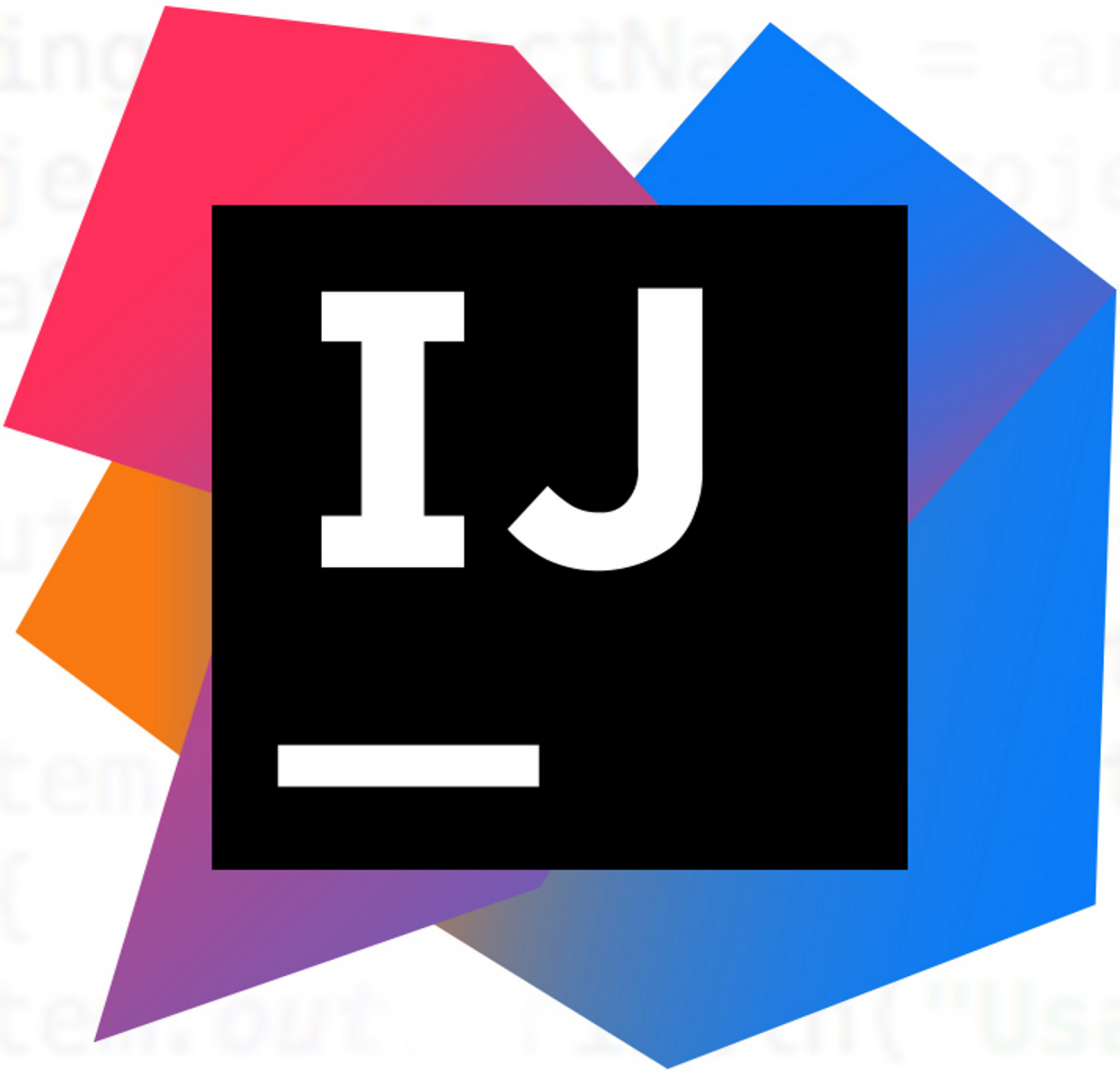
INTELLIJ IDEA HANDBOOK

Hot Recipes for the IntelliJ IDEA IDE



JAVA CODE GEEKS

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

IntelliJ IDEA Handbook

Contents

Preface

IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software. It is developed by JetBrains, and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition. Both can be used for commercial development.

The IDE provides for integration with build/packaging tools like grunt, bower, gradle, and SBT. It supports version control systems like GIT, Mercurial, Perforce, and SVN. Databases like Microsoft SQL Server, ORACLE, PostgreSQL, and MySQL can be accessed directly from the IDE.

IntelliJ supports plugins through which one can add additional functionality to the IDE. One can download and install plugins either from IntelliJ's plugin repository website or through IDE's inbuilt plugin search and install feature (Source: https://en.wikipedia.org/-wiki/IntelliJ_IDEA).

In this ebook, we provide a compilation of IntelliJ IDEA tutorials that will help you kick-start your own programming projects. We cover a wide range of topics, from setup and configuration, to plugins installation and UI creation. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at https://www.javacodegeeks.com/

Chapter 1

Intellij IDEA Tutorial for Beginners

This example demonstrates how you can get started with IntelliJ IDEA for the first time. Intellij IDEA is a JetBrains IDE product that supports multiple languages and frameworks (e.g. Java, Scala, Groovy) and works cross platforms i.e Windows, OS X, and Linux.

Intellij IDEA comes in two editions:

- Community Edition - Free and open source
- Ultimate Edition - Commercial

A comprehensive feature comparison between the two editions can be accessed from the JetBrains website.

1.1 Setup

In this example we shall use Intellij IDEA 15.0 Community Edition, which can be downloaded from the JetBrains website.

The following is the list of recommended system requirements for running Intellij IDEA 15.0:

- 2 GB RAM
- 300 MB hard disk space + at least 1 G for caches
- JDK 1.6 or higher

1.2 Creating a new project

When you working with intelliJ everything that you do you do it within the context of a project. A project is a representation of a complete solution that you have developed and comprises of source code, libraries and configuration files.

When you open intelliJ after the installation has been done you will be greeted by the welcome screen:

Figure 1.1: Welcome screen

Click → **Create New Project**

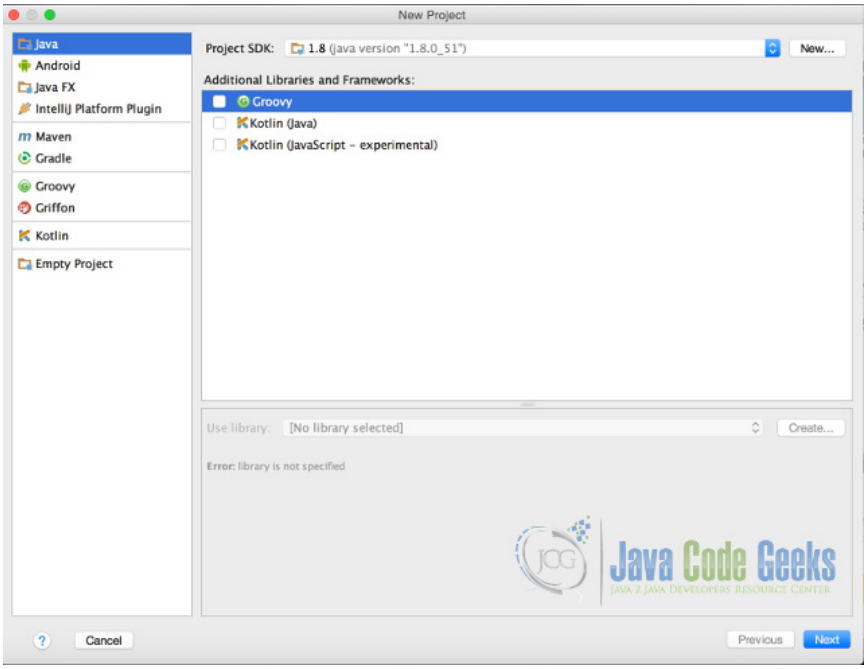


Figure 1.2: New Project

1.2.1 Project Type

When creating a new project in IntelliJ you have several project types to choose from as displayed in the left panel of the New Project dialog. When creating a standard Java project you will select **Java** as the project type. IntelliJ will then create for you the project structure for a typical Java application. The structure that is created by IntelliJ can still be updated to suit your own personal preferences.

In an instance where you want to use a build configuration tool like maven, you will then choose **Maven** as the project type. In this example we shall look at the Java project type.

1.2.2 Project SDK

Every project that you create will require a Software Development Kit (SDK). An SDK is a software platform of toolsets required to develop an application for a particular language, framework or environment. In the case of a java application you will require the Java Development Kit(JDK). As part of creating a new project, In the project dialog window you have to specify the SDK that will be used for a project. To define the SDK for the first time, Click → **New**, a dropdown menu appears where you can select the Home Directory of the installed JDK.

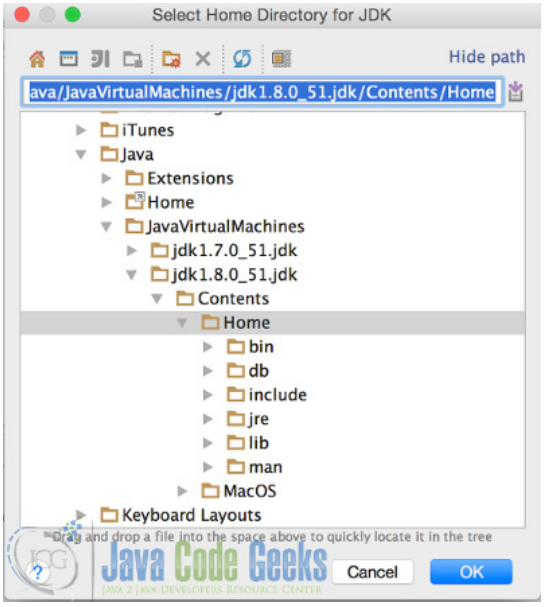


Figure 1.3: JDK Directory

1.2.3 Project Template

The next form allows you to use a predefined template to create your project. Click → **Create project from template**. Then select → Command Line App. The command line app gives you a simple Java application that includes a class with a `main()` method.

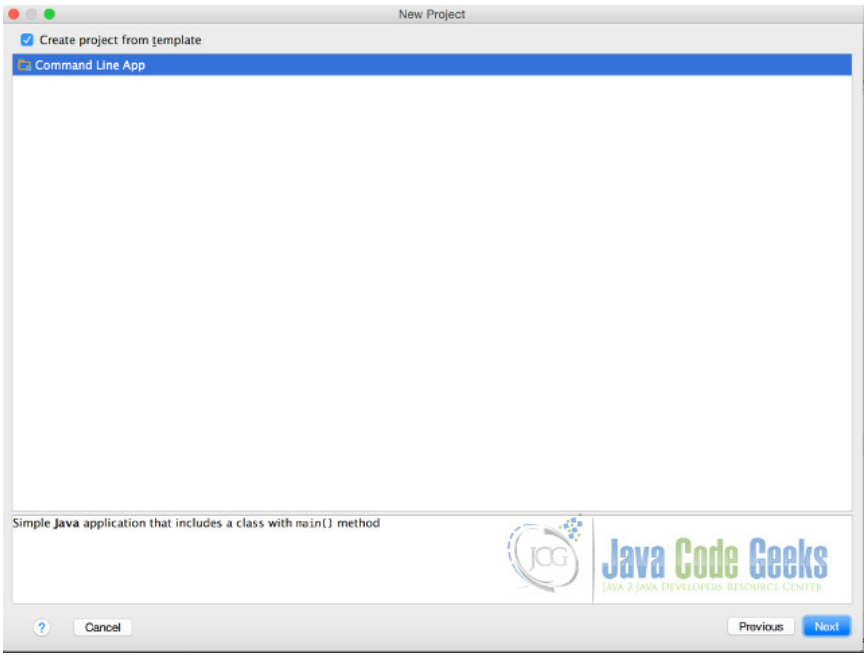


Figure 1.4: Project from template

The next form prompts for a project name. In this tutorial we shall implement the **String Calculator Kata** as an example of how you create a java project in IntelliJ.

Insert StringCalculator as the Project Name and also insert the location where your project files will be saved (if you want to change the default location). Specify the base package for the project. In this example we are going to use: **com.javacodegeeks.example**

Click → **Finish**

1.3 Project Directory Structure

It is always desirable to have a clear separation of application java source files and test files. The next set of instructions will achieve just that.

- Right-Click the src directory and select Mark Directory As → Unmark as Sources Root.
- Right-Click the src directory and select New → Directory.
- Enter **java** as the name of the new directory.
- Right-Click the src directory and select New → Directory.
- Enter **test** as the name of the new directory.
- Right-Click the java directory and select Mark Directory As → Sources Root.
- Right-Click the test directory and select Mark Directory As → Test Sources Root.

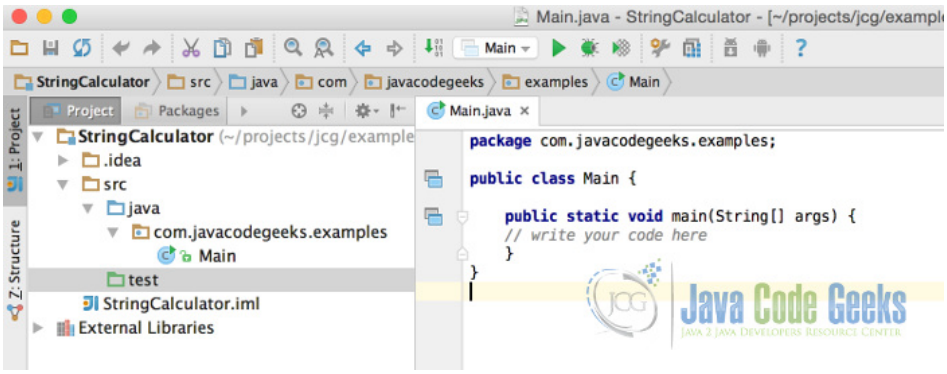


Figure 1.5: Project structure

1.4 Create New Class

- Right-Click **java** directory on the project view situated on the left panel
- Select New → Java Class
- A popup appears that prompts you to enter the name of the new class → Name: **StringCalculator**
- Click Ok

Implement the add method that takes an input of numbers in string format separated by delimiters that can take any form e.g.`1,2#3,4`. The method returns the sum of the input of numbers.

```
public class StringCalculator {
    public int add(String input) {
        int output = 0;
        String[] numbers = new InputNormalizer().normalize(input);
        InputValidator.validate(numbers);
        for (String num:numbers) {
            int numInt = Integer.parseInt(num);
            if(numInt < 1000) {
                output += numInt;
            }
        }
        return output;
    }
}
```

Add the InputNormalizer class. This class takes the input of numbers in string format and split them up based on the defined delimiters. The delimiters are defined at the beginning of the string, the start of the delimiters is marked by `//` and the end of the delimiters is marked by `n`. Each delimiter is enclosed within the open and closed brackets. e.g.`//[***][%]n1***2%3`

```
public class InputNormalizer {

    public String[] normalize(String input) {
        String delimiters = "\n|,";
        if(input.contains("//")) {
            delimiters = getDelimiters(input, delimiters);
            input = stripDelimiters(input);
        }
        return input.split(delimiters);
    }
```

```
    private String getDelimiters(String numbers, String defaultDelimiters) {
        StringBuilder delimiters = new StringBuilder(defaultDelimiters);
        while(numbers.indexOf("[") >= 0) {
            delimiters.append("|");
            String delimiter = numbers.substring(numbers.indexOf("[") + 1,numbers.indexOf("
                ]"));
            for(char delimiterChar:  delimiter.toCharArray()) {
                delimiters.append("\\").append(delimiterChar);
            }
            numbers = numbers.substring(numbers.indexOf("]") + 1);
        }
        System.out.println(" delimiters.toString() = " + delimiters.toString());
        return delimiters.toString();
    }

    private String stripDelimiters(String numbers) {
        return numbers.substring(numbers.indexOf("\n") + 1);
    }
}
```

Add the InputValidator class. This class validates that the numbers in a string are all positive numbers. If a negative number is found an exception is raised that displays all the negative numbers.

```
public class InputValidator {
    public static void validate(String[] numbers) throws RuntimeException {
        StringBuilder negativeNumbers = new StringBuilder();
        for (String number:numbers) {
            int numInt = Integer.parseInt(number);
            if(numInt < 0) { if(negativeNumbers.length()> 0) {
                    negativeNumbers.append(",");
                }
                negativeNumbers.append(numInt);
            }
        }
        if(negativeNumbers.length()> 0) {
            throw new RuntimeException(String.format("[%s] negative numbers not allowed.",
                negativeNumbers.toString()));
        }
    }
}
```

1.5 Run Application

To be able to run the application we implement the main class to call the SimpleCalculator class.

```
public class Main {
    public static void main(String[] args) {

        String numbers = "//[***][%]\n1***2%3";

        StringCalculator calculator = new StringCalculator();
        int output = calculator.add(numbers);
        System.out.println(" output = " + output);

    }
}
```

Right-Click anywhere in the Main source file and select → Run `Main`

1.6 Debugging an Application

Left-Click on the line where you want to add a breakpoint on the left pane of the source editor. A pale red circle will appear and the line will be highlighted in pale red.

