

Overview

This document is intended to give you the feeling of the advantages of IntelliJ IDEA static code analysis tool that helps you to maintain and clean up your code through the analysis performed without actually executing the code.

IntelliJ IDEA is capable of detecting dozens of error-types and inconsistencies. First of all, it helps you to find probable bugs that are not “compilation errors”. Its flexible mechanism of resolving problems allows you to easily improve the code structure, conform your code to numerous guidelines and standards, detect performance issues and so on.

IntelliJ IDEA’s static code analysis is performed on-the-fly. Various inconsistencies, probable bugs, redundancies, spec violations, etc. are highlighted in the editor right while you are typing. Moreover, IntelliJ IDEA provides you with intelligent quick-fixes. So it ensures code quality from the very beginning, without interrupting your coding process. And that makes you more productive and saves your time and money.

In this document we will show you several examples of how static code analysis works. Also we’ll describe how IntelliJ IDEA allows you to manage static code analysis options. With the help of customizable inspection profiles you can specify the inspections you want to be applied to the scopes, modules or etc. Moreover we’ll mention here other advantages of IntelliJ IDEA static code analysis such as ability to suppress inspections for some pieces of code.

Examples of code inspections use

More than 600 automated Code Inspections help you easily detect different inconsistencies. In IntelliJ IDEA you’ll find that all inspections are grouped by their goals and sense. Every inspection has the appropriate description, so we won’t describe each group separately. But we’ll try to highlight the most common tasks that are covered by the static code analysis. They are:

- Finding probable bugs
- Locating the “dead” code
- Detecting performance issues
- Improving code structure and maintainability
- Conforming to coding guidelines and standards
- Conforming to specifications

Finding probable bugs

IntelliJ IDEA analyzes the code you are typing and is capable of finding and fixing probable bugs as non “compilation errors” right on-the-fly. Here is an example of such situation.

Potential NPEs that can be thrown at application runtime.

Before

```
public double requestCurrentRate(String fromCurrency, String toCurrency) {
    if (fromCurrency == null && toCurrency == null) {
        return Double.NaN;
    }
    double answer = 0;
    if (fromCurrency.equals("USD") && toCurrency.equals("CDN")) {
```

Method invocation 'fromCurrency.equals("USD")' may produce 'java.lang.NullPointerException'.

- Assert 'fromCurrency != null'
- Surround with 'if (fromCurrency != null)'
- Invert If Condition
- Replace '&&' with '||'
- Flip '&&'
- Replace .equals() with ==
- Flip '.equals()'

Here the first if condition may lead to NullPointerException being thrown in the second if, as not all situations are covered. At this point adding an assertion in order to avoid a NullPointerException being thrown during the application runtime would be a good idea.

After

```
public double requestCurrentRate(String fromCurrency, String toCurrency) {
    if (fromCurrency == null && toCurrency == null) {
        return Double.NaN;
    }

    double answer = 0;
    assert fromCurrency != null;
    if (fromCurrency.equals("USD") && toCurrency.equals("CDN")) {
        answer = rate;
    }
    if (fromCurrency.equals("CDN") && toCurrency.equals("USD")) {
        answer = 1 / rate;
    }
    return answer;
}
```

So, this is exactly what we get from the intention action.

Locating the “Dead” Code

IntelliJ IDEA highlights in the editor pieces of so-called “dead” code. This is the code that is never executed during the application runtime. Perhaps, you don’t even need this part of code in your project. Depending on situation, such code

may be treated as a bug or as a redundancy. Anyway it decreases the application performance and complicates the maintenance process. Here is an example.

