TESTING

Refactoring Java, Part 1: Driving agile development with test-driven development

Refactoring makes your organization's code simpler, which means fewer bugs and easier maintenance.

by Mohamed Taman

October 9, 2020

Refactoring is about enhancing the consistency of the code by simplifying the code. Simpler code facilitates versatility and the ability to rapidly change the code, introduce new functionality, and meet the organization's ever-changing needs.

In this series, I'll teach refactoring by practicing refactoring. The goal is to write and refactor code, optimize it, and add new features along the way. Well-factored code helps developers solve problems more quickly and develop and produce high-quality software products that customers enjoy quickly.

What are we going to learn?

This hands-on kata-based series of articles teaches refactoring basics, tailored to agile development. (I'll describe what a kata is shortly.) This first article will help set up a test-driven development (TDD) environment and walk through basic refactoring techniques, such as variable renaming, extracting methods, and inlining methods.

The second article will explain how legacy code can be stabilized by addressing outstanding technical debt—inefficiencies and errors introduced due to sloppy and careless programming.

The third article will demonstrate refactoring to simplify code, remove duplication, and build more reusable objects. You'll see how refactoring complements an agile workflow by demonstrating how to add a new feature to the simplified codebase quickly.

Let’s start with some basic definitions and concepts.

What is refactoring? Refactoring is your code improvement process. The goal of refactoring is to improve the quality and to encourage the modification of your software product. Refactoring makes the code simpler. If you have fewer code lines, you will have fewer potential bugs.

Two principles of agile software development applicable here are simplicity and technical excellence:

- Simpler code enables technical agility—the ability to quickly change the code to add new functionality, for example.
- Technical agility enables business agility—the opportunity to change our tech products easily to satisfy the ever-changing needs of our users and other stakeholders.

Therefore, a well-factored codebase provides the flexibility to solve business problems faster and to build and deliver high-quality software products quickly—products that people love.

Quality first: Test-driven development. Rigorous testing is the foundation of refactoring. Without well-defined test cases that verify the correct behavior of your code, you cannot be sure that refactoring hasn’t altered the code’s external behavior. The refactoring process is about changing the internal structure of your code without affecting its external behavior.

This article will follow the TDD practice of writing tests, where I’ll write automated tests to prove the correct behavior of the code. When I write new code, the TDD will guide writing the valid code. And when I refactor, that is, improve the current code, TDD gives guide rails to make it safe.

When writing new features, TDD is an iterative process or a three-step dance—red, green, refactor:

- Write a new test for that new feature; of course, the code will fail the test because the feature doesn’t work yet. TDD says the test has gone red.
- Write Java code to implement the new feature until the new test passes. All the other tests must continue passing as well. This means the tests have gone green.
- Clean up and improve the production code by refactoring the code, which results in better code. All the tests must continue to pass.

Repeat these steps over and over until we’re satisfied with the new feature, as shown in **Figure 1**.

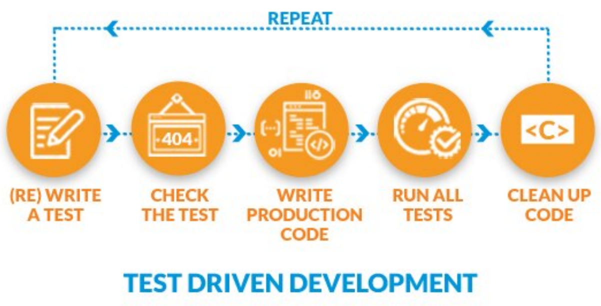


Figure 1. The iterative dance of test-driven development

For a deeper look into TDD, you can read my article, "Test-driven development: Really, it's a design technique."

Learning with code katas. A code kata is a technique for learning new coding skills. In martial arts, kata is a series of movements that you do regularly. You repeat your actions so many times that the body builds muscle memory. Without thinking about them, you will execute the moves.

A code kata is like that but for a coding problem: Take a simple problem and follow a sequence of moves to solve it. Repeat the series of moves so many times that they become muscle memory, so your body and brain will just know what to do the next time they encounter a similar coding problem.

The sequence of moves I'll show you here is the red-green-refactor loop of test-driven development. You'll repeat the red-green-refactor loop so often that you'll build the muscle memory for good refactoring. The goal is that when you write real production code, you won't need to think about refactoring; you'll just do it automatically.