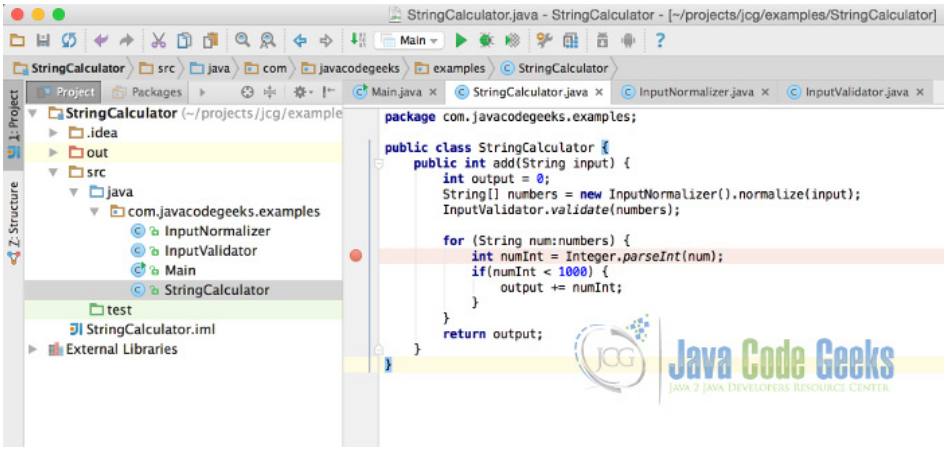


Figure 1.6: Debug

Open the main class, right click on it and select debug main. When the program is running it will stop on the line that has a breakpoint and the debug window will be opened that will allow you to interrogate your program at runtime.

1.7 Adding a test

When unit testing your application you will typical use a testing framework like JUnit. In order to use JUnit in our application we will need to add the JUnit Library. Download the JUnit Library. In this example we are going to use version 4.12

Select→ File → Project Structure → Libraries - > New Project Library → Java

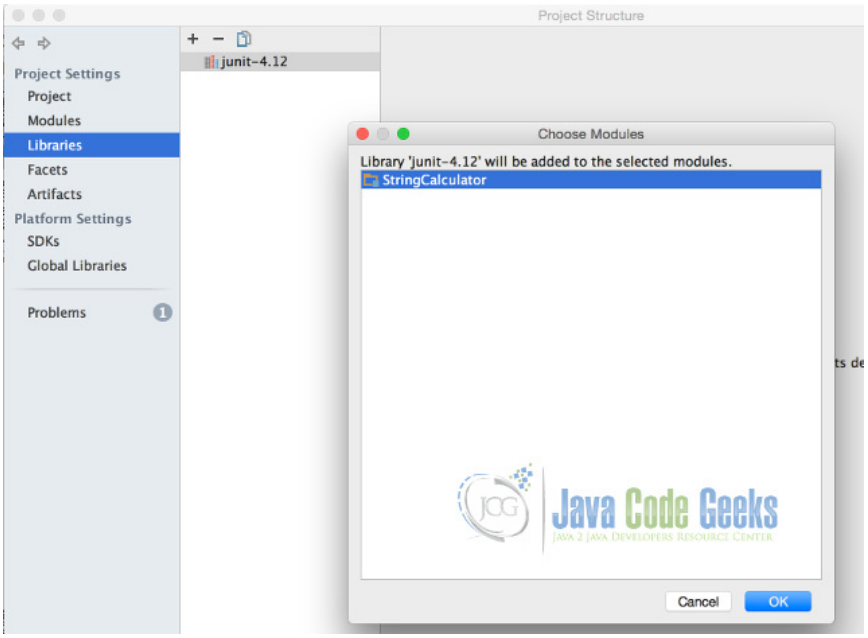


Figure 1.7: Add library

Locate JUnit library in the directory where it has been downloaded to and add it to the StringCalculator module. JUnit 4.12 has a dependency on hamcrest-core in this example we are using version 1.3. Add the hamcrest-core jar following the same steps used for adding the JUnit library.

In the test directory of the StringCalculator Project add new class:

```
package com.javacodegeeks.examples;

import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class StringsCalculatorTest {

    StringCalculator calculator;

    @Before
    public void init() {
        calculator = new StringCalculator();
    }

    @Test
    public void testDefaultDelimiters() {
        String input = "1\n2,3";
        assertEquals(6,calculator.add(input));
    }

    @Test
    public void testMultiDelimiters() {
        String input = "//[***][%]\n1***2%3";
        assertEquals(6,calculator.add(input));
    }

    @Test
    public void testNumbersMoreThan1000() {
        String input = "//[***][%]\n1***2%3,2000";
        assertEquals(6,calculator.add(input));
    }
}
```

1.8 Download the IntelliJ Project

This was an example of using IntelliJ IDEA to create the String Calculator.

Download You can download the full source code of this example here: **Intellij IDEA Example**

Chapter 2

How to Install IntelliJ IDEA on Ubuntu Linux

Some years ago, Eclipse was probably the king of the IDEs of Java development. But, in the last times, IntelliJ IDEA, from JetBrains (developers of WebStorm, PhpStorm or TeamCity, among many other IDEs and software engineering tools), has become very popular, being considered by many developers the best Java IDE nowadays. This tutorial will show how to install it on Linux.

For this tutorial, Linux Mint 18 (an Ubuntu based Linux distribution) and IntelliJ IDEA 2016.2.5 have been used.

2.1 Download

We can download IntelliJ IDEA from the official JetBrains site. As you can see, JetBrains offers two editions: Community Edition, free and open source (GitHub repository); and Ultimate Edition, proprietary and non-free (check prices).

In this tutorial, we will use the Community Edition. You can follow the direct download link.

After the download, we should verify the file checksum, to check that the download has been made correctly and that the file is not corrupted. In this case, for the 2016.2.5 version, the SHA-256 checksum is the following:

```
1bccc6689c5a140cc8e3b6496a07c7c7c2d2b403034aaff98ace0c9badd63734
```

(Got from IntelliJ IDEA).

For checking the checksum of the file we just have downloaded, we have to open the terminal, navigate to the directory of the download, and execute the `sha256sum` command:

```
cd path/to/download
sha256sum ideaIC-2016.2.5.tar.gz
```

In this case, we got the same checksum, so we can proceed with the installation. If we would have got a distinct checksum, we would have to download it again.

2.2 Installation

Of course, the first step is to extract the downloaded compressed file:

```
tar -zxvf ideaIC-2016.2.5.tar.gz
```

Now, we have to execute the installation script (`idea.sh`) inside the `bin/` directory of the directory where the files have been extracted. So, before, we have to give execution permissions to the script:

```
cd idea-IC-162.2228.15
sudo chmod a=+rx bin/idea.sh
bin/idea.sh
```

Note: if you want to install IntelliJ IDEA for all the user, you must execute the script with superuser privileges:

```
sudo bin/idea.sh
```

If everything went correctly, the installation window should appear.

Figure 2.1: Initial Window

If you have a previous IntelliJ IDEA installation and you want to preserve its configuration, you can choose the first option, specifying the configuration folder of the version. If not, just leave the second option.

Now, you can choose your favorite theme for the IDE. The theme does not affect to IntelliJ IDEA behavior.

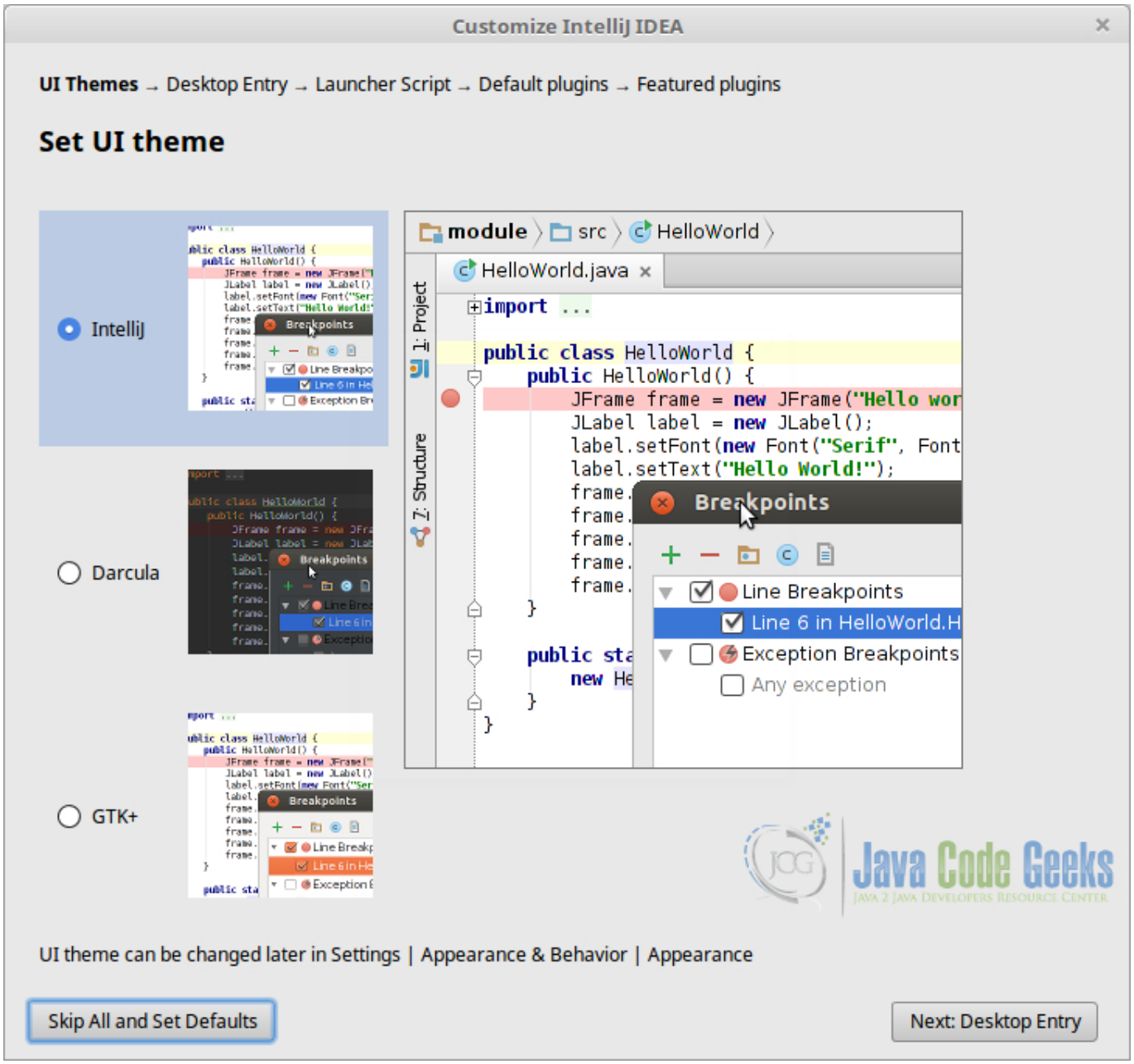


Figure 2.2: Select Theme

The next window is for selecting if we want to integrate IntelliJ IDEA with system application menu. In almost every case **the answer will be yes, otherwise, for starting IntelliJ IDEA, we would have to manually execute the `idea.sh` script**, which is not the most comfortable way.

Figure 2.3: Integrate in Menu

The launcher creation script. . .

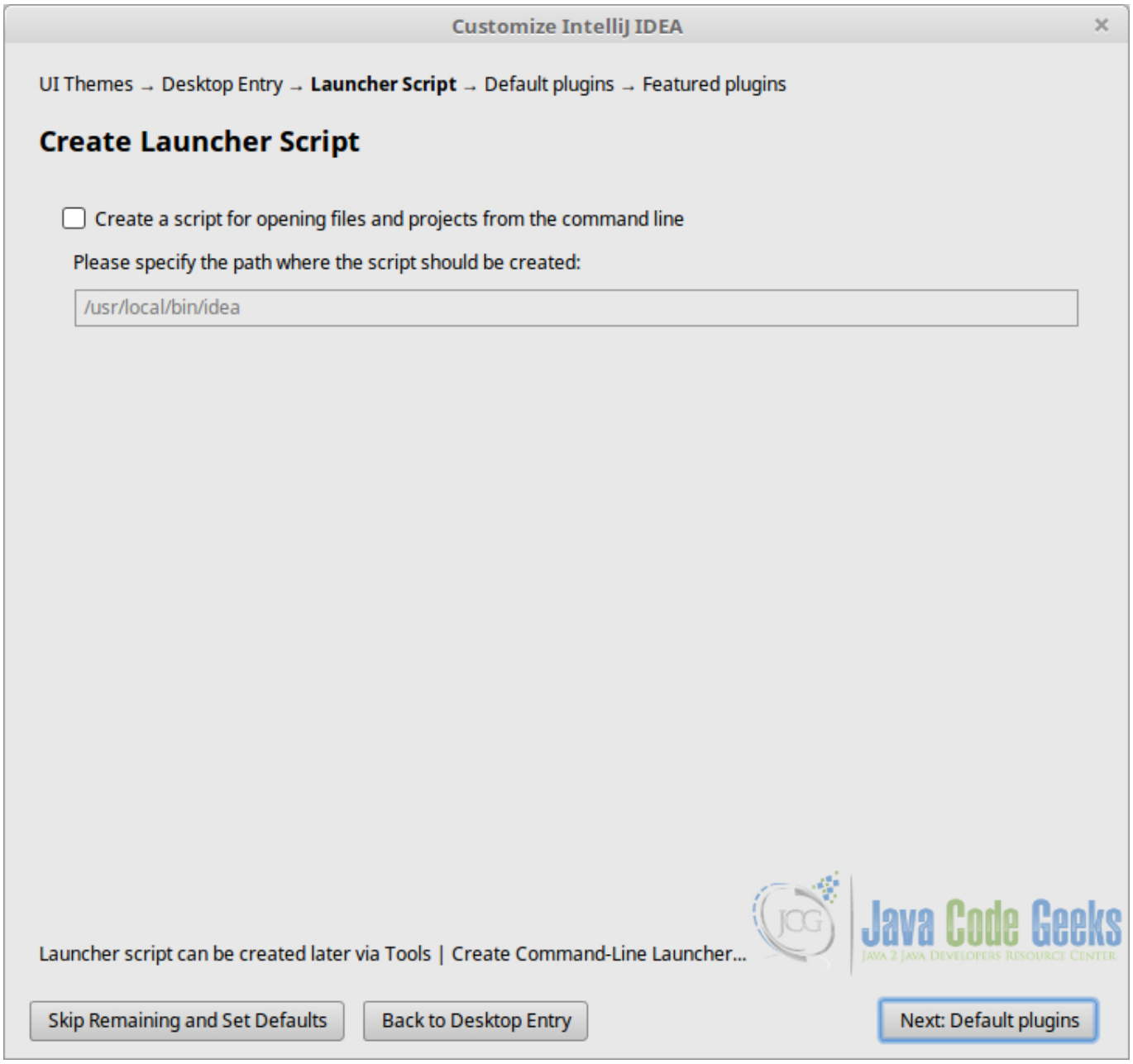


Figure 2.4: Launcher Script

In the following window, we can choose some useful tools, including build tools (Apache Ant, Maven, Gradle), version controlling tools (of course Git, and Mercurial, Subversion, etc.), and many others, as shown in the following image. Don't worry if you disable something and later you want to use it; you can change the configuration later.