So-called “constant conditions” - conditions that are never met or are always “true”, for example. In this case the responsible code is not reachable and actually is a “dead” code.

```

attribute = parseAttribute(isempty, asp, php);

if (attribute == null) {
    ...
    return;
}

value = parseValue(attribute, false, isempty, delim);

if (attribute != null) {
    ... Condition 'attribute != null' is always 'true'.
}

else {
    av = new AttVal( null, null, null, null,
                    0, attribute, value );
    Report.attrError(this, this.token, value,
                    Report.BAD_ATTRIBUTE_VALUE);
}

```

IntelliJ IDEA highlights the if condition as it's always true. So the part of code surrounded with else is actually a dead code as it is never executed.

Detecting performance issues

IntelliJ IDEA suggests users an easy way of solving problems concerned with performance throughout an entire project. IntelliJ IDEA knows a lot of situations where there are some patterns known to be performed faster than any others. So it can suggest solutions to increase the performance. So, here are several examples:

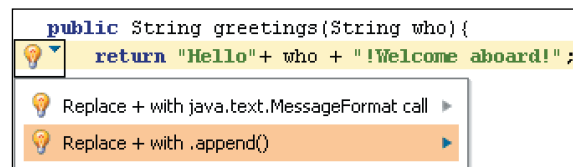
1. String operations.

Before

```

public String greetings(String who){
    return "Hello"+ who + "!Welcome aboard!";
}

```



The image shows a code snippet with a lightbulb icon next to the return statement. A dropdown menu is open, showing two suggestions: "Replace + with java.text.MessageFormat call" and "Replace + with .append()". The second option is highlighted in orange.

IntelliJ IDEA suggests you several actions to improve the performance of this string operation. We select the second one as it is known to be faster working.

After

```

public String greetings(String who){
    return new StringBuilder().append("Hello ")
        .append(who)
        .append("! Welcome aboard!")
        .toString();
}

```

And here is the result.

2. So called “tail recursion”, that appears when your code contains method that calls itself as its last action. That means the recursive call is the last thing that happens in the method. Such construction not seems to be an error but may lead to performance problems, especially when used frequently or on rather complicated methods.

Before

```
private static boolean afterSpace(Node node) {
    Node prev;
    int c;

    if (node == null || node.tag == null) return true;

    prev = node.prev;

    if (prev != null) {
        if (prev.type == Node.TextNode && prev.end > prev.start) {
            c = ((int)prev.textarray[prev.end - 1]) & 0xFF;
            if (c == 160 || c == ' ' || c == '\n') return true;
        }
        return false;
    }
    return afterSpace(node.parent);
}
```

Tail recursive call afterSpace()

- Replace tail recursion with iteration
- Expand boolean use to if-else

In this example IntelliJ IDEA detects such “tail recursion”, as described above, and suggests a quick-fix for it.

After

```
private static boolean afterSpace(Node node) {
    while (true) {
        Node prev;
        int c;

        if (node == null || node.tag == null) return true;

        prev = node.prev;

        if (prev != null) {
            if (prev.type == Node.TextNode && prev.end > prev.start) {
                c = ((int) prev.textarray[prev.end - 1]) & 0xFF;
                if (c == 160 || c == ' ' || c == '\n') return true;
            }
            return false;
        }
        node = node.parent;
    }
}
```

With the help of quick-fix, this tail recursion is easily transformed into iteration.

Improving code structure and maintainability

IntelliJ IDEA performs on-the-fly analysis of dependencies through a project,

module or package, detecting them and helping you in controlling dependencies through breaking your code into modules, using temporary cyclic dependencies, error highlighting. IntelliJ IDEA also looks for structural duplicates, even “fuzzy” duplicates.

To learn more about analyzing code dependencies please read

Analyzing Code Dependencies (part 1) –

<http://blogs.jetbrains.com/idea/2006/04/analyzing-code-dependencies-part-ii/>

and

Analyzing Code Dependencies (part 2) –

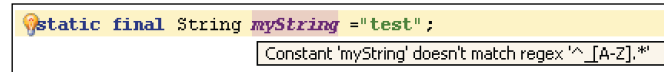
<http://blogs.jetbrains.com/idea/2006/04/analyzing-code-dependencies-part-i/> .