What should you know? To get the most from these articles, you should be familiar with writing code. Refactoring is applicable in any programming language. I will do all the development and refactoring in Java, but you don't have to be a Java guru to follow along with me.

You can get the full source code files from my GitHub repository or clone it:

```
~$ git clone https://github.com/mohamed-taman
```

If you would like to follow along with me without coding, simply follow the steps in this article. But if you would like to navigate the code, then each article is divided into steps, and in each step, I have a git commit for each TDD red-green-refactor change. So, when navigating commits, you can notice the differences and the refactoring that is done toward the final kata requirements.

To get the most value, please fire up your favorite IDE and write and refactor code along with the article instructions. To do that, you'll need the following:

- The latest Java Development Kit; I will use Java SE 15.
- Any IDE you want; I will use JUnit for automated testing, and I'll use code coverage tools.
- IntelliJ IDEA Ultimate edition; I'll use this edition in the articles because JUnit and code coverage tools are built in.

Now we can set up our first exercise, which is to write software that converts Roman numerals to Arabic numerals. Arabic numerals are the numbers that we use every day, such as 1, 2, and 3. Roman numerals are numbers written with letters, such as I, V, and X. Here's the user story for this kata: As a schoolteacher, I want a tool that converts Roman numerals to Arabic numerals, so my students can check their homework grades quickly.

Step 1: Setting up the new code kata

In this example, I won't go deep for the complete requirements. Instead, I'll present a few simple cases to lay out the more critical concepts and techniques of the TDD refactoring process. So, don't judge the process by this simple example; there are wealthy concepts, tips, and tricks coming next.

If you want to know more about the complete Roman numerals example with TDD, my article referenced above goes into details.

To get started on the kata, follow these instructions—and remember that although I'm using IntelliJ, you can use any IDE:

1. Launch IntelliJ and create a new project. On the left side, select `Maven` and click `next`.

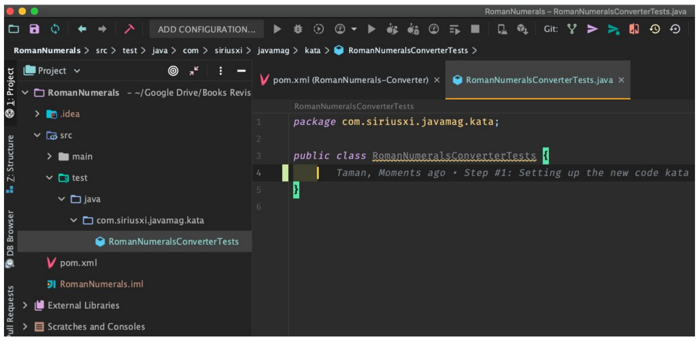
2. Because this is the Roman numerals kata, for `Name` type something like `RomanNumerals`, for `Groupid` type `com.siriusxi.javamag.kata`, and for `Artifactid` type `RomanNumerals-Converter`. Then click `finish`.

3. In the bottom right corner of the screen, you'll see a window that says, "`Maven projects need to be imported`." Click `enable auto-import`.

4. In the project navigator, open `RomanNumerals`, open the source `(src)` folder, and then open the `test` folder. Right-click the `java` folder and click `new package`. The package name matches the Maven Groupid. Type `com.siriusxi.javamag.kata` and click `okay`.

5. Right-click the `com.siriusxi.javamag.kata` package and select `new Java class`. For the name of the new class, type `RomanNumeralsConverterTests` and click `okay`.

You'll notice that IntelliJ autogenerates the first few lines of code (see **Figure 2**). There are no executable code lines, which means this is the perfect piece of software: The zero lines of code have zero bugs.


RomanNumerals – RomanNumeralsConverterTests.java
ADD CONFIGURATION...
Git:
RomanNumerals > src > test > java > com > siriusxi > javamag > kata > RomanNumeralsConverterTests >
1: Project
Z: Structure
DB Browser
Pull Requests
Project
RomanNumerals ~/Google Drive/Books Revis
.idea
src
main
test
java
com.siriusxi.javamag.kata
RomanNumeralsConverterTests
pom.xml
RomanNumerals.iml
External Libraries
Scratches and Consoles
pom.xml (RomanNumerals-Converter)
RomanNumeralsConverterTests.java
RomanNumeralsConverterTests

```
1  package com.siriusxi.javamag.kata;
2
3  public class RomanNumeralsConverterTests {
4         Taman, Moments ago • Step #1: Setting up the new code kata
5  }
6
```

Figure 2. The IDE autogenerates zero lines of code, which had zero bugs.