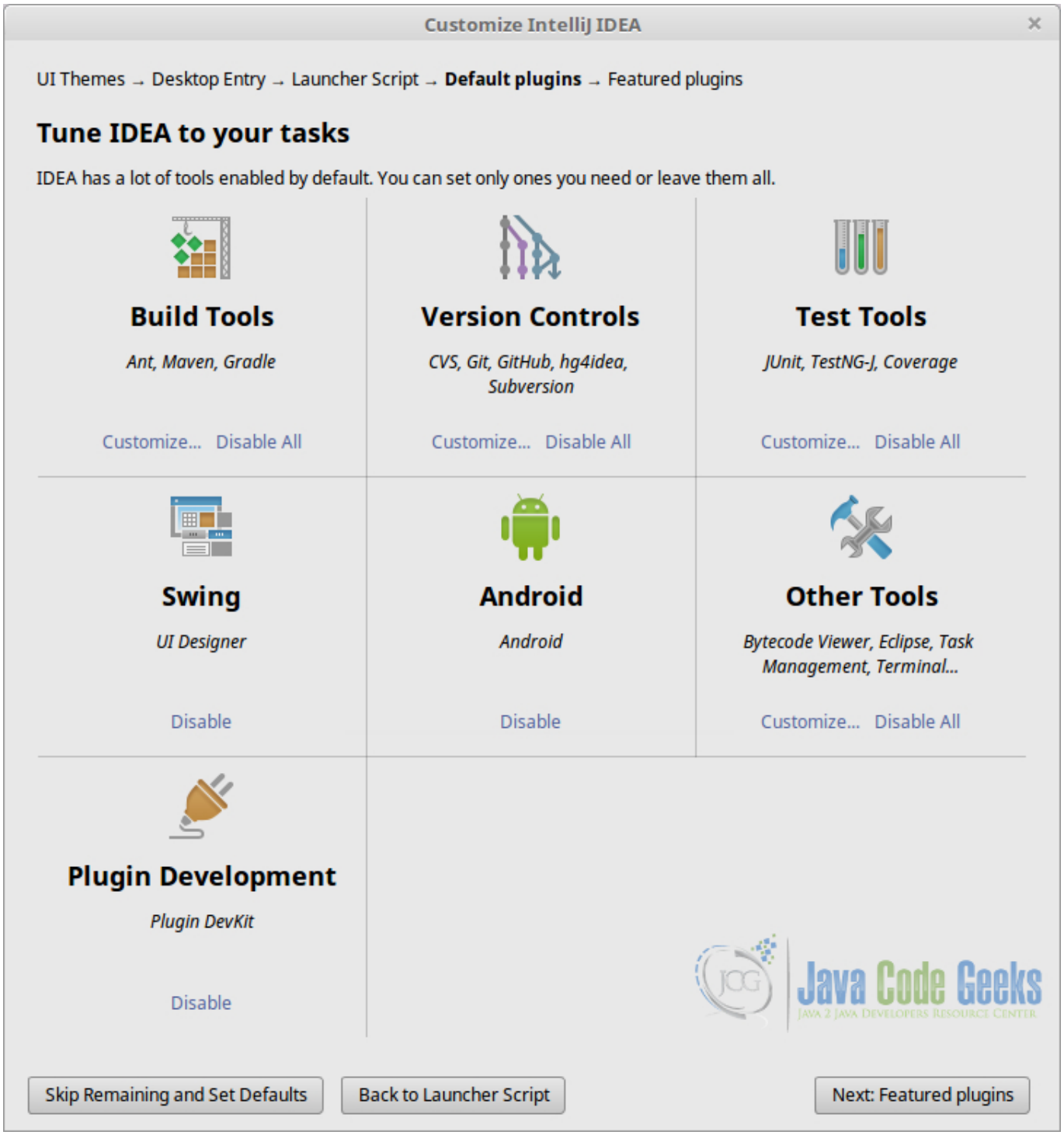


Figure 2.5: Tune IntelliJ IDEA

The last window shows two featured plugins. One is for Scala developing, and the other for emulating the Vim editor in IntelliJ IDEA. Of course, you can install these, and many more plugins, whenever you want.

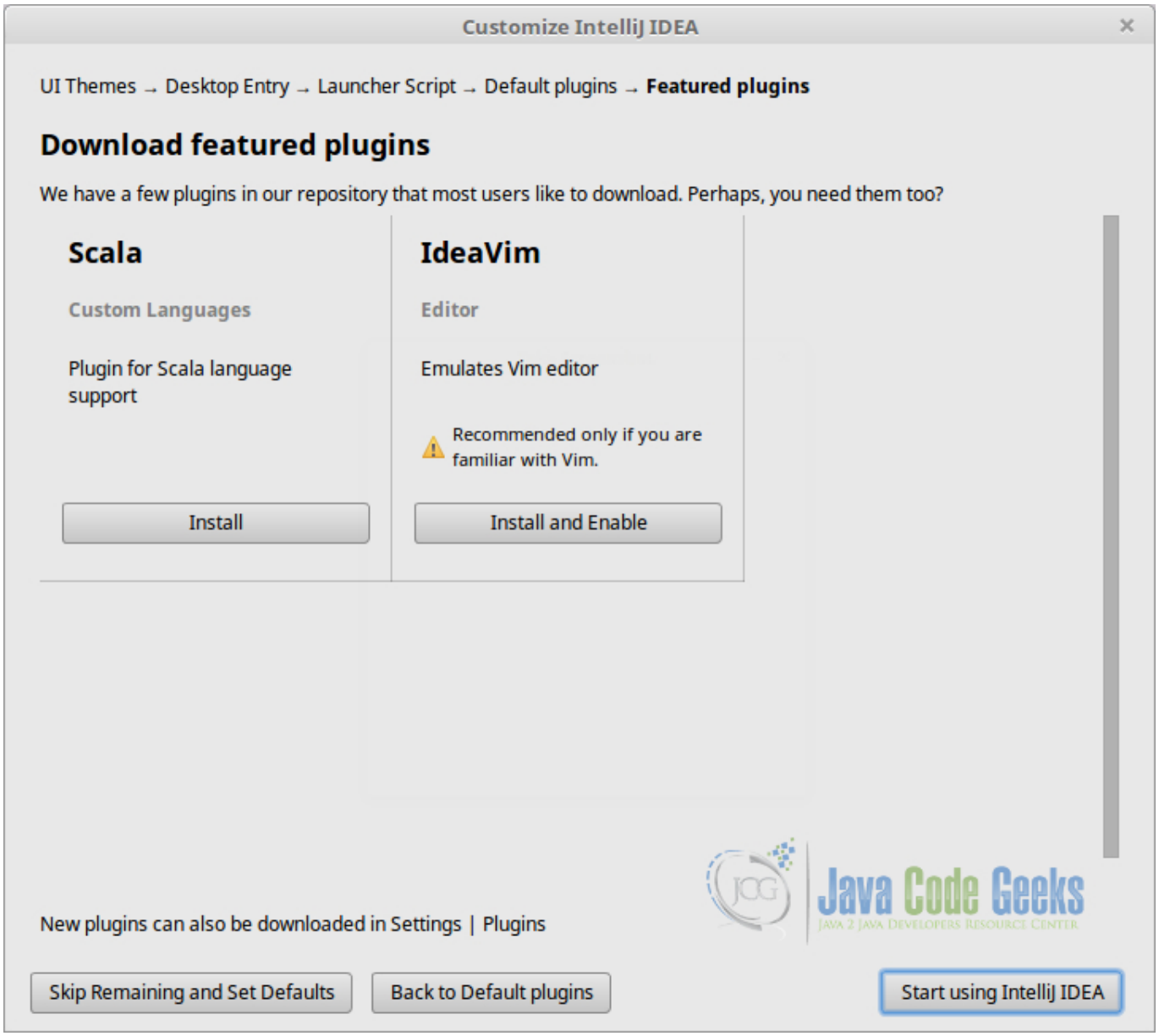


Figure 2.6: Featured Plugins

After this step, we finally can start IntelliJ IDEA!

2.3 Become an IntelliJ IDEA expert!

Check out Java Code Geeks' tutorials about IntelliJ IDEA:

- IntelliJ IDEA Format Code Example: an example about formatting the code, in many aspects.
- IntelliJ IDEA Create Test Tutorial: about creating JUnit tests with IntelliJ IDEA.
- IntelliJ IDEA Keyboard Shortcuts Tutorial: useful shortcuts for the IDE to save time.
- Top 10 IntelliJ IDEA plugins: a top of most useful plugins (among many) for IntelliJ.

- One and the Only One Reason to Customize IntelliJ IDEA Memory Settings: about advanced settings regarding the memory usage.
- IntelliJ IDEA Color Schemes / Themes Configuration: for customizing IntelliJ.

2.4 Summary

This tutorial has shown you how to download and install IntelliJ IDEA Java IDE on Linux systems. We have seen how to install it step by step, understanding the meaning and implications of each one. To end up, we have seen a list of other JCG posts to dive into IntelliJ IDEA, once successfully installed.

Chapter 3

Top 10 IntelliJ IDEA Plugins

IntelliJ IDEA has inspired many Java developers to write plug-ins, from J2EE to code editing tools to games. Now it has a robust plugin ecosystem with more than 1500 available plugins and new ones appearing nearly every week. In this article we are going to present the 10 most useful to our point of view plugins for any developer using this IDE.

3.1 Shifter

Detects type of selection, line or keyword at caret and shifts it "up" or "down" on keyboard shortcut. If there's only one shiftable word in a line, it can be shifted without the caret touching it. Lowercase/uppercase or lower case with upper first character of shifted words is maintained.

Default keyboard shortcuts:

- Ctrl+Shift+Alt+Comma : Shift Down
- Ctrl+Shift+Alt+Period : Shift Up
- Ctrl+Shift+Alt+K : Shift Up More
- Ctrl+Shift+Alt+J : Shift Down More

Shift more: repeats shifting the selected value multiple times. The amount of repetitions can be configured in the plugin configuration (default: 10).

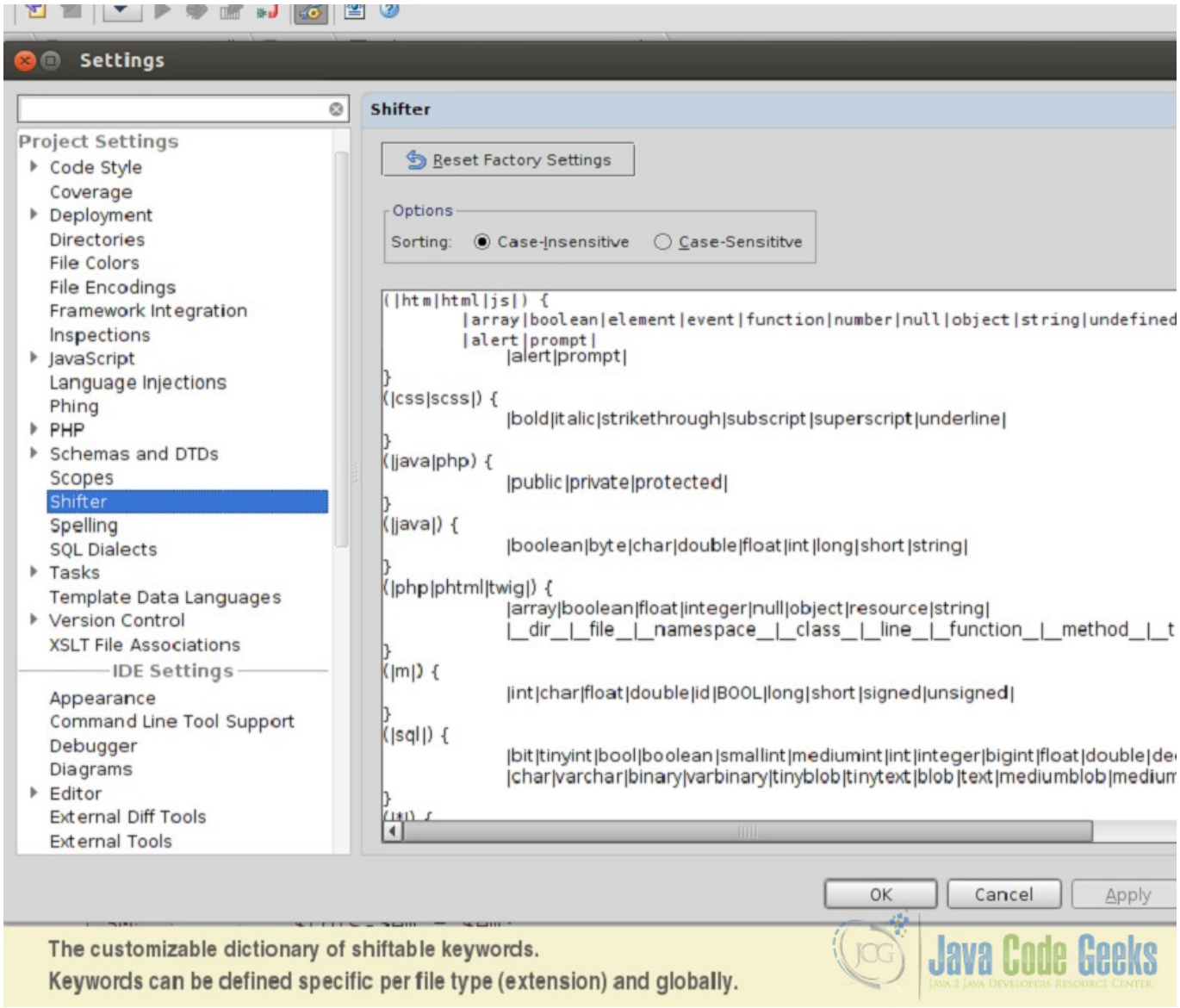


Figure 3.1: Shifter Intellij Plugin

3.2 BrowseWordAtCaret

Allows to easily browse next/previous word at caret and highlight other appearances of selected word. **Usage:** Browse with CTRL-ALT-UP, CTRL-ALT-DOWN (note: on default-keymap this shortcut is also for next/previous occurrence).

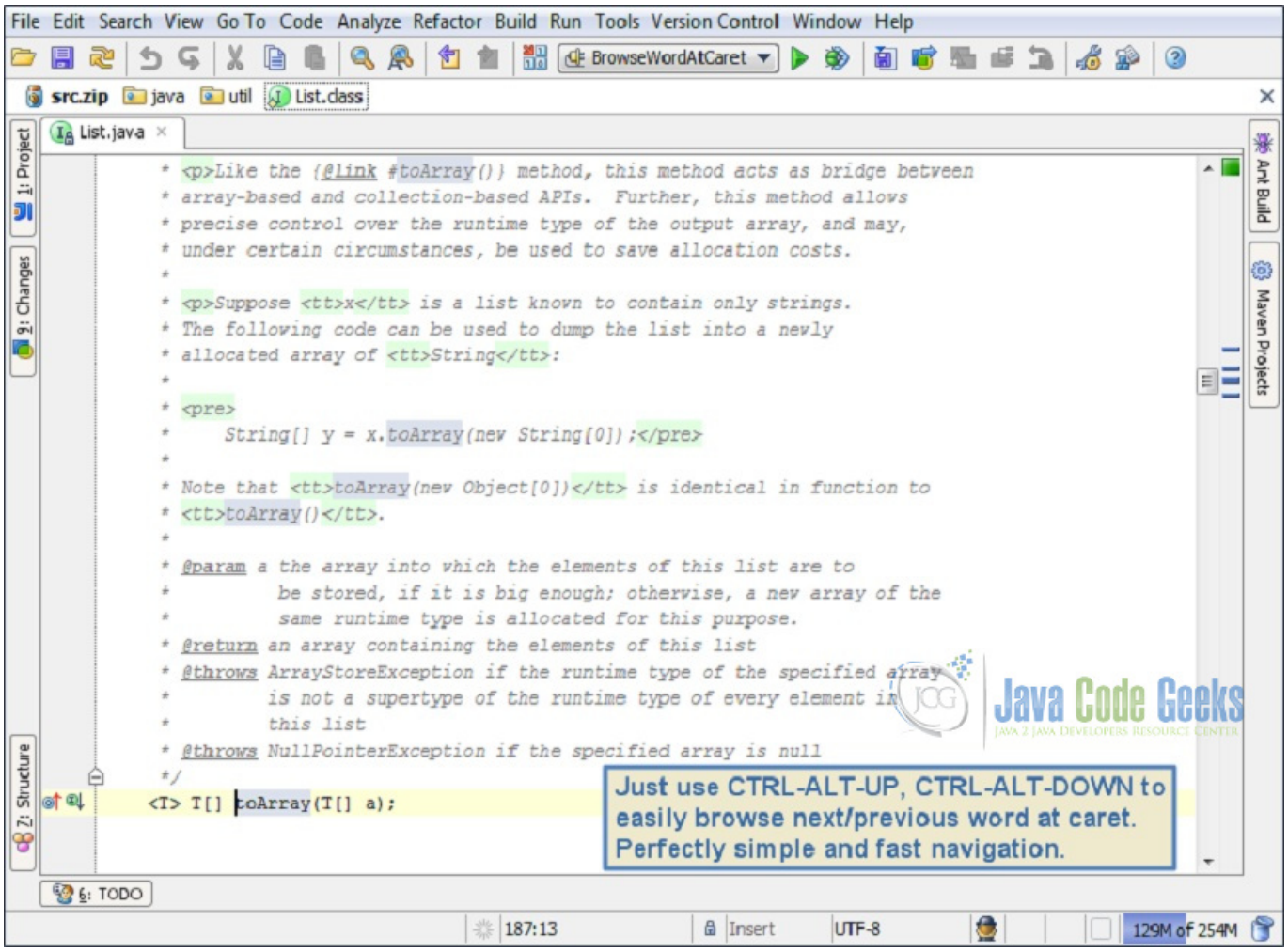


Figure 3.2: BrowseWordAtCaret Intellij Plugin

3.3 BashSupport

Bash language support for IntelliJ. It supports syntax highlighting, rename refactoring, documentation lookup, inspections, quick-fixes. BashSupport can directly run scripts within IntelliJ. You can create a new run configuration for Bash scripts. Here you can set which interpreter is used to run it. Whenever a script is executed the output is logged. If Bash prints out syntax errors then the erroneous lines are clickable to jump to the location of the error.