Conforming to coding guidelines and standards

IntelliJ IDEA allows you to find different inconsistencies concerned with numerous coding guidelines and standards, from common Java standards to any specific corporate standards. All places where, for example, Javadoc inconsistencies are found, are highlighted in the editor on-the-fly.

Here we include for example:

Naming conventions



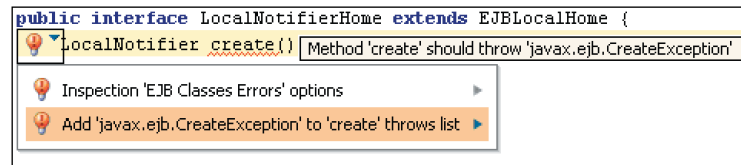
```
static final String myString = "test";
```

In this case we defined that all constants in the project should begin with “_” symbol. The yellow light bulb alarms that the constant does not match our own standard that we set for such names. IntelliJ IDEA can help you with solving the problem. Just press Alt + Enter, and select to rename the symbol. All the usages of this symbol in the code will be also renamed.

Conforming to specifications

IntelliJ IDEA highlights all specification violations, like EJB, JSP, JSF, etc. Unlike guidelines inconsistencies, spec violations result in impossibility to deploy the application on the server. That’s why it’s important to find such code places on-the-fly. Here is an example.

Before



```
public interface LocalNotifierHome extends EJBLocalHome {
    LocalNotifier create();
}
```

According to EJB specification, method “create” should throw corresponding exception. IntelliJ IDEA highlights this violation and suggests the quick-fix. If we won’t fix it, the application won’t be able to deploy at the server side.

After

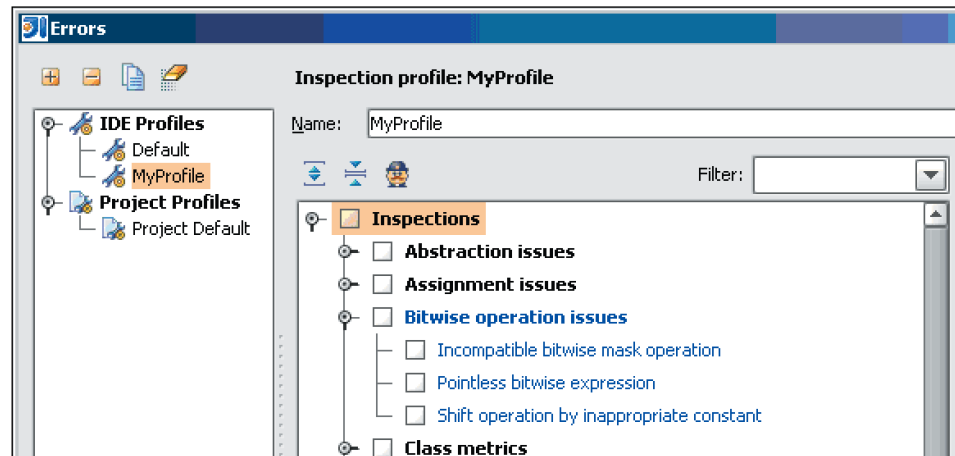
```
public interface LocalNotifierHome extends EJBLocalHome {
    LocalNotifier create() throws CreateException;
}
```

When quick-fix is applied, the exception is added.