The goal through the rest of the kata is to keep the code as close to this ideal as possible. You want to have the fewest lines of code that you can, because that way, you will have the fewest bugs and the most technical agility and business agility.

Step 2: Setting up the TDD testing environment

Let's set up JUnit as the test framework to proceed with the kata at hand. Remember the red-green-refactor three-step dance? First, we will create a failing test that ensures JUnit exists and works properly:

1. In your code editor, at line 4, type `@Test`, and then press Enter.
2. Continue typing the method name, `public void isJunitWorking()`, and a left squiggly bracket, and then press Enter.
3. And at line 6, type `assertTrue(false)`.

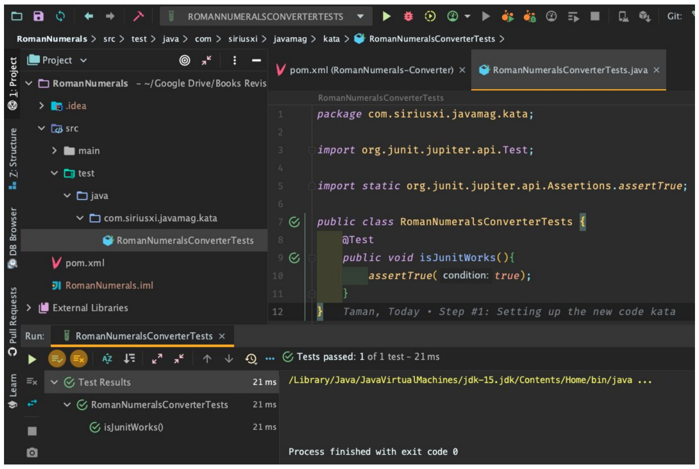
Notice all the red on the screen; this code is so red that it will not even compile. The problem here is that the project is not configured yet to use the JUnit framework. To resolve this quickly so we can focus more on solving the kata, I will ask IntelliJ to fix the configuration for us.

1. Move the cursor to line 4 on top of the word `@Test`, and then press `Option+Enter` on macOS (`alt+enter` on Windows) and select to add `JUnit 5.4` to the classpath. Behind the scenes, IntelliJ adds JUnit to the project configuration. Also notice at line 3 that the `import` statements appeared.
2. Notice that `assertTrue` at line 8 is still red. Press `Option+Enter` as IntelliJ suggests, and the `import` statements appear at line 5. Also, notice that all the red has disappeared. This code can now compile.
3. Now to run the unit test: Right-click `RomanNumeralsConverterTests` and select

Run `RomanNumeralsConverterTests`. In the JUnit window at the bottom of the screen, see that `isJunitWorking` has failed. This is perfect. We have a failing test. This is the first step of the red-green-refactor dance in TDD.

4. The goal is to write the least amount of code to make this test go green. At line 10, change the word `false` to `true`.

5. Rerun the tests and notice that everything has gone green. You are now at the green part of the red-green-refactor dance. Look at the code. It looks like there's nothing to refactor here—that is, there's no room for improvement—so you are done (see **Figure 3**).