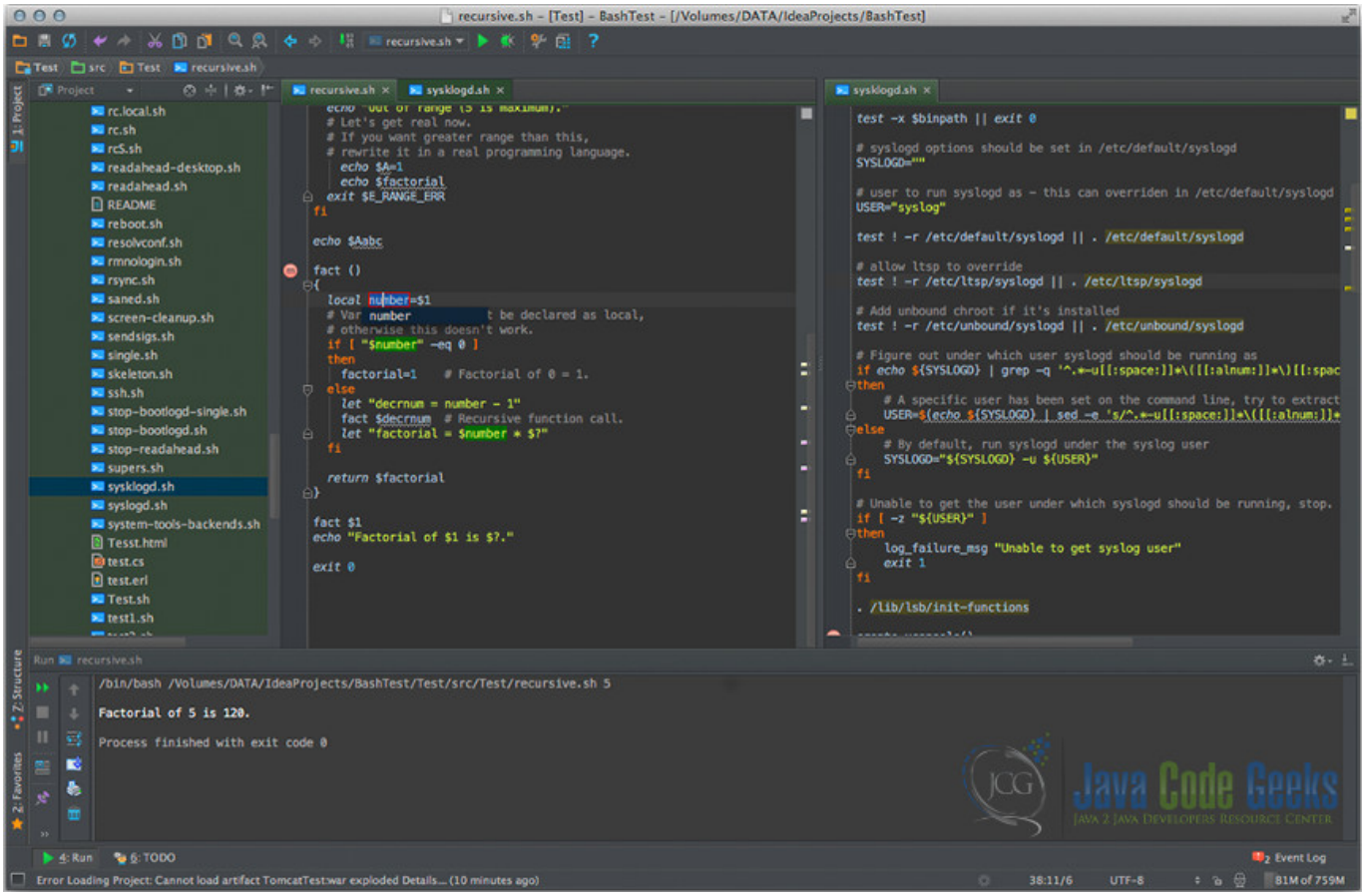


Figure 3.3: BashSupport Intellij Plugin

3.4 IdeaVim

IdeaVim is a Vim emulation plug-in for IDEs based on the IntelliJ platform. Use the IDE's plugin manager to install the latest version of the plugin. Start the IDE normally and enable the Vim emulation using "Tools | Vim Emulator" menu item. At this point you must use Vim keystrokes in all editors.

If you wish to disable the plugin, select the "Tools | Vim Emulator" menu so it is unchecked. At this point IDE will work with its regular keyboard shortcuts. Keyboard shortcut conflicts between the Vim emulation and the IDE can be resolved via "File | Settings | Vim Emulation", "File | Settings | Keymap" and key mapping commands in your ~/.ideavimrc file.

3.5 LiveEdit

It allows you to view your changes in an actual browser window instantly without a page refresh, reload corresponding browser pages (related to opened file in IDE) and highlight corresponding element. Currently only Google Chrome is supported (extensions for Firefox and Safari are still postponed). If you edit your PHP file or Smarty template, it is so boring to switch to the browser and reload tab manually, isn't it? Forget about it! Just View → Reload in Browser. Want to see your change instantly? No problem. And yes, SASS/CoffeeScript are supported (external watching compiler is still needed). CSS and JavaScript (Kotlin, CoffeeScript and any other compilable to JS) will be hot-swapped without page refresh (of course, hot-swapped JavaScript will be effective only if it used in cycle or event-driven).

3.6 Maven Helper

Provides actions to run/debug the current test file. If maven-surefire-plugin is configured to skip or exclude the test, *verify* goal will be used. It also provides actions to run/debug maven goals for a module that contains the current file and an easy way to find and exclude conflicting dependencies

Usage:

- Right click in Editor | Run Maven
- Right click in Project View Toolbar | Run Maven
- CTRL + ALT + R - "Quick Run Maven Goal" action
- Customize goals: Settings | (Other Settings) | Maven Helper
- Define shortcuts: Settings | Keymap | Plug-ins | Maven Helper

Open pom file, click on *Dependency Analyzer* tab, right click in the tree for context actions.

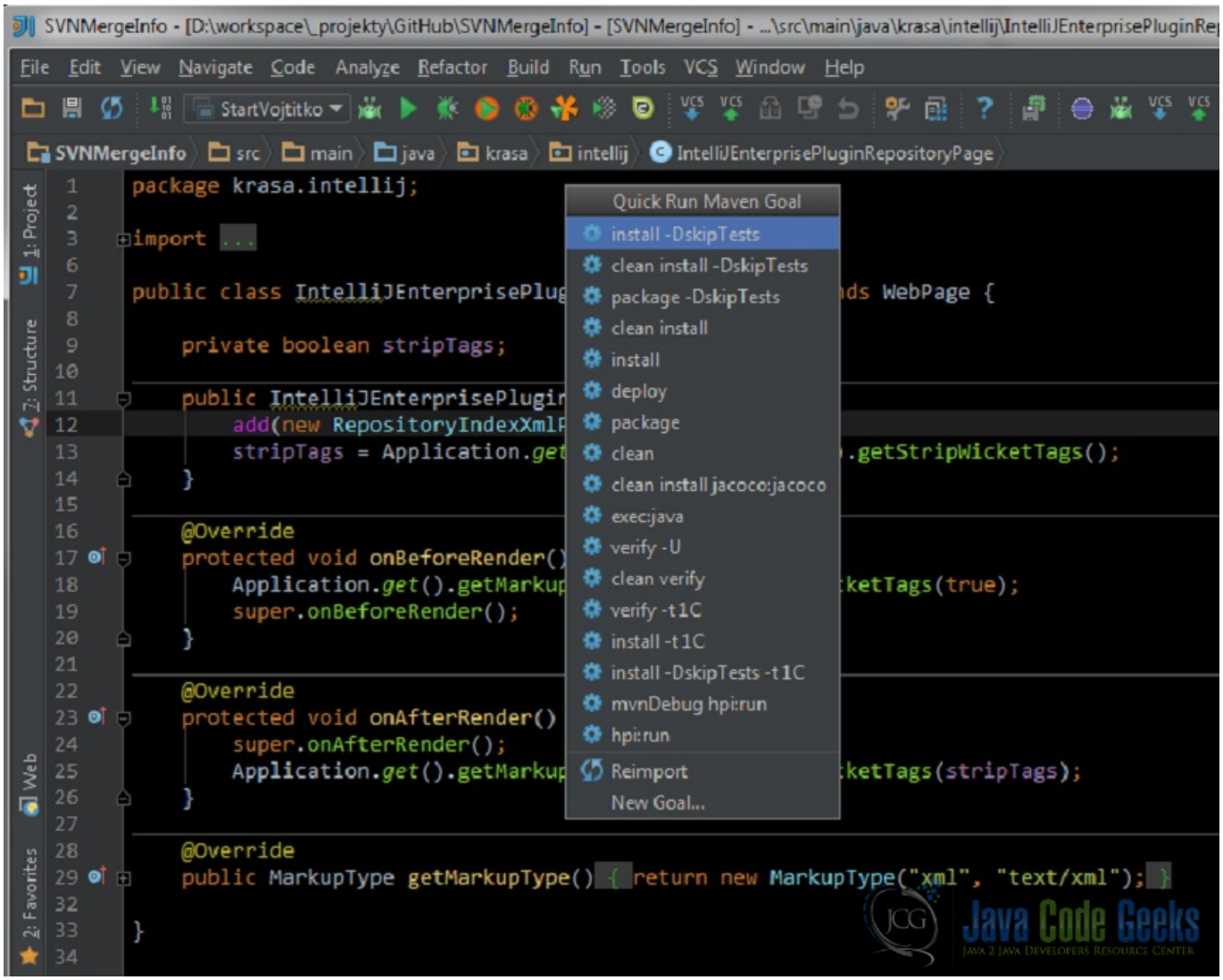


Figure 3.4: Maven Helper Intellij Plugin

3.7 String Manipulation

Provides actions for text manipulation:

- Toggle style (camelCase, hyphen-lowercase, HYPHEN-UPPERCASE, snake_case, SCREAMING_SNAKE_CASE, dot.case, words lowercase, Words Capitalized, PascalCase)
- To SCREAMING_SNAKE_CASE (or to camelCase)
- To snake_case (or to camelCase)
- To dot.case (or to camelCase)
- To hyphen-case (or to camelCase)
- To hyphen-case (or to snake_case)
- To camelCase (or to words)
- To PascalCase (or to camelCase)
- Capitalize selected text (when nothing is selected, then either nothing or whole line or a particular element is selected) report an issue if you find something to improve, each file type needs its own implementation to work flawlessly.
- Un/Escape:
- Un/Escape selected Java text
- Un/Escape selected JavaScript text
- Un/Escape selected HTML text
- Un/Escape selected XML text
- Un/Escape selected SQL text
- Un/Escape selected PHP text
- Convert non ASCII to escaped Unicode
- Convert escaped Unicode to String
- Encode/Decode:
- Encode selected text to MD5 Hex16
- De/Encode selected text as URL
- De/Encode selected text to Base64 Other:
- Format selected text to columns/table by a chosen separator/delimiter
- Increment/decrement all numbers found.
- Duplicate line and increment/decrement all numbers found.
- Trim selected text
- Trim all spaces in selected text
- Remove all spaces in selected text
- Remove duplicate lines
- Remove empty lines
- Swap characters at caret
- Grep selected text, All lines not matching input text wil be removed. (Does not work in column mode)
- Actions are available under Edit menu, or via the shortcut "alt M" and "alt shift M". You can setup your own shortcuts for better usability.

3.8 SQL Query Plugin

A tool for executing SQL statements through a JDBC connection.

Features:

- Syntax highlighting
- Executing multiple statements at once
- Highlighting of primary and foreign keys
- Editing of results
- Support for easier configuration of common JDBC driver
- Support of different column formats (including custom formats)
- Browser for database structure
- Statement templates for often used statements
- Support for Java BLOB's
- Export into XML, HTML, CSV and Excel
- Data Load

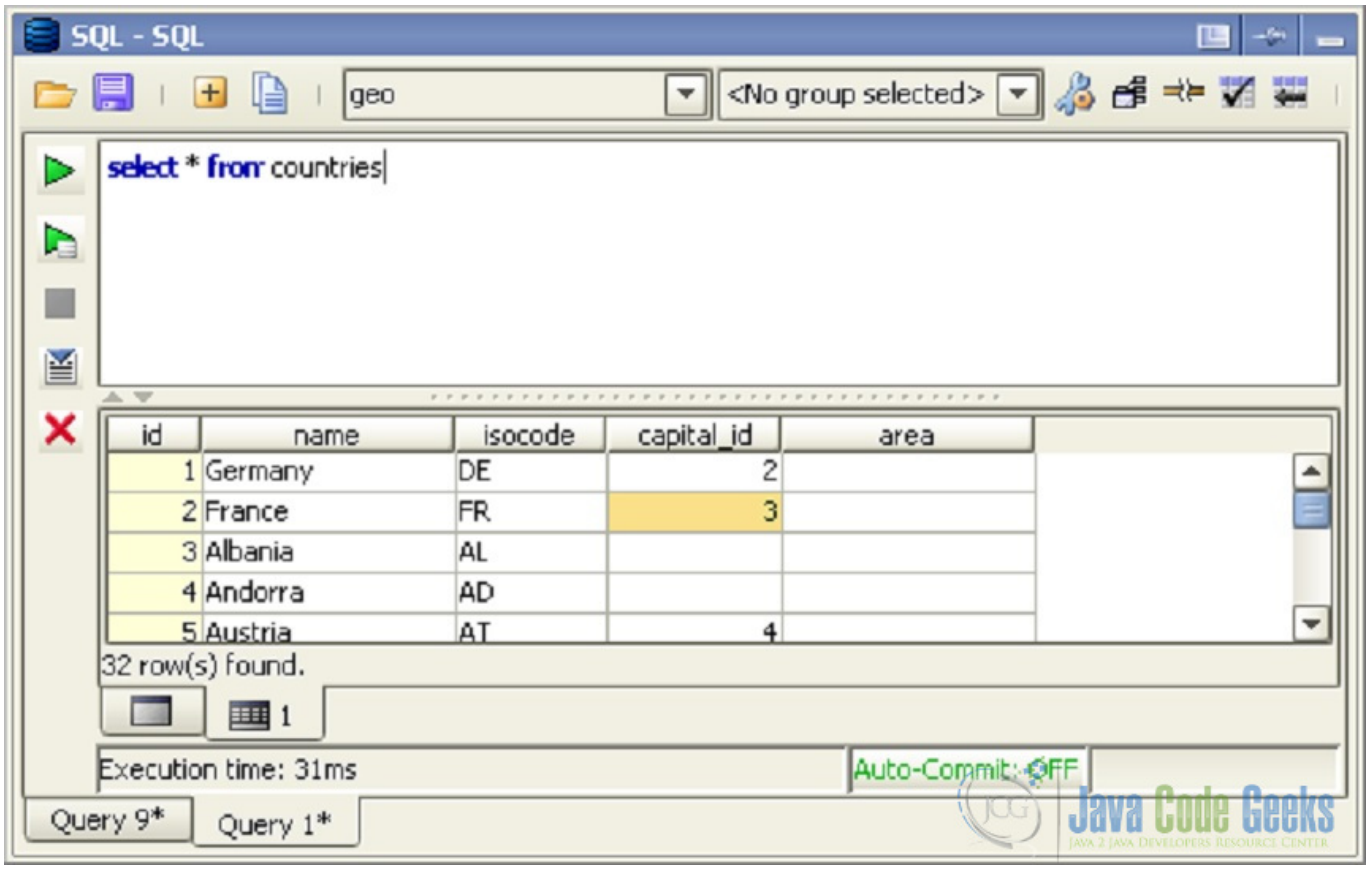


Figure 3.5: SQL Query Intellij Plugin

3.9 JRebel for IntelliJ

JRebel is a productivity tool that allows developers to reload code changes instantly. It skips the rebuild, restart, and redeploy cycle common in Java development. JRebel enables developers to get more done in the same amount of time and stay in the flow while coding. JRebel supports a majority of real-world enterprise java stacks and is easy to install into existing development environments. The JRebel plugin for IntelliJ IDEA includes the JRebel agent and helps you to automatically generate the JRebel configuration. It enables JRebel for applications and servers with a single click and improves the debugging support in the IDE.