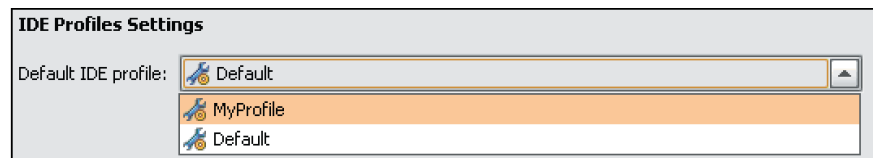
Managing static code analysis

Customizing inspection Profiles

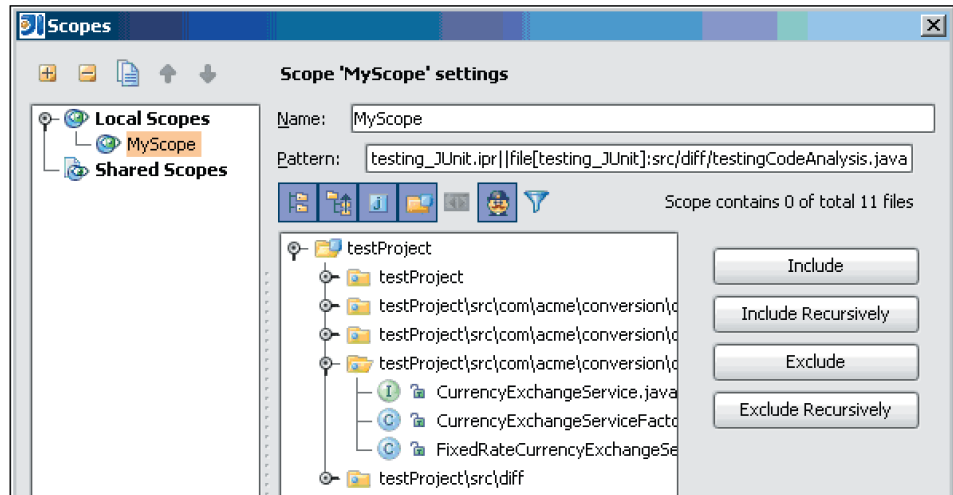
It's obvious that you won't need all inspections in each project. But while working on different projects or modules you may need different combinations of inspections to be applied to the code being written. For this purposes IntelliJ IDEA allows you to specify your own inspections set for each module or project, or you can customize the profile for the whole IDE. You can add a new profile in the **Settings | Errors** area. Just click the plus button and specify the name for the profile. Then select the inspections to be used in this profile.



In the list of available inspections you can find out that some items are black-colored and the others are blue-colored. Black color means that the item has the same state as in the default profile, blue color means the item differs from the default profile item. You can create a profile with all necessary inspections specified and tie it to the project, or use the default one that also can be changed or administered.

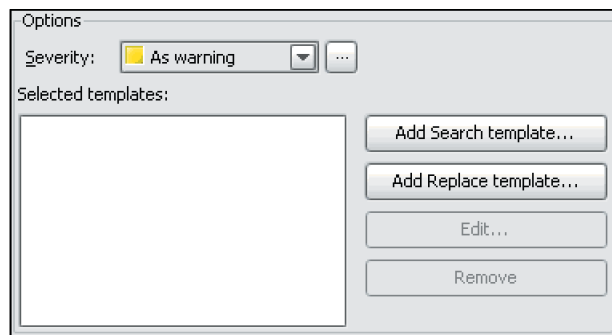


Furthermore you can define the scope for the selected inspections to be applied to in the **Project Profile Settings**. A scope defines the set of files on which code analysis is performed. A scope can include a single file, a folder or package, class or project. Shared scopes are available to your team mates and help you share settings, e.g. inspection profiles defined for specific code parts, the most common search scopes, etc.



