HelloFX.java:23)
        at hellofx.HelloFX.start(HelloFX.java:31)
        at
javafx.graphics/com.sun.javafx.application.LauncherImpl.lambda$launc
hApplication1$9(LauncherImpl.java:839)
        at
javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runAn
dWait$12(PlatformImpl.java:483)
        at
javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runLa
ter$10(PlatformImpl.java:456)
        at
java.base/java.security.AccessController.doPrivileged(AccessControll
```

```
er.java:400)
        at
javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runLa
ter$11(PlatformImpl.java:455)
        at
javafx.graphics/com.sun.glass.ui.InvokeLaterDispatcher$Future.run(In
vokeLaterDispatcher.java:95)
```

Note that the cause now points directly at the exact method that returned the invalid `null` value. This is of course a rather trivial example, but it shows how Cabe can simplify your workflow.

> ⚠ If you look at the HelloFX source code, you might notice that the method return value was not even marked as `@NonNull`. Instead the class was marked as `@NullMarked`. This basically means that unless something is explicitly marked as `@Nullable`, everything inside that class is implicitly treated as if it were annotated as `@NonNull`.

> ℹ The `@NullMarked` annotation can be applied to classes, packages, and even Jigsaw modules. Read more about how to use JSpecify annotations in the Nullness User Guide (https://jspecify.dev/docs/user-guide/).

Will using Cabe impact performance?

Depending on the configuration used, there may be a minor overhead. Depending on your requirements, you can use different configurations.

- During development, it is recommended to enable all checks so that programming errors will show up during early testing.
- Once you are certain your project is thoroughly tested, you can restrict parameter checking to methods called by third-party code. When the standard setting is used, parameters to private API methods are checked using standard assertions that can be enabled or disabled at runtime. Standard assertions are usually optimized out by the JIT compiler when assertions are disabled. There may still be a minor impact due to increased class file size.

- You can also disable all checks by using the `NO_CHECK` configuration and there will be no performance hit at all. Be aware that you might trade correctness to speed in this case as invalid inputs will not be detected.

When in doubt, profile your application when compiled using the different settings.

How are Null Checks implemented?

This depends on the configuration used. There are four types of checks. The examples assume a non-nullable parameter #named `p`.

Check	Equivalent Java code
NO_CHECK	[N/A]
STANDARD_ASSERTION	`assert p!=null : "p is null";`
ASSERT_ALWAYS	`if (p==null) throw new AssertionError("p is null");`
THROW_NPE	`if (p==null) throw new NullPointerException("p is null");`

Public vs Private API

When developing a library, you can configure different checks for:

- the **public API** of your library so that invalid parameter values are detected when the user of your library calls your code with disallowed `null` values for a parameter,
- the **private API** of your library, i.e., code that cannot be directly called by users of your library.

> In the standard configuration, parameters that are part of the public API will always be checked and throw a `NnullPointerException` when an `null` parameter value is detected where it is not allowed and for parts of your private API, standard assertions are used that can be enabled at runtime using the standard `-ea` flag.

What are the predefined configurations?

The predefined configurations are:

Name	Public API	Private API	Return Values
DEVELOPMENT	ASSERT_ALWAYS	ASSERT_ALWAYS	ASSERT_ALWAYS
STANDARD	THROW_NPE	ASSERT	NO_CHECK
NO_CHECK	NO_CHECK	NO_CHECK	NO_CHECK

Things to note

Here are some points that you should be aware of when using Cabe.

Records

Cabe supports Java Records.

Do I need to explicitly add a Record Constructor so that Parameters can be checked?

No, Cabe adds the checks by evaluating the record declaration:

```
record MyRecord(@NonNull String a, @Nullable String b) {}

void foo() {
    // this works
    MyRecord r1 = new MyRecord("a", null);
    // this will throw an exception "a is null"
    MyRecord r1 = new MyRecord(null, "b");
}
```

Standard Assertions cannot be generated for Record classes

Cabe currently cannot inject standard assertions into record classes because of technical restrictions. That is why for records, THROW_NPE is used instead of ASSERT. ASSERT_ALWAYS works as it does for other classes.

Technical background

Standard assertions use a special boolean flag `$assertionsDisabled` that is initialised by the JVM when the class is loaded to the value obtained by calling `Class.getDesiredAssertionStatus()`.

For classes that do not contain any assertions in their source code, this flag is not present in the class file and has to be injected into the byte code. the initialisation is then done in a static initializer block. This does not work for records and results in an `InvalidClassFileException`.

Arrays

Cabe will detect when `null` is passed for a non-nullable array parameter. It will however not detect null values contained in an array.