3.9.1 External servers and JRebel Cloud/Remote

When launching the application server from a command line or using a remote server, open **Settings > JRebel** and select **Startup**.

- **Run via IDE** - This displays the server launch instructions from within the IDE.
- **Run using CLI** - Select this option to access JRebel's integrated CLI instructions. You will be presented with a set of drop-down menus. Select your desired Java version and operating system to receive a set of pre-configured instructions. Follow the steps provided to start your external, command line server with JRebel enabled.
- **Run on a remote server** - Select this option to configure JRebel in a JRebel Cloud/Remote setting. Follow the steps provided.

3.10 Grep Console

Allows you to define a series of regular expressions which will be tested against the console output or file. Each expression matching a line will affect the style of the entire line, or play a sound. For example, error messages could be set to show up with a red background.

Additional Features: ANSI colouring

File Tailing:

- Main menu | Tools | Open File in Console
- Main Menu | Help | Tail (IntelliJ) Log in Console
- Drag&Drop "Tail File" panel in main toolbar
- http and Windows context menu integration

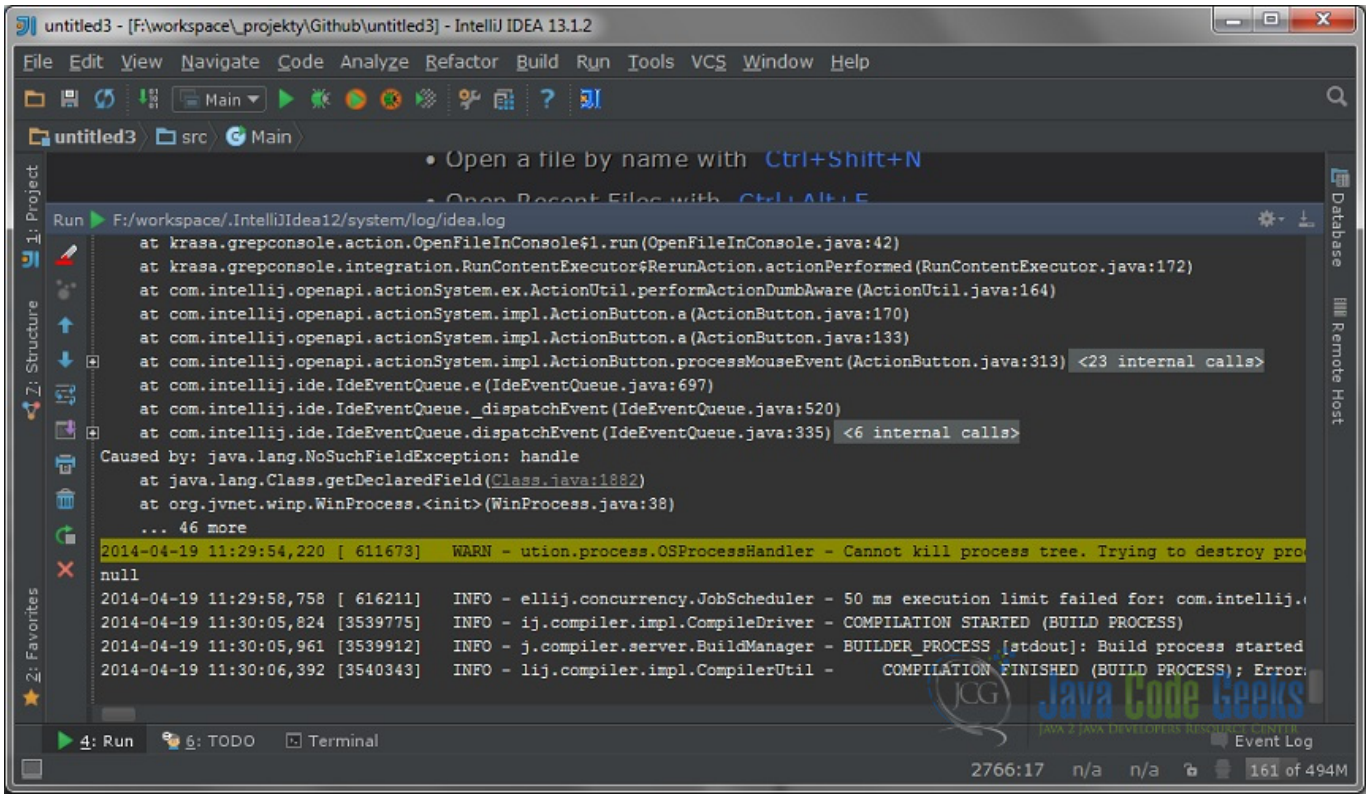


Figure 3.6: Grep Console Intellij Plugin

3.11 Conclusion

So this is list of experiences, tips and thoughts about the best IntelliJ IDEA plugins out there. In this article we tried to present some of the best choices for you developers in order to make the development process more convenient. I hope you find it useful and by using them or another plugin of your choice, become even more productive and deliver code faster and in better quality.

Chapter 4

IntelliJ GUI Designer Example

This example demonstrates how you can utilize IntelliJ GUI Designer to create a graphical user interface that is based on the Swing library components. It must be noted that IntelliJ does not support modeling of non-swing components.

When working with the GUI Designer you operate in design view where you are able to drag and drop swing components within the context of a form. All GUI information related to the form are stored in a file with .form extension. When creating a GUI component in IntelliJ you begin by creating a form, this form serves as a container that groups all other components that are required for your application. When a form is created, IntelliJ provides you with an option of also creating a Bound Class. A Bound Class is a Java class that is bound to a form and contains auto-generated code that reflects the components that are added on the form. The bound class can be updated at any point to add specific behavior on the components that have been added in the form.

In this example we shall create a simple calculator application to demonstrate how you work with the GUI Designer.

4.1 Creating a new project

Launch IntelliJ and create a new project called: **CalculatorGUI**.

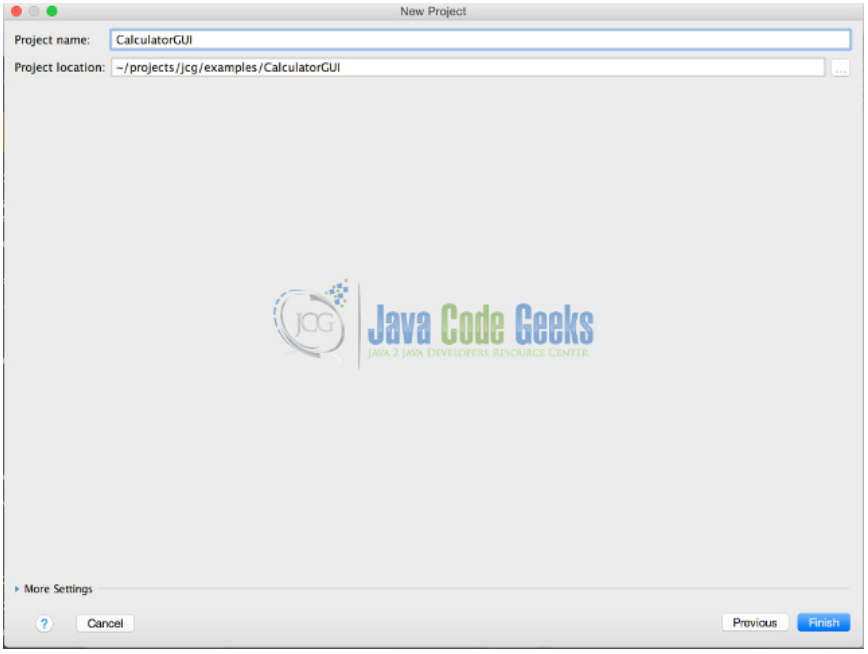


Figure 4.1: New Project

Create a new package in the src/java directory called **com.javacodegeeks.example**

Right-click the new package and select → **New** → **GUI Form**

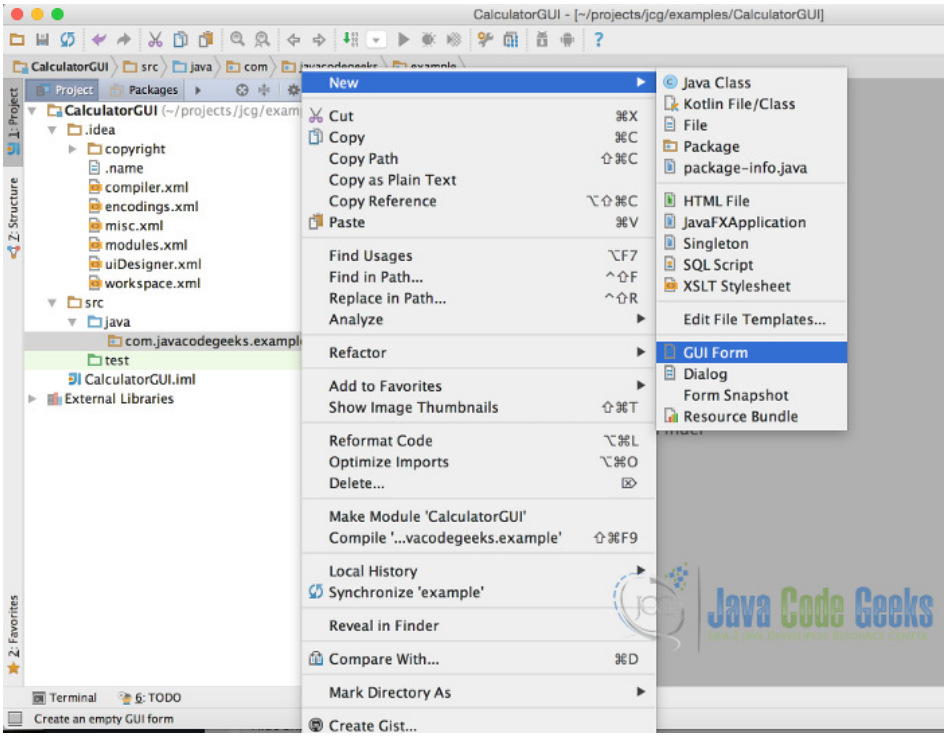


Figure 4.2: Create GUI Form

In the New GUI Form Window populate the following:

- **Form Name**: Calculator
- **Base layout manager**: GridLayoutManager(IntelliJ)
- **Create bound class**:selected
- **Class name**: Calculator

Clicking Ok on the GUI Form window once its populated, should display design-time graphical view as seen below:

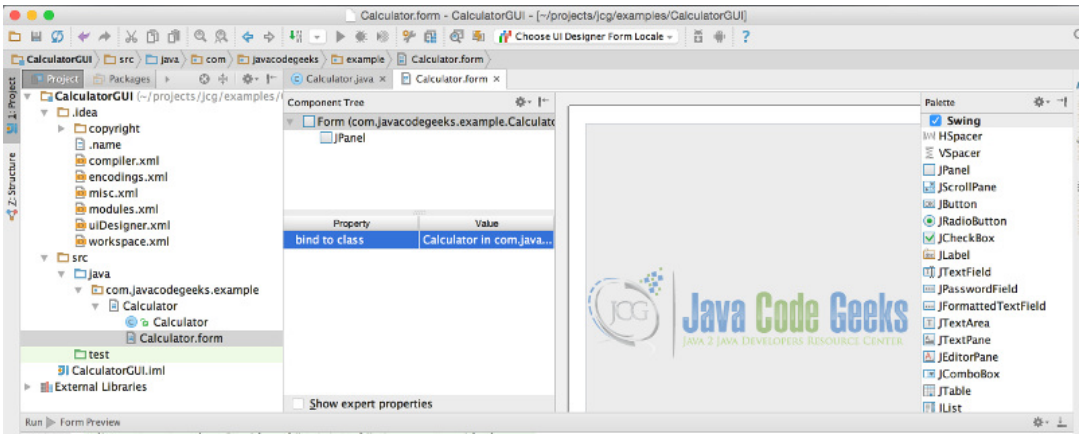


Figure 4.3: Design view

4.2 Graphical view

The graphical view allows you to drag and drop swing components to be added on the Form of the application. When any file that has a `.form` extension has been selected into the view, the following windows are displayed:

- Component tree - displays the hierarchical view of the components that have been added on the form.
- Property editor - displays the properties of the components added on the form which can be edited.
- Design area - the graphical view of the form. Components are dragged into this area to be added on the form.
- Palette - contains all available swing components that can be selected to be added on the form.

4.3 Creating the GUI

4.3.1 Add Results display

Drag the `JTextField` from the palette and drop it in the design area. Update the field name in the property editor to read: **results**

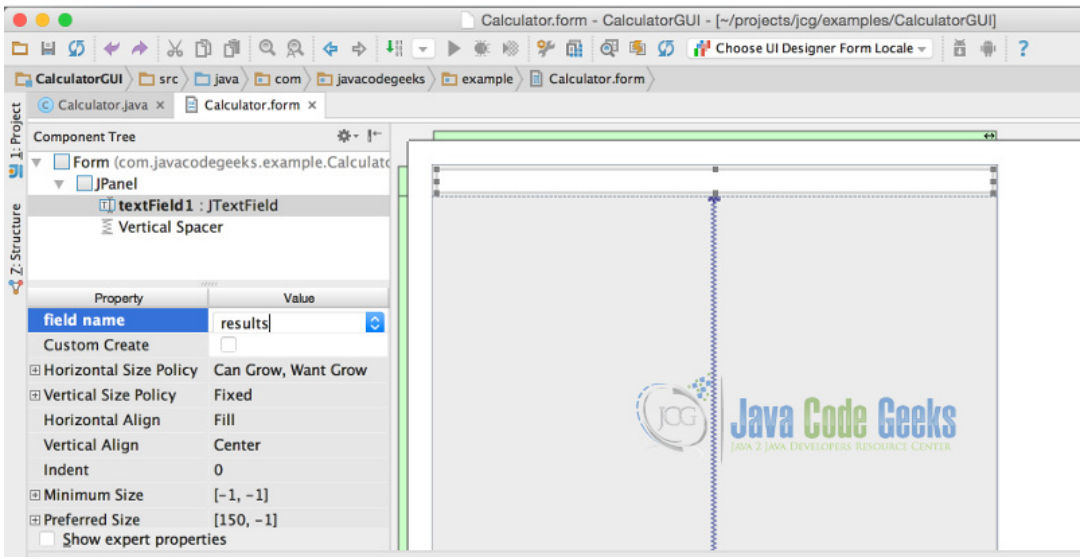


Figure 4.4: Adding JTextField

When you add the `JTextField` the `Vertical Spacer` also gets automatically added.

4.3.2 Add Buttons

Drag the `JButton` from the palette and drop it in the design area. Add the JButton on the left-hand side of the `Vertical Spacer`. As you release the mouse a tooltip will be displayed showing `JPanel(Row 1, Before Col 0)`, which indicates the position where the component will be placed in the grid. Repeat the process to add 4 buttons in the same row. Update the properties of the 4 buttons to the following:

- **button1**: field name change to **clearBtn**, Text change to **AC**
- **button2**: field name change to **signBtn**. Text change to **+/-**
- **button3**: field name change to **percentBtn**.Text change to **%**

- **button4**: field name change to **divideBtn**. Text change to **/**

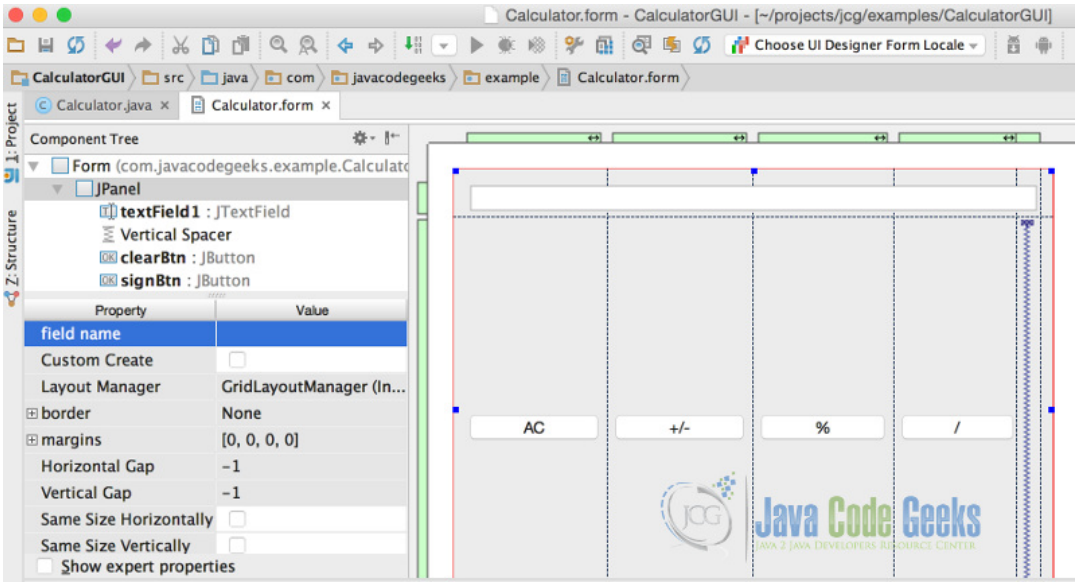


Figure 4.5: Add Buttons

Add the rest of the buttons, in total should have 5 rows and 4 columns populated with buttons. You can now remove the `Vertical Spacer`.