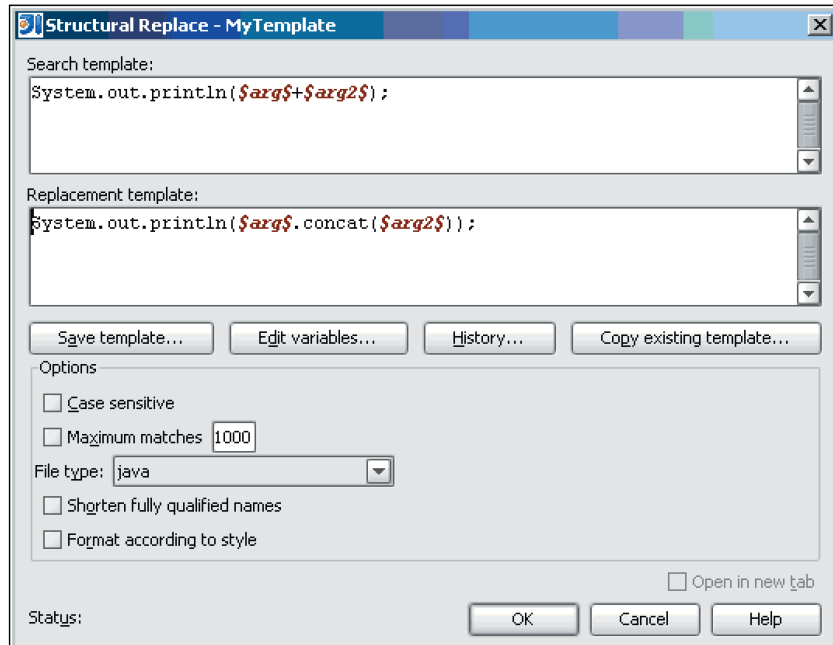
Creating your own inspections

IntelliJ IDEA allows you to create your own inspections based on code templates. In the inspection list you can find Structural Search Inspection that actually is a powerful tool for creating search templates.

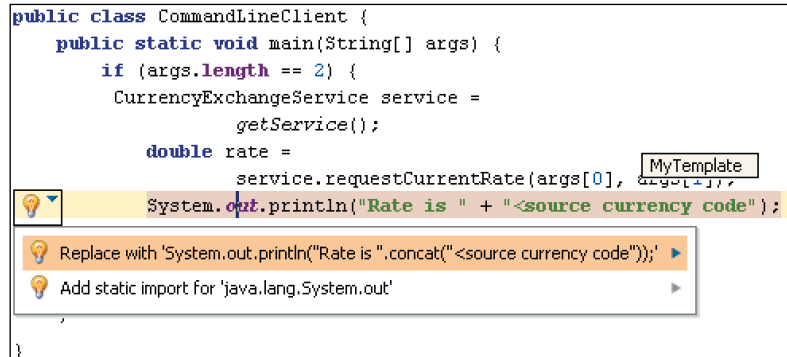


With the help of this inspection you can create any number of search code templates as well as search and replace template combinations.

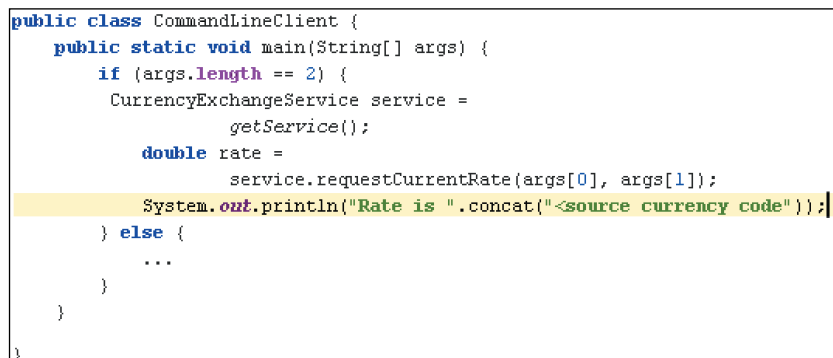
For example we'll create search and replace template for string operations.



We activate this inspection in the project profile and IntelliJ IDEA highlights all inconsistencies in the code.