ROMANNUMERALSCONVERTERTESTS
Git:
RomanNumerals > src > test > java > com > siriusxi > javamag > kata > RomanNumeralsConverterTests >
Project
RomanNumerals - ~/Google Drive/Books Revis
.idea
src
main
test
java
com.siriusxi.javamag.kata
RomanNumeralsConverterTests
pom.xml
RomanNumerals.iml
External Libraries
1: Project
Z: Structure
DB Browser
Pull Requests
Learn
pom.xml (RomanNumerals-Converter)
RomanNumeralsConverterTests.java
RomanNumeralsConverterTests
Run:
RomanNumeralsConverterTests

```
1   package com.siriusxi.javamag.kata;
2
3   import org.junit.jupiter.api.Test;
4
5   import static org.junit.jupiter.api.Assertions.assertTrue;
6
7   public class RomanNumeralsConverterTests {
8       @Test
9       public void isJunitWorks(){
10          assertTrue( condition: true);
11      }
12  }   Taman, Today • Step #1: Setting up the new code kata
```

```
Tests passed: 1 of 1 test – 21 ms

Test Results                    21 ms
  RomanNumeralsConverterTests   21 ms
    isJunitWorks()              21 ms

/Library/Java/JavaVirtualMachines/jdk-15.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

Figure 3. The Java code doesn't do anything, but at least it passes its first unit test.

By the way, I let IntelliJ do the project configuration work for me. Modern IDEs are good at automating certain kinds of work, such as project configuration. As we proceed through the kata, I'll rely on the IDE as much as possible to automate project configuration and similar kinds of work.

Step 3: First refactoring—renaming a variable

To solve the Roman numerals kata, I'll start with the simplest test case—single-digit Roman numerals. For example, the Roman numeral *I* converts to Arabic numeral *1*, *V* means 5, *X* means 10, and so on. We will start to solve the problem by writing our first test of the behavior.

At line 11, press `enter` twice. Type `@Test`, press `enter`, and then type `public void convertsSingleRomanDigit(){}`.

TDD will guide you toward architecting the solution to the problem. The test you write will guide you toward a solution code. You are looking for a method that returns an integer. The method will have a name that's like "convert." It will take an input string, and the input string is the Roman numeral that you want to convert. Thus, it'll look something like this:

```
int arabic = convert("I");
```

To finish the test case, type `assert equals 1`, which is the expected result of converting the Roman numeral *I* to an Arabic integer and the return value from that `convert` method: `assertEquals(1, arabic);`

There's a lot of red here. This is good. We're back to the first step of the red-green-refactor dance. Once again, the code is so red it won't even compile. Let's fix that:

1. Here IntelliJ is offering us a hint. If we press `Option+Enter`, it will insert an `import` statement for us at line 5. So, the `assert equals` will compile.
2. And `convert` is red because there is no actual `convert` method in the Java code. Click the red-light icon to get some suggestions. The `create method convert` is the one we want. Press `enter` a few times to accept all the defaults.
3. Running the tests gives a red result; the code compiles, but the new test is failing. You are going to write the least amount of code to make the test go green.
4. Back at line 21, we'll change this to return one. Rerun the tests. And it's all green. It's time for the third step of the dance: Are there opportunities to improve the code?

Step 3.1: Code improvement. The first thing I see is the argument name `i` at line 20. That single letter is not very descriptive, and that's not good for long-term maintenance. I will change that to be more descriptive.

Highlight the `i`, right-click, select `refactor`, and then select `rename`. We'll change the name of this argument to be more descriptive about what it does. Give it a name like `romanNumeral`. Press `enter` and rerun the tests to make sure we didn't break any of the external behavior. We're still all green.

That's the first refactoring: I changed the variable's name to better reveal the purpose of the variable.

Step 4: A series of microrefactorings

As I look at the code, I see that something else can be improved. The new method is created in the test class as if it were part of the test code, and this is not correct. The new method is part of the solution, so we should move this method into a separate class. That class should be part of the main area of the source code and not part of the test area of the source code.

The excellent process that we are going to perform is a series of refactorings. Sometimes refactoring is large, and that can be risky, since it might introduce a new defect. Therefore, instead of doing a single large refactoring, you'll do a series of smaller microrefactorings. After each microrefactoring, rerun the test to make sure nothing broke.

Step 4.1: Change convert() to static. The first microrefactoring we'll try is to move the `convert` method to its class.

1. Right-click `convert`, click `refactor` and then `move`. Now IntelliJ suggests that we should first make it static to be able to move it to its class. So, this is our first microrefactoring.

2. Click `yes`, and then click the `refactor` button to confirm that this is what we want.

3. Rerun the tests to make sure this didn't change any external behavior. Everything is still green, so we're ready for the next microrefactoring.

Step 4.2: Move convert() to its own class. The second microrefactoring will move `convert` into its own class.

1. Right-click `refactor`, `move`, and type the new class's fully qualified class name.

2. The class will be in the same package, `com.siriusxi.javamag.kata`. I'll call the class `RomanNumeralsConverter`; it will be the class with the solution kata. Click the `refactor` button and confirm that this is what you want.

3. Rerun the tests to make sure this didn't change any external behavior. Everything is still green, so you're ready for the next microrefactoring.

Step 4.3: Move RomanNumeralsConverter to main src folder. Move the Roman class from the test source code area to the main source code area. To do that, follow these instructions:

1. Drag the `RomanNumeralsConverter` class across the project navigator, from the test `java` folder to the main `java` folder. IntelliJ prompts for the package's name; give it the same package name, `com.siriusxi.javamag.kata`.

   Note: IntelliJ defaults to putting the class into the source test directory, but we want it to go into the source main directory, so select that instead. Click `okay` to put the class into the correct destination directory.

2. Click `refactor`. Rerun the tests to make sure this didn't change any external behavior. Everything is still green, so you're ready for the next microrefactoring.

You have successfully completed three microrefactorings, which add up to a large-scale refactoring.

Step 5: Adding new single-digit Roman numerals cases

Let's continue to solve the kata. The next text cases will be for the Roman numerals V and X.

1. First, write a test to make sure that V returns 5. Running all the tests gives you a red test result, as expected.

2. To resolve that, write the least amount of code to make this case go to green. For example,
   `if Roman is 1 then return 1, else return 5.`

Rerun the tests, and all tests are green. Great. It looks like there's nothing to improve here, so let us move to the next test case.

3. Let's add another test that goes red. Repeat the previous steps 1 and 2, but for the new case Roman X: `assert equals 10.` Run the tests. This test is red because you have not implemented that solution code yet, so now write the most straightforward code that will make that test go green.

4. Add an `else if` branch: `Roman equals v, return 5, else return 10.` Rerun the tests, and they are all green. Now we are at the refactor step of the red-green-refactor three-step dance.

The first refactoring to tackle is repeated code: DRY, or Don't Repeat Yourself. Duplicated code is challenging to maintain over time, because if you need to make a small change to one of these repeated code lines, you will have to make that change over and over; it will slow you down and decrease your technical agility. Plus, if you miss a change, you could introduce a hard-to-find defect.

1. Do a refactoring here to inline some of the code. Right-click on `Arabic` at line 23, click `refactor`, and `inline` or `option+command+N`. This will replace that variable at line 23 with the value that you assigned to it at line 22 and remove line 22.

2. Click `refactor` to apply the refactoring. Rerun the tests to make sure this didn't change any external behavior. Everything is still green, so you're good.

3. Repeat the same steps for lines 20 and 17. For each step, rerun the tests to make sure you didn't change any external behavior.

One last thing: You should beautify this code a little bit by deleting lines 17 and 18. This won't change any actual code, but you should rerun the tests anyway. Get into the habit of running the tests after every change. Everything is still green, so you're all set.

Step 6: Continue solving the code kata