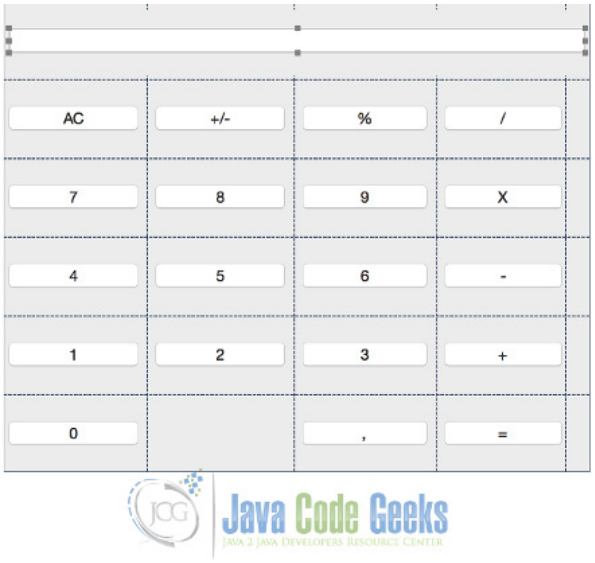


Figure 4.6: Calculator design view

Select the `JPanel` in the Component tree of the form view and update the field name property to `calculatorView`. Clicking on the `Calculator.java` should now have the following fields inserted:

Calculator.java

```
package com.javacodegeeks.example;

import javax.swing.*;
```

```
public class Calculator {
    private JTextField resultsTxt;
    private JButton clearBtn;
    private JButton signBtn;
    private JButton percentBtn;
    private JButton divideBtn;
    private JButton sevenBtn;
    private JButton eightBtn;
    private JButton nineBtn;
    private JButton multiplyBtn;
    private JButton fourBtn;
    private JButton fiveBtn;
    private JButton sixBtn;
    private JButton minusBtn;
    private JButton oneBtn;
    private JButton twoBtn;
    private JButton threeBtn;
    private JButton addBtn;
    private JButton zeroBtn;
    private JButton equalBtn;
    private JButton digitBtn;
    private JPanel calculatorView;
}
```

4.4 Making the form functional

In order for the form to be functional it requires a runtime frame to be created. We will create the `main()` method that will will be responsible for creating and disposing the runtime frame.

In the code editor of `Calculator.java` file select → Generate... → Form `main()`

The following code gets generated:

Calculator.java main method

```
public static void main(String[] args) {
        JFrame frame = new JFrame("Calculator");
        frame.setContentPane(new Calculator().calculatorView);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
```

Now we create an enum for handling the different calculator operations. The constructor of the enum takes in a function in this case `DoubleBinaryOperator` (provided in Java 8) which represents an operation on two double-valued operands that produces a double-valued result.

Operation.java

```
package com.javacodegeeks.example;

import java.util.function.DoubleBinaryOperator;

public enum Operation {
    ADDITION((x, y) -> x+y),
    SUBTRACTION((x, y) -> x-y),
    DIVISION((x, y) -> x/y),
    MULTIPLICATION((x, y) -> x*y),
    PERCENTAGE((x, y) -> x%y);
```

```
    private DoubleBinaryOperator operator;

    Operation(DoubleBinaryOperator operator) {
       this.operator = operator;
    }

    public DoubleBinaryOperator getOperator() {
        return operator;
    }
}
```

4.5 Putting everything together

Now we add action listeners that will be triggered when the buttons of the calculator get clicked. We then bind the buttons to those action listeners.