```
void foo(@NonNull String[] array) {}
void bar(@NonNull String @NonNull[] array) {}

void test() {
    // exception will be thrown
    foo(null);
    // no exception will be thrown although null elements are
disallowed
    bar(new String[] {"123", null});
}
```

Generics

Cabe will detect violations for generic parameters when it can be determined at compile time that a type is non nullable:

```
@NullMarked
class Generic<T, U extends @Nullable String> {

    void foo(T t, U u) {}

    void test() {
        // will throw a NullPointerException
        foo(null, "u");

        // will not throw
```

```
        foo("t", null);
    }
}
```

If cannot check parameters where the nullability can not be determined at compile time:

```
@NullUnmarked
class Generic2<T> {
    void foo(T t) {}
}

{
    // this does not throw, null is allowed
    new Generic2<String>().foo(null);

    // this also does not throw because the nullability can not be
determined at compile time of foo()
    new Generic2<@NonNull String>().foo(null);
}
```

SpotBugs

If you use SpotBugs in your build, it may report unnecessary null checks, i.e., when Cabe is configured to check method return values and SpotBugs byte code analysis infers the checked value will be non-null anyway. In that case, you might want to use a SpotBugs exclusion file.

```
<FindBugsFilter>
    <Match>
        <!-- Bugs reported by SpotBugs for automatic injected null
checks -->
        <Or>
            <Bug
pattern="RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE"/>
            <Bug
pattern="RCN_REDUNDANT_COMPARISON_OF_NULL_AND_NONNULL_VALUE"/>
            <Bug pattern="SA_LOCAL_SELF_ASSIGNMENT" />
        </Or>
```

```
    </Match>
</FindBugsFilter>
```

Using Cabe in your Gradle Build

Cabe can be used either as a standalone program that you can run manually to instrument your class files or as a Gradle plugin that runs automatically in your build process. Let's see how it is done with Gradle.

To use Cabe in your Gradle build, add the plugin to your build script and configure the plugin:

Kotlin DSL

```
plugins {
  id("com.dua3.cabe") version "3.0-beta-11"
}
```

Groovy DSL

```
plugins {
  id "com.dua3.cabe" version "3.0-beta-11"
}
```

This will run the Cabe processor in your build. When no configuration is given, a standard configuration is used.

Configure the Cabe Task

To configure the instrumentation, you can configure Cabe like this to use one of the predefined configurations:

Kotlin DSL

```
cabe {
    config.set(Configuration.STANDARD)
}
```

If you omit the configuration block in your build, the standard configuration will be used.

Using different Configurations for Development and Release Builds

You can also automatically select a configuration based on your version string. In this example, strict checking is done for snapshot and beta versions whereas a release build will use the standard configuration:

Kotlin DSL

```
val isSnapshot =
project.version.toString().toDefaultLowerCase().contains("snapsh
ot")
cabe {
    if (isSnapshot) {
        config.set(Configuration.DEVELOPMENT)
    } else {
        config.set(Configuration.STANDARD)
    }
}
```

Defining Custom Configurations

You can define a custom configuration that differs from the provided predefined configurations by providing a configuration String:

```
cabe {

config.set(Configuration.parse("publicApi=THROW_NPE:privateApi=ASSER
T:returnValue=ASSERT_ALWAYS"))
}
```

When using a configuration String, you can use either

- a predefined name: "STANDARD", "DEVELOPMEN", "NOCHECKSS"
- a single Check to be used public and private API and return values
- multiple combination of keys ("publicApi", "privateApi", "returnValue") and checks; in this the remaining will be set to "NO_CHECK"

Examples:

Configuration String	Public API	Private API	Return Value
"STANDARD"	THROW_NPE	ASSERT	NO_CHECK
"DEVELOPMENT"	ASSERT_ALWAYS	ASSERT_ALWAYS	ASSERT_ALWAYS
"NO_CHECKS"	NO_CHECK	NO_CHECK	NO_CHECK
"THROW_NPE"	THROW_NPE	THROW_NPE	THROW_NPE
"ASSERT"	ASSERT	ASSERT	ASSERT
"ASSERT_ALWAYS"	ASSERT_ALWAYS	ASSERT_ALWAYS	ASSERT_ALWAYS
"NO_CHECK"	NO_CHECK	NO_CHECK	NO_CHECK
"THROW_NPE"	THROW_NPE	THROW_NPE	THROW_NPE
"publicApi=THROW_NPE"	THROW_NPE	NO_CHECK	NO_CHECK
"publicApi=THROW_NPE:returnValue=ASSERT"	THROW_NPE	NO_CHECK	ASSERT

You can also use the standard record constructor of `Configuration`

```
cabe {
    config.set(new Configuration(Check.THROW_NPE, Check.ASSERT,
Check.ASSERT_ALWAYS))
}
```

What about the Name?

In Javanese, both cabe and lombok both refer to chili peppers. At the same time, Lombok is a... well, I really don't know. It is something between a library and a language on its own that extends Java with certain features. One of these features are annotations to mark nullable and non-nullable types and code instrumentation to do runtime checks based on these annotations.

While widely used, Lombok is quite controversial, you will find plenty of discussions on this topic on the internet.

Newer Java versions brought many features that developers used Lombok for, perhaps most notably Java recorrds.

Having automated null checks in your code is one Lombok feature that I liked but could not find any non-Lombok alternative. That's why I started Cabe, and that's where the name came from.