I have added all the additional tests and the solution code for all the single-digit Roman numeral cases. You can complete that part of the kata yourself, or you can look at the solution code that I've provided.

Let's move on to the next test case. In Roman numerals, adding new digits often looks like addition:

- II is the same thing as 1 plus 1, and in our code, we'll be looking to return the value of 2.
- III would be the same as 1 plus 1 plus 1, and our code should return the value of 3.

- VI would be the same as 5 plus 1, and our code should return the value of 6.

Now, my examples are obviously simplified, because IV means 4, not 6; a smaller digit to the left means subtraction, not addition. However, the goal here is to demonstrate TDD and refactoring, not to master the art of Roman numerals.

That said, I'll write the first test case for the simple addition:

1. At line 23 press `enter` twice, and type `@Test public void romanNumeralAddition(){}`. And the first assertion will be `assertEquals(2, RomanNumeralsConverter.convert("II"))`.

2. The tests will fail when run. This is perfect; we're at the red step of the red-green-refactor loop.

3. Write the least amount of code to make the tests go green. At line 4, press `enter`, and type `if(romanNumeral.equals("II")) return 2`. To make the code compile, put an `else` before `if` on the next line.

4. Run the tests to make sure everything goes green.

Now, this is obviously not the final solution to the problem. This is a very naive solution; we are only saying if we get the string II, return 2. We're not even doing any real addition here. This is how kata works. The point of the kata is not necessarily to solve the Roman numeral problem!

The point of the kata is to practice the moves. And the moves we're practicing are the red-green-refactor three-step dance of test-driven development. Going from red to green, we're going to write the least amount of code. Let's run the tests and see if we get green. Alright, they're all green.

Let's add more assertions for III and VI. Our next assertions will follow the same steps:

- `Assert Equals` 3 and `RomanNumeralsConverter.convert("II")`, which is 1 plus 1 plus 1. Run the tests, and notice that this test is; this is good. Write the least amount of code to make that go green.
- The least amount of code to make that go green will look like this: `else if romanNumeral.equals III return 3`. Again, it's not the best solution, but the goal is to practice the moves. Run the tests. And you'll get green.
- Finally, let's do one more for 6, which is VI in Roman numerals.

If you are looking for a complete solution with TDD in practice for the full Roman to Arabic problem, it's in the article referenced earlier, "Test-driven development: Really, it's a design technique."

Step 6.1: Rearrange the code for ease of next refactoring. The `convert` method is too long to fit on the screen. The title of the method has scrolled off the top of the screen. I can't see the bottom of the method. It's scrolled off the bottom of the editor.

A couple of microrefactorings will improve this. The first microrefactoring will be to take the code segment that deals with the single-digit Roman numerals and move it to the top of the method.

1. If it's a single-digit Roman numeral, type that as `if(romanNumeral.length() == 1)`. Then do that single-digit Roman numeral stuff. However, if it's a multidigit Roman numeral, then we'll do the multidigit addition stuff.
2. To do this microrefactoring, manually cut and paste the code for single-digit Roman numerals. Starting at line 15, select the code down to line 29, cut that code, and paste it at line 7. To make the code compile and run successfully, go to line 29 and delete the dangling `else`.
3. The code will also need a final `return` statement that's not connected to an `if`. Delete that last `if`, and we'll just say `else return 6`. Run the tests. They are still green, so this microrefactoring was successful.

Now, the method is still too long, but we've made it easy for the next microrefactoring.

Step 6.2: Refactor the extract method. The next microrefactoring is to extract lines 7 through 21 into a separate method. Highlight lines 7 through 21, right-click, and select `refactor`, `extract and method`, or press `option+command+M`. Give this new method a name such as `convertSingleDigit`. Accept the rest of the defaults and click `refactor`, and now there's a new method that IntelliJ created for you at line 18.

Now you can see the whole method on the screen at once. It's easy to understand, and you have better technical agility.

Oh, let's run the tests. That's the next move in the dance. Everything's green, so we're certain we haven't changed any of the external behavior. This refactoring has succeeded.

We have one more refactoring on our list. This one is called the `extract` method. It creates a new method from part of a larger method. You can work on that on your own.

Keep working on this kata. Don't worry about whether you solve the kata. The point is to practice the red-green-refactor dance.

Conclusion

Refactoring your code helps you to develop quality code—the foundation you need to quickly react to change, add new features, and deliver high-performance products.

This article went through a hands-on kata-based Roman numeral to Arabic numeral converter problem that teaches basic refactoring techniques, tailored to agile development.

You learned how to set up a test-driven development environment for new code. And we walked through basic refactoring techniques, such as variable renaming, moving code, extracting methods, and inline methods, by doing a microrefactoring. The point is to practice the moves: red, green, refactor.

In the next article, I'll explain how to stabilize legacy code that has outstanding technical debt—inefficiencies and errors introduced due to sloppy and careless programming.

And in the third article, you will use refactoring to simplify the legacy code, remove duplication, and build more reusable objects. Finally, I will show you how refactoring complements an agile workflow by demonstrating how to add a new feature to the simplified codebase quickly.

Dig deeper

- Test-driven development: Really, it's a design technique
- JUnit 5—A special issue of *Java Magazine*
- Interview with Kent Beck, the parent of JUnit and creator of TDD
- Unit testing your application with JUnit
- Simplified test-driven development with Oracle Visual Builder
- Two books: *Refactoring to Patterns* by Joshua Kerievsky and *Refactoring: Improving the Design of Existing Code* by Martin Fowler

Mohamed Taman

Mohamed Taman (@_tamanm) is the CEO of SiriusXI Innovations and a Chief Solutions Architect for Effortel Telecommunications. He is based in Belgrade, Serbia, and is a Java Champion, and Oracle Groundbreaker, a JCP member, and a member of the Adopt-a-Spec program for Jakarta EE and Adopt-a-JSR for OpenJDK.

Share this Page