Calculator.java

```
package com.javacodegeeks.example;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Calculator {
    private JTextField resultsTxt;
    private JButton clearBtn;
    private JButton signBtn;
    private JButton percentBtn;
    private JButton divideBtn;
    private JButton sevenBtn;
    private JButton eightBtn;
    private JButton nineBtn;
    private JButton multiplyBtn;
    private JButton fourBtn;
    private JButton fiveBtn;
    private JButton sixBtn;
    private JButton minusBtn;
    private JButton oneBtn;
    private JButton twoBtn;
    private JButton threeBtn;
    private JButton addBtn;
    private JButton zeroBtn;
    private JButton equalBtn;
    private JButton digitBtn;
    private JPanel calculatorView;
    private Double leftOperand;
    private Double rightOperand;
    private Operation calcOperation;

    public Calculator() {

        sevenBtn.addActionListener(new NumberBtnClicked(sevenBtn.getText()));
        eightBtn.addActionListener(new NumberBtnClicked(eightBtn.getText()));
        nineBtn.addActionListener(new NumberBtnClicked(nineBtn.getText()));
        fourBtn.addActionListener(new NumberBtnClicked(fourBtn.getText()));
        fiveBtn.addActionListener(new NumberBtnClicked(fiveBtn.getText()));
        sixBtn.addActionListener(new NumberBtnClicked(sixBtn.getText()));
        oneBtn.addActionListener(new NumberBtnClicked(oneBtn.getText()));
```

```
        twoBtn.addActionListener(new NumberBtnClicked(twoBtn.getText()));
        threeBtn.addActionListener(new NumberBtnClicked(threeBtn.getText()));
        zeroBtn.addActionListener(new NumberBtnClicked(zeroBtn.getText()));

        percentBtn.addActionListener(new OperationBtnClicked(Operation.PERCENTAGE));
        multiplyBtn.addActionListener(new OperationBtnClicked(Operation.MULTIPLICATION));
        divideBtn.addActionListener(new OperationBtnClicked(Operation.DIVISION));
        minusBtn.addActionListener(new OperationBtnClicked(Operation.SUBTRACTION));
        addBtn.addActionListener(new OperationBtnClicked(Operation.ADDITION));
        equalBtn.addActionListener(new EqualBtnClicked());
        clearBtn.addActionListener(new ClearBtnClicked());
        signBtn.addActionListener(new SignBtnClicked());
        digitBtn.addActionListener(new DigitBtnClicked());
    }

    private class NumberBtnClicked implements ActionListener {

        private String value;

        public NumberBtnClicked(String value) {
            this.value = value;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            if(leftOperand == null || leftOperand == 0.0) {
                value = resultsTxt.getText() + value;
            }else{
                rightOperand = Double.valueOf(value);
            }
            resultsTxt.setText(value);

        }
    }

    private class OperationBtnClicked implements ActionListener {

        private Operation operation;

        public OperationBtnClicked(Operation operation) {
            this.operation = operation;
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            calcOperation = operation;
            leftOperand = Double.valueOf(resultsTxt.getText());
        }
    }

    private class ClearBtnClicked implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            resultsTxt.setText("");
            leftOperand = 0.0;
            rightOperand = 0.0;
        }
    }

    private class DigitBtnClicked implements ActionListener {
```

```
        @Override
        public void actionPerformed(ActionEvent e) {
            resultsTxt.setText(resultsTxt.getText() + ".");

        }
    }

    private class EqualBtnClicked implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            Double output = calcOperation.getOperator().applyAsDouble(leftOperand,  ↩
                rightOperand);
            resultsTxt.setText(output%1==0?String.valueOf(output.intValue()):String.valueOf ↩
                (output));
            leftOperand = 0.0;
            rightOperand = 0.0;
        }
    }

    private class SignBtnClicked implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            resultsTxt.setText("-"+ resultsTxt.getText());
        }
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("Calculator");
        frame.setContentPane(new Calculator().calculatorView);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

4.6 Running your application

Right-click `Calculator.java` and select Run `Calculator.main()`

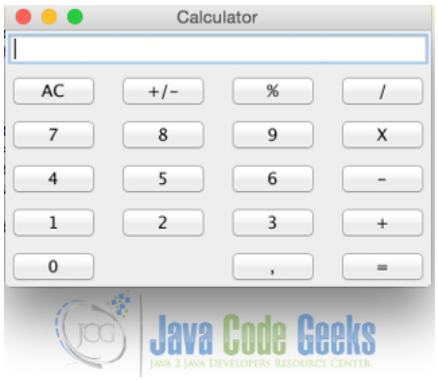


Figure 4.7: Calculator

4.7 Download the IntelliJ Project

This was an example of using IntelliJ GUI Designer to create a Swing Calculator.

Download You can download the full source code of this example here: **CalculatorGUI**

Chapter 5

IntelliJ Idea Color Schemes / Themes Configuration

In this article we are going to present how to customize the IntelliJ IDEA built-in color schemes or provide your own color theme for use in the IDE. Using a color scheme may prove to be quite useful for a developer especially in terms of code readability. For this purpose we will use the IntelliJ IDEA 14.1.2 community edition which is available for free for all end users.

Color scheme management in IntelliJ IDEA was modified to simplify the work of scheme designers and make schemes look equally well for different programming languages even if not designed specifically for these languages. Previously language plug-ins were using fixed default colors incompatible, for example, with dark schemes. The new implementation from version 12.1 and above allows to specify a dependency on a set of standard text attributes which are linked to a scheme but not to any specific language. Language-specific attributes still can be set by a scheme designer if needed but it's optional. New color schemes have got a new .icls (Idea CoLor Scheme) extension to avoid confusion about compatibility problems with older platform versions: if only standard attributes are set, they will not be used by the version prior to 12.1 and this will result in different highlighting colors.

5.1 Color Schemes

With IntelliJ IDEA, you can maintain your preferable colors and fonts layout for syntax and error highlighting in the editor, search results, Debugger and consoles via font and color schemes. IntelliJ IDEA comes with a number of pre-defined color schemes. You can select one of them, or create your own one, and configure its settings to your taste. Note that pre-defined schemes are not editable. You have to create a copy of a scheme, and then change it as required. The 14.1.2 version which we are using comes with 2 predefined color themes / schemes:

- Darcula: dark color scheme which seems more natural for many developers. Preferable to software developers who are used to text editors and nix-based OS console look.
- Default: light color theme with white console background

5.1.1 Configuring general color scheme

In order to configure color and font scheme, open File→Settings, and under the Editor node, click Colors & Fonts. Select the desired scheme from the Scheme name drop-down list. If you need to change certain settings of the selected scheme, create its copy. To do that, click Save as button, and type the new scheme name in the dialog box:

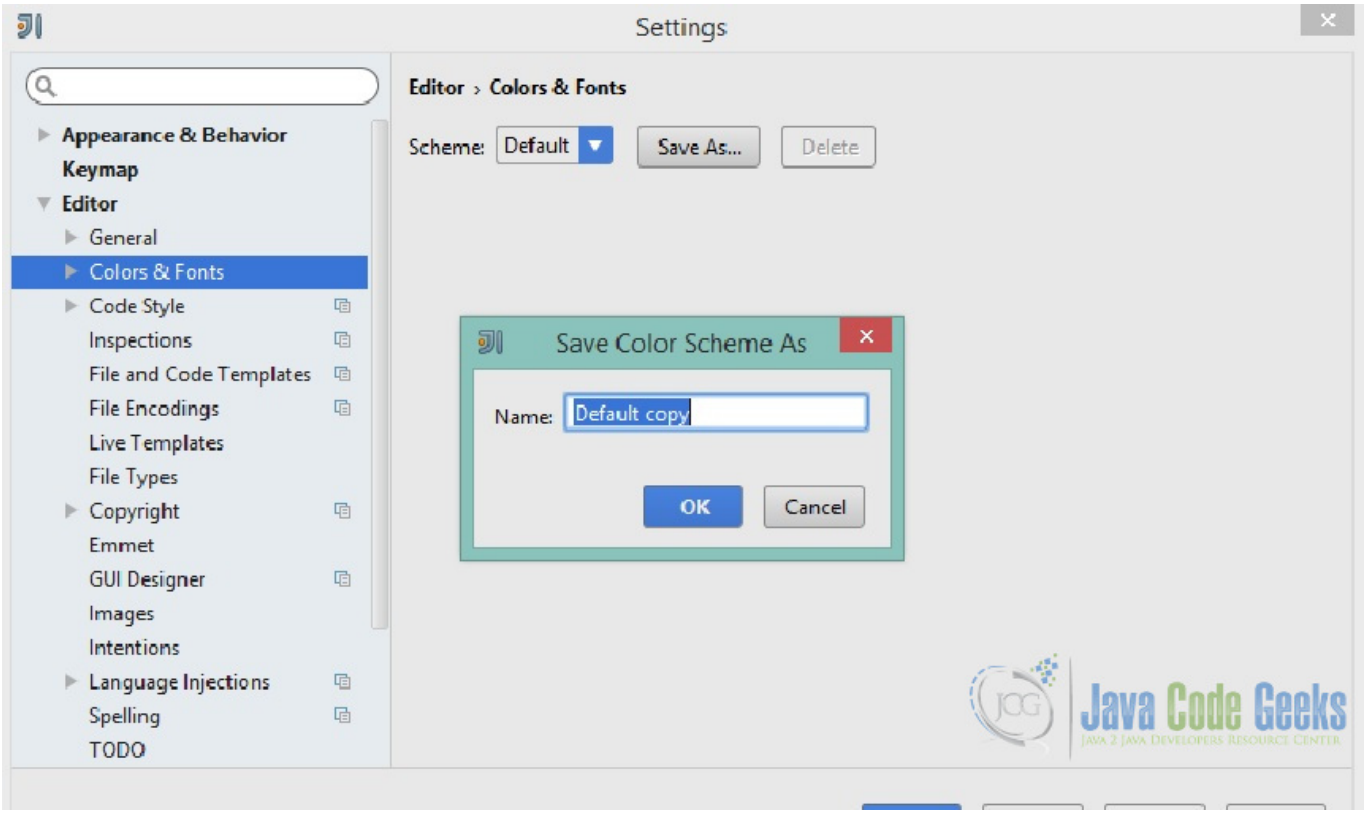


Figure 5.1: IntelliJ IDEA Colors and Fonts Settings

You can quickly switch between various color schemes, keyboard layouts, and look-and-feels without actually invoking the corresponding page of the Settings dialog box. To switch between schemes choose View → Quick Switch Scheme on the main menu. In the pop-up window that opens select the desired scheme (Colors and Fonts, Code Style, etc.). In the suggestion list, click the desired option.

5.1.2 Color Preferences for specific IntelliJ IDEA components

Under the Colors and Fonts node, open pages to configure specific color preferences and font types for the different supported languages and IntelliJ IDEA components. The user may set his own color and font schemes for the Console, for the Debugger, for Android LogCat and for a numerous of other components used as part of the code development (e.g. Groovy, HTML, JSON, RegExp, XML etc.). As an example we are going to change the colors and fonts for the IntelliJ IDEA Java component.

Click Java node as shown in the image below:

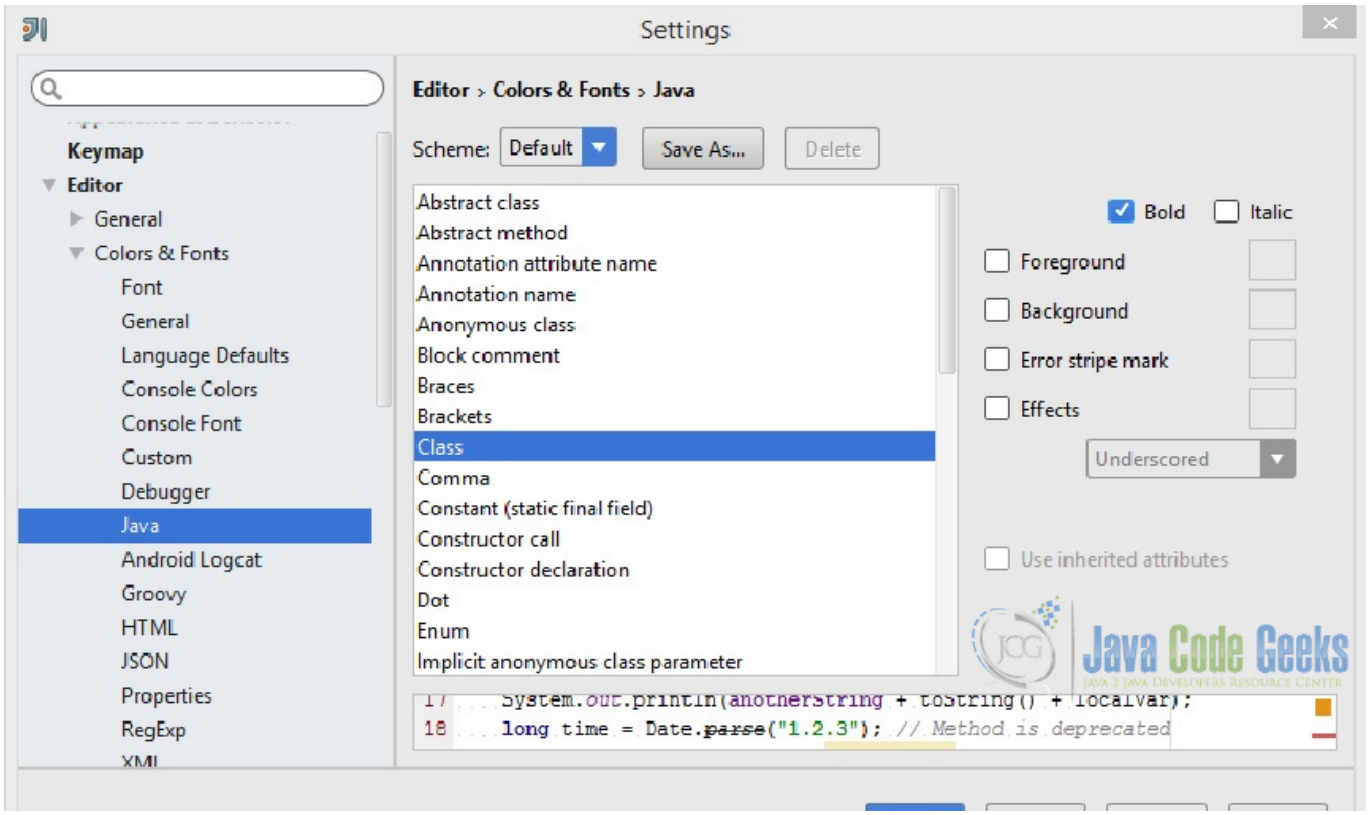


Figure 5.2: IntelliJ IDEA Java Node Colors and Fonts

On the above window, you can select from the dropdown menu the specific java component which you need to customize regarding the colors and fonts. Keep in mind that in order to be able to proceed with the customization, a custom scheme must be defined in the Scheme drop down menu (meaning a scheme other than the predefined ones).

After you have saved your custom Scheme you can select a Java component, e.g Class and then on the right of the window come the customized selections for the component. You have a variety of selections like foreground / background color, error stripe mark color, specific effects color (drop down menu) and bold or italics checkboxes for the component. The below image illustrates the window opened after pressing the corresponding color picker button:

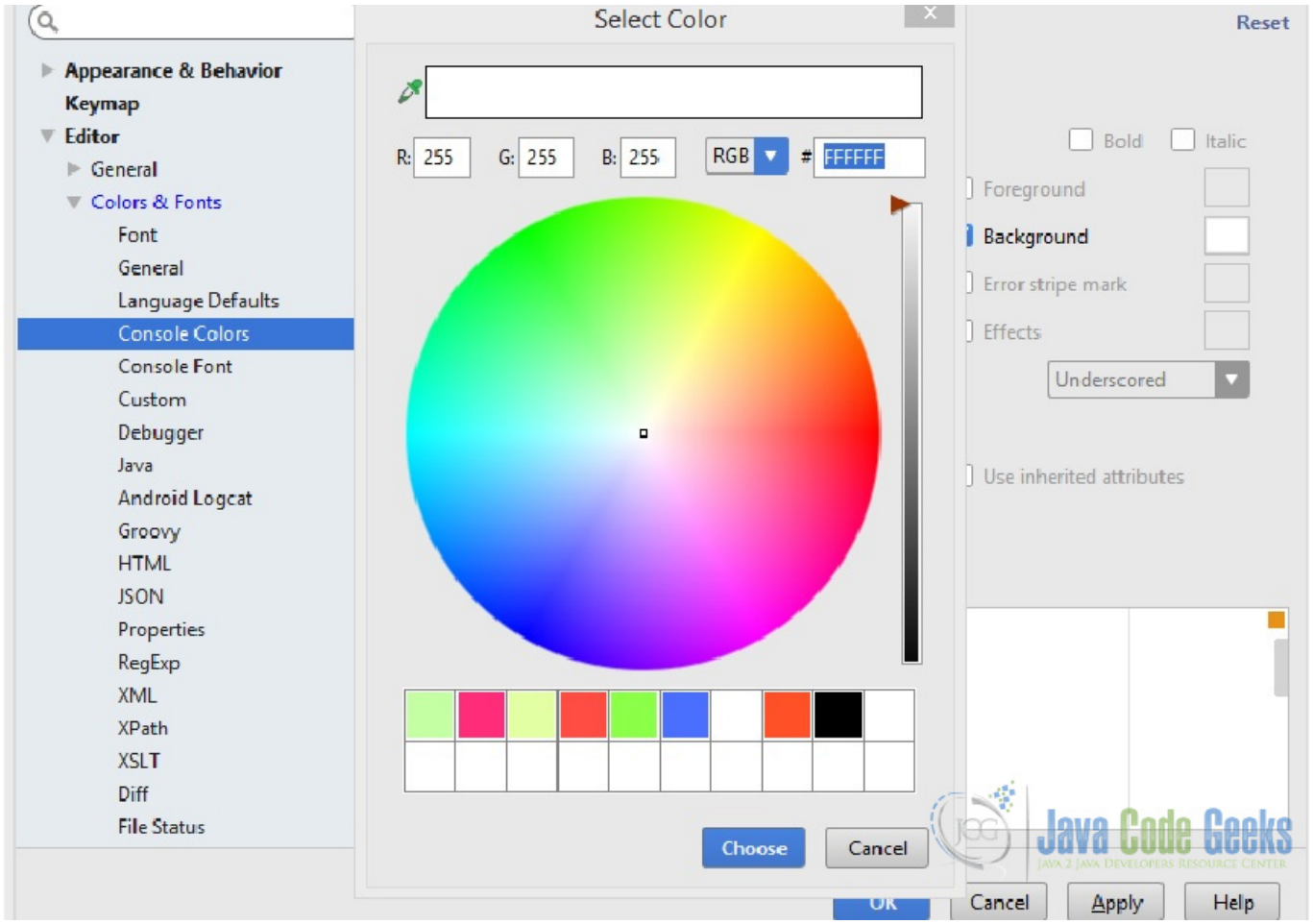


Figure 5.3: IntelliJ Color Picker Window

A preview of your selections is available before applying the settings. The below image illustrates a custom color scheme for the Java Class component:

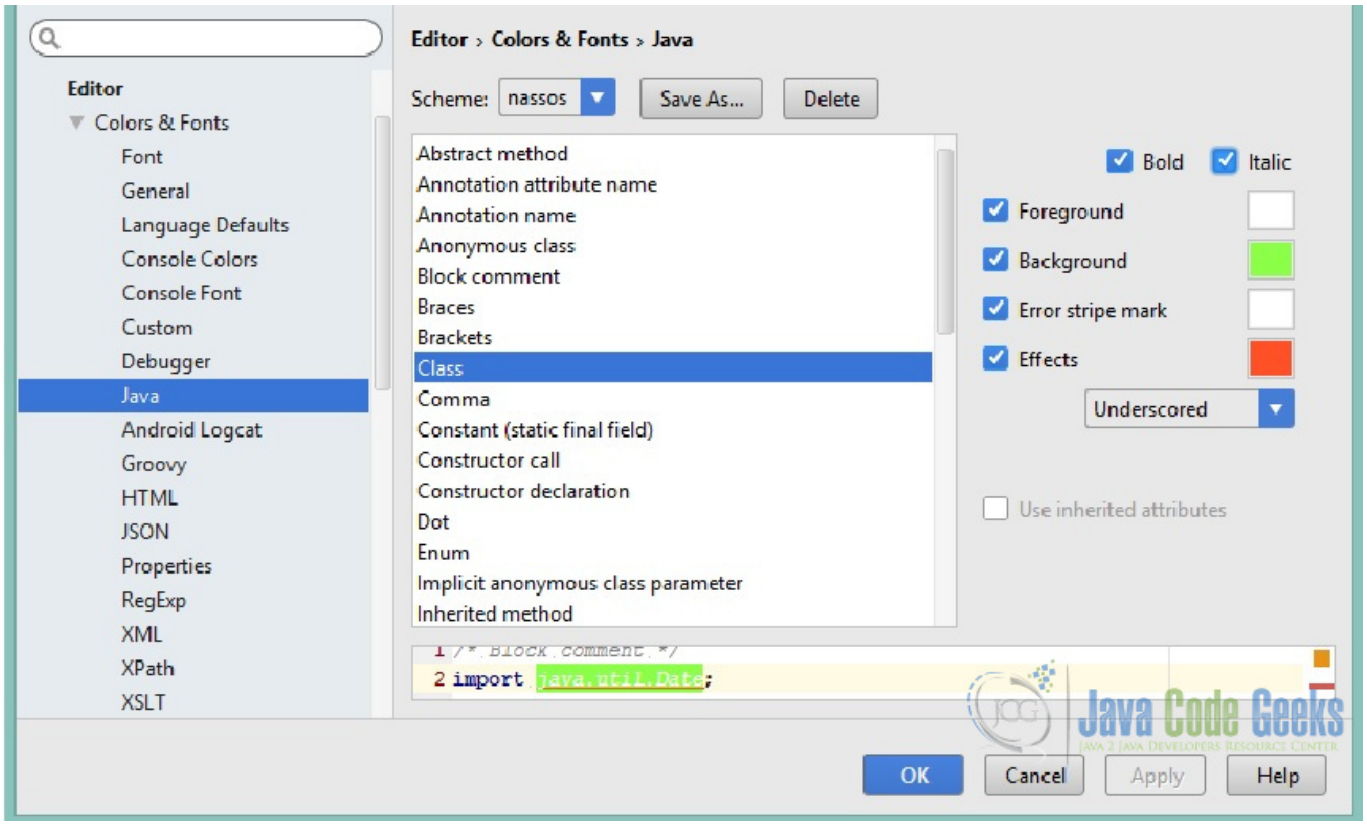


Figure 5.4: Java Class Color Scheme

5.2 Custom color themes

Apart from the predefined color themes provided by IntelliJ IDEA, the user may create his own color theme or import one of the publicly available color themes created by other users.

5.2.1 Install using "Import Settings..."

- Go to File → Import Settings... and specify the jar file with the settings of the custom theme. Click OK in the dialog that appears.

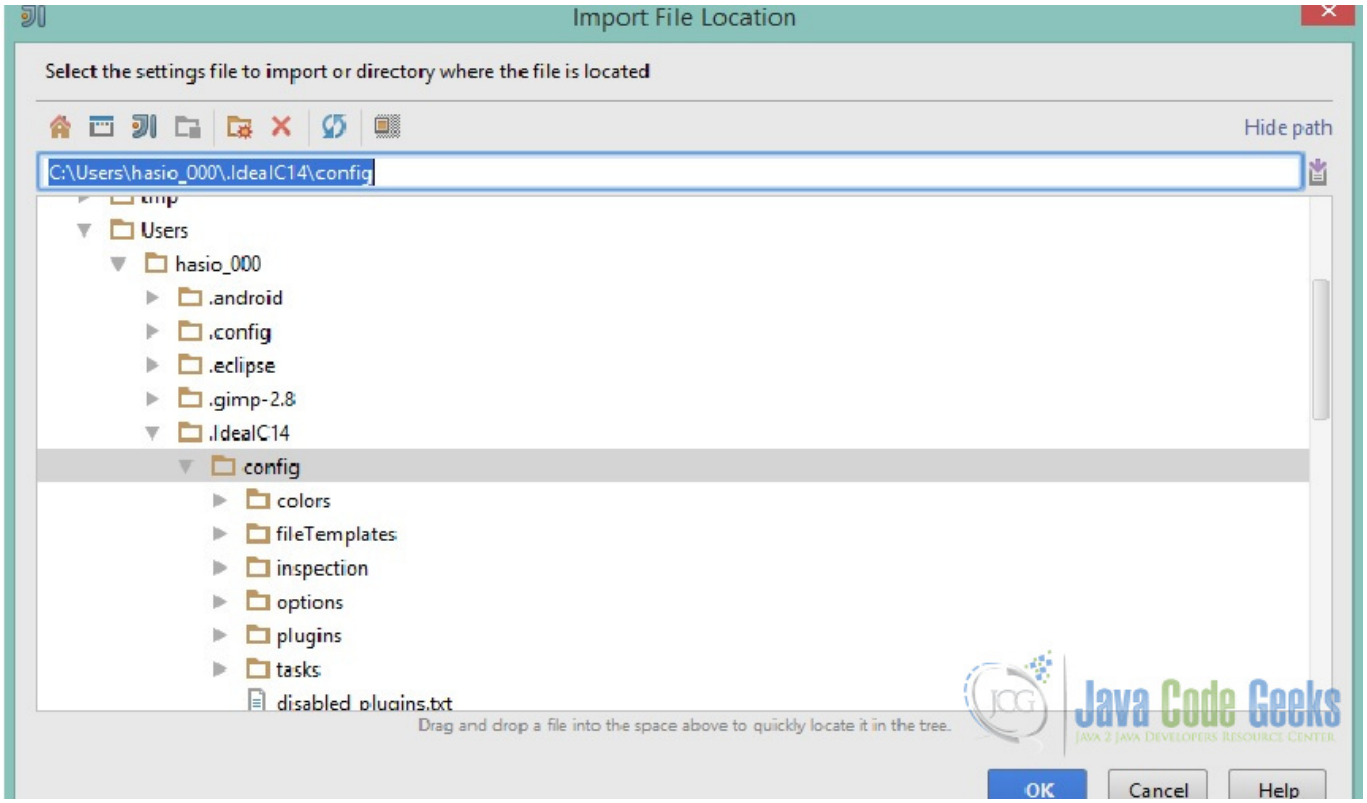


Figure 5.5: IntelliJ Import Settings

- Restart IntelliJ IDEA
- Go to Setting → Editor > Colors & Fonts and select one of the new color themes.

5.2.2 Manual installation

- Copy the .icls file of the custom theme to your IntelliJ IDEA preferences color directory.
- The directory varies, depending on which JetBrains IDE you are using. For the 14.1.2 version which we are using the directory is the following: %USERPROFILE%.IdeaIC14configcolors
- In case you need to reset to the default color schemes, you can simply delete the whole configuration folder while IntelliJ IDEA is not running. Next time it restarts, everything is restored from the default settings.

5.2.3 Conclusion

As a developer, you are not obliged to stick to the default fonts and colors of the IDE. There is a significant amount of themes out there one of which may be the proper one for your taste or your color deficiency. IntelliJ also gives you the possibility to easily create a color scheme of your own. Dark or bright theme, just a matter of taste and configuration.

Chapter 6

IntelliJ Increase Memory Settings

In this article we are going to provide some guidelines on how to increase memory heap on IntelliJ IDEA and on how to configure the VM options. For our example we are using IntelliJ IDEA Community Edition version 14.1.2.

6.1 Configuring IntelliJ IDEA VM options

The default VM options for IntelliJ IDEA may be not optimal when your project contains a quite large number of classes (e.g more than 15000) and developers often try to change the default options to minimize IntelliJ IDEA hangtime. But sometimes the changes make things even worse. So, how to configure IntelliJ IDEA VM options optimally? That's not so easy question to answer, since the configuration strongly depends on the project being developed. Therefore, we can recommend some settings that developers use and explain the general memory policy.

6.1.1 Optimal VM options

```
-Xms = 128m
```

```
-Xmx = 256m
```

Please note that very big Xmx and Xms values are not optimal. In this case, the garbage collector has to deal with a large part of memory at a time and causes considerable hang-ups. On the other hand, too small values can lead to the OutOfMemory exception. The specified values provide enough memory and at the same time the garbage collector works often but rather fast.