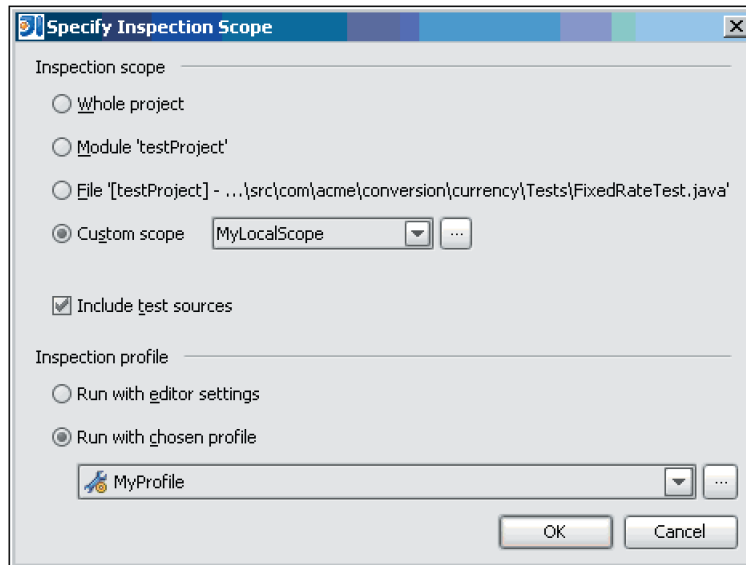
Moreover, it provides the replace template that we defined earlier. Selecting it we'll get exactly what we wanted.



For more details on Structural Search and Replace please read the following article <http://www.jetbrains.com/idea/documentation/ssr.html>

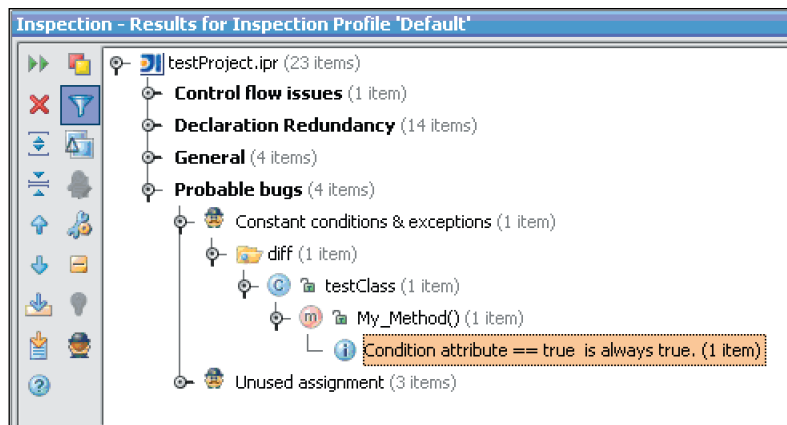
Running inspections

Although the analysis is performed on-the-fly, you may want to run it for the whole project or custom scope and view the results in a tree view. To do so, just select **Analyze | Inspect Code** from the menu. The following dialog appears.

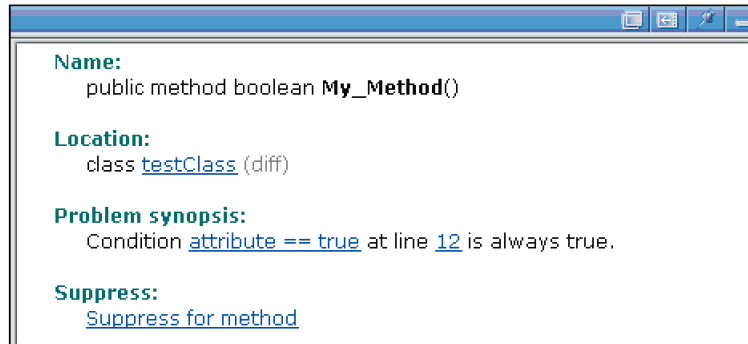


Here you can select the inspection profile, specify the scope and choose whether you want to include test sources to the analysis or not. The results of the analysis are presented at the bottom in the following form:

On the left hand side you see the tree view with all inconsistencies detected



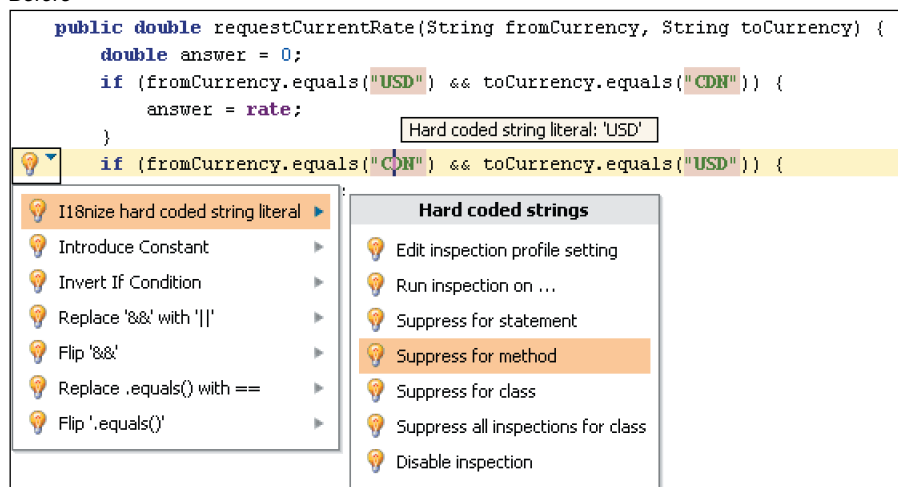
On the right hand side you see the description of the currently selected in the tree view inconsistency. From here you can move directly to the code that is considered to be unsatisfactory. You can apply quick fixes to it, or suppress the inspection.



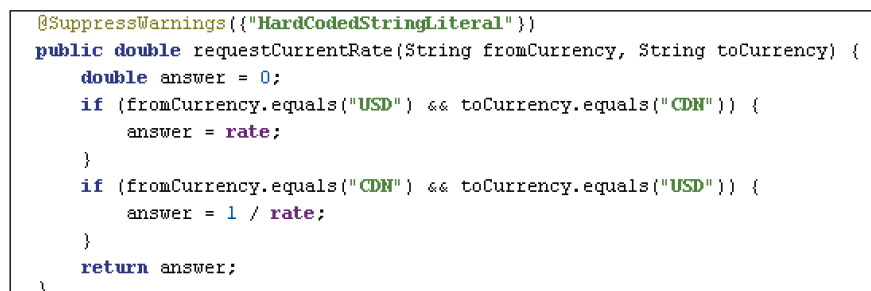
Suppressing inspections for the code

Of course, the static code analysis can not be treated as a final instance, and it only aims to help you in writing high-quality, coherent and maintainable code. And sometimes you may want to leave as is for some reasons some parts of your code, that are reported as problematic. For example, IntelliJ IDEA considers some code to be “dead”, and at this point it is never executed, but may be useful later. IntelliJ IDEA allows you to suppress the inspection for this code so the code will be left as is without any changes.

Before



In this example there are hard coded strings that we'd like to leave as is. So we can suppress this inspection for this method.



Conclusion

Thank you for your time and attention. Of course, it is not possible to cover each inspection and each advantage of code analysis separately just in one article. Also we did not mention that there are also code inspections for CSS, JavaScript, etc. For more details you can refer to the following articles:

- Analyzing Code Dependencies (part 1)
<http://blogs.jetbrains.com/idea/2006/04/analyzing-code-dependencies-part-i/>
- Analyzing Code Dependencies (part 2)
<http://blogs.jetbrains.com/idea/2006/04/analyzing-code-dependencies-part-ii/>
- Structural Search and Replace
<http://www.jetbrains.com/idea/documentation/ssr.html>
- Optimizing your CSS
<http://blogs.jetbrains.com/idea/2006/05/optimizing-your-css/>
- i18n support
<http://blogs.jetbrains.com/idea/2006/03/i18n-support/>