```
-XX:MaxPermSize=92m
```

This is a default value, and in most cases you don't need to change it. You may increase it only if you get "OutOfMemoryError" in "PermGen space".

```
-server
```

Some people find IntelliJ IDEA more responsive with this option. But it is not guaranteed. We don't recommend to use the following options at all, since they are not very stable:

```
-XX:+UseParallelGC
```

If you are on a multi-core machine, enabling parallel garbage collection can help (reduces gc pauses), although there are stability issues with this option.

```
-XX:+UseAdaptiveSizePolicy
```

```
-XX:-UseConcMarkSweepGC
```

6.1.2 Changing IntelliJ IDEA properties

IntelliJ IDEA makes it possible to change *.vmoptions and idea.properties files without editing them in the IntelliJ IDEA installation folder. In IntelliJ IDEA 14.1.2 these files are located under `C:/Program Files (x86)/JetBrains/IntelliJ IDEA Community Edition 14.1.2/bin` installation directory. You should edit a copy of these files. The reason is that app bundle is signed and you should not modify any files inside the bundle; therefore, you need to make a copy in IDE preferences folder, which you will modify.

6.1.2.1 Managing *.vmoptions file

For Mac OS X systems

Since version 12.0.0: The file `/Applications/IntelliJ Idea XX.app/bin/idea.vmoptions` or `/Applications/IntelliJ Idea CE XX.app/bin/idea.vmoptions` should be copied to `~/Library/Preferences/IntelliJIdeaXX/idea.vmoptions` or `~/Library/Preferences/IdeaICXX/idea.vmoptions`.

Since version 14.0.0: The file `/Applications/IntelliJ Idea XX.app/Contents/bin/idea.vmoptions` or `/Applications/IntelliJ Idea CE XX.app/Contents/bin/idea.vmoptions` should be copied to `~/Library/Preferences/IntelliJIdeaXX/idea.vmoptions` or `~/Library/Preferences/IdeaICXX/idea.vmoptions`.

For the older versions, the settings are stored in `/Applications/IntelliJ IDEA .app/Contents/Info.plist`.

For *NIX and Windows systems

To avoid editing files in the IntelliJ IDEA installation folder, one should:

Do one of the following:

- Go to Run→Edit Configurations… and set the VM options field with the desired values:

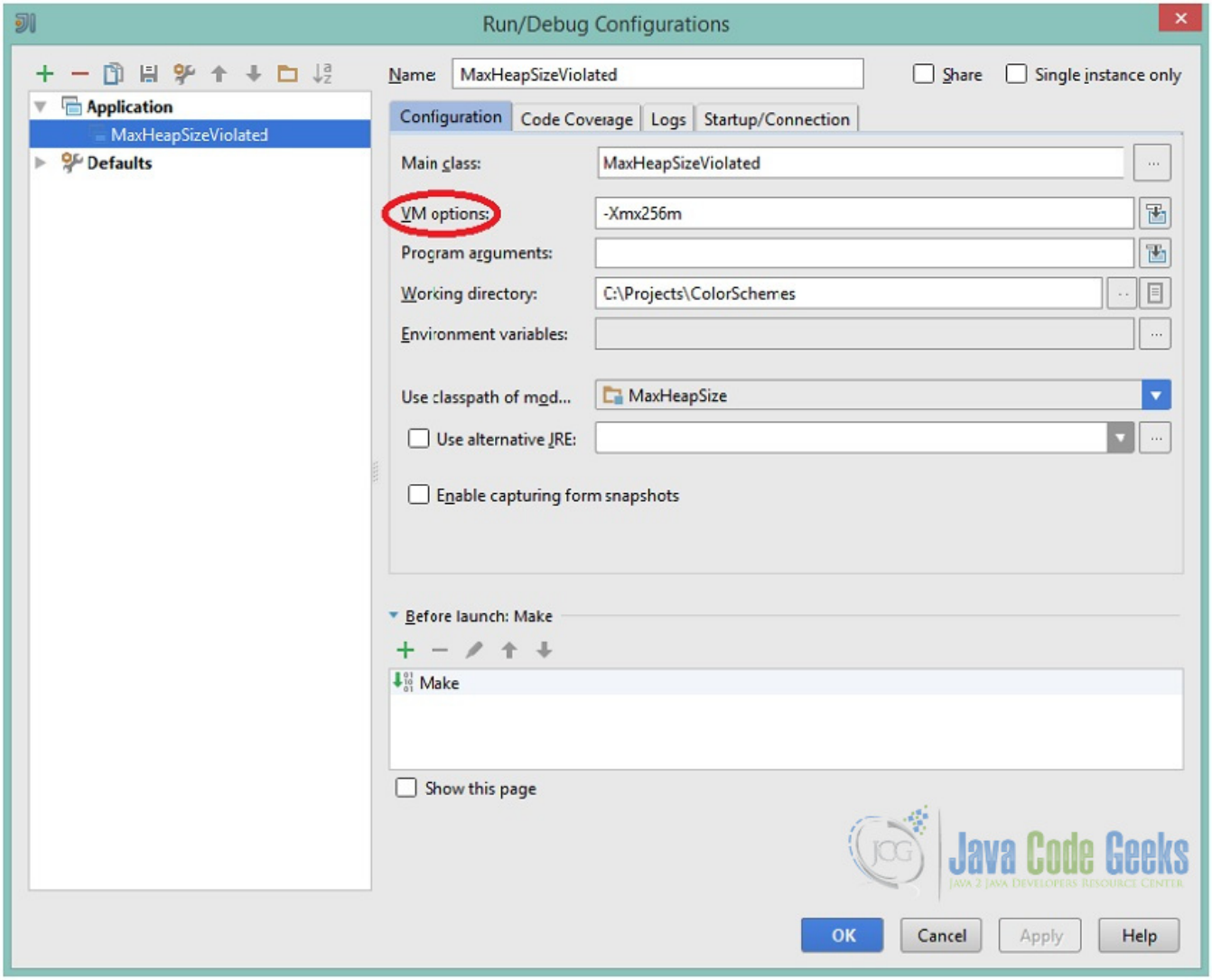


Figure 6.1: VM options configuration

- Copy the existing file from the IntelliJ IDEA installation folder somewhere and save the path to this location in the environment variable IDEA_VM_OPTIONS. If IDEA_VM_OPTIONS environment variable is defined, or the *.vmoptions file exists, then this file is used instead of the file located in the IntelliJ IDEA installation folder.
- Copy the existing file /bin/idea$BITS.vmoptions from the IntelliJ IDEA installation folder into the location under user home:
  For Windows: %USERPROFILE%.IntelliJIdeaXXidea%BITS%.exe.vmoptions or %USERPROFILE%.IdeaICXXidea%BITS%.exe.
  The value of the variable BITS depends in the JVM used to run IntelliJ IDEA:
  - For 32-bit JVM it is empty.
  - For 64-bit JVM it is 64.
- Edit this file in the new location.

6.2 Increasing memory heap of the build process

To increase a memory heap:

Open the Build→Compiler Settings dialog box:

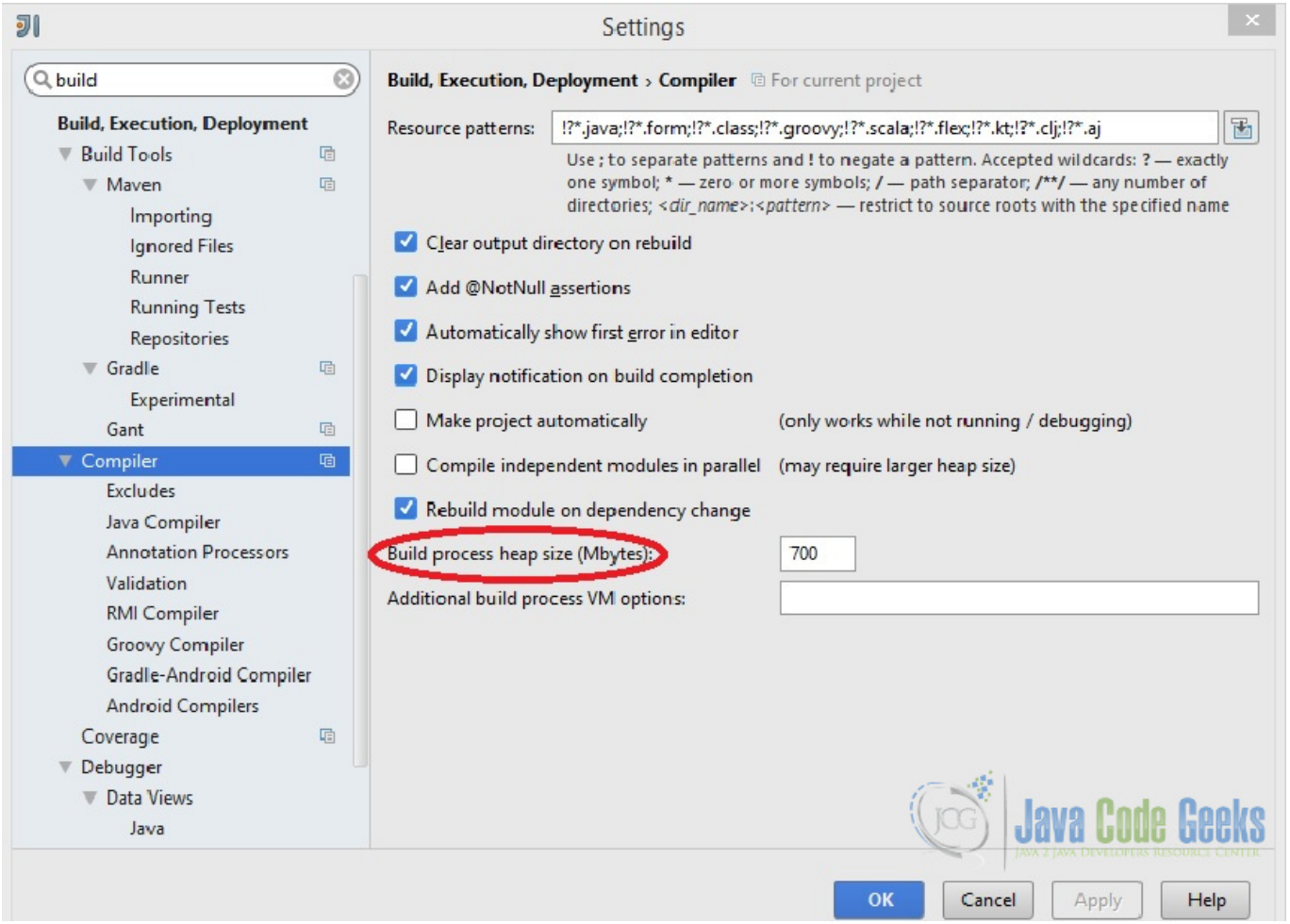


Figure 6.2: Build process heap size

In the Build process heap size field, type the required amount of memory in MB.

Please note the following:

The memory heap of the build process is independent of IntelliJ IDEA memory heap, and is released after the build process is complete. The IntelliJ IDEA memory heap is indicated on the bottom right of the IDE and specifically on the status bar which shows the current IDE state and lets you carry out certain environment maintenance tasks. Visibility of this section in the Status bar is defined by the Show memory indicator check box in the Appearance page of the Settings dialog as depicted in the image below. It is not shown by default.

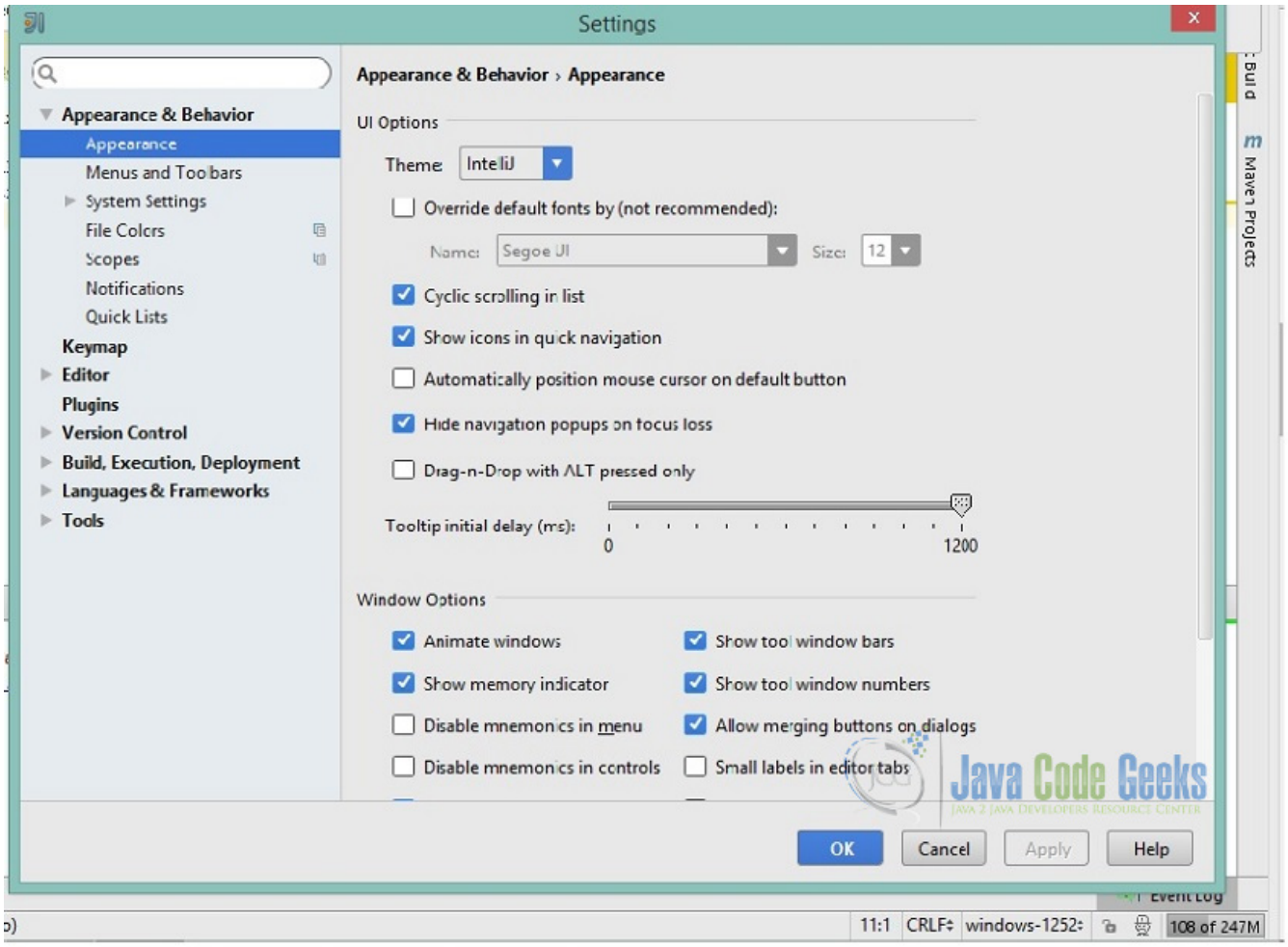


Figure 6.3: Memory Indicator

Upon clicking the indicator, the garbage collector runs.

6.3 Code example

In this section we are going to present a code example in which we use the default settings (-Xms64m, -Xmx128m) and leads to an OutOfMemoryException and the same example running successfully after setting VM options in Run Configurations with -Xmx256m.

```
public class MaxHeapSizeViolated {

    static final int SIZE=30*1024*1024;
    public static void main(String[] a) {
        int[] i = new int[SIZE];
    }
}
```

The array of integers allocated with the above code example leads to an OutOfMemoryError when using the default -Xmx128m vm option:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at MaxHeapSizeViolated.main(MaxHeapSizeViolated.java:8)
```

After setting the -Xmx256m vm option in Run configurations the above code does not lead to an OutOfMemory exception.

6.4 Conclusion

In this example we tried to demonstrate the various ways of increasing / modifying memory settings inside IntelliJ IDEA and it looks like that even some simple changes in the VM options can significantly improve performance of our work. The purpose is to use the IDE in order to run different applications with regard to memory requirements and for this reason the developer should be able to find a balance between performance gain and memory consumption.

Chapter 7

IntelliJ IDEA Keyboard Shortcuts Tutorial

The developers, systems administrators, etc., when dealing with files, have the need of being able to perform actions without depending on the mouse. Not only for speed and comfort, but also to avoid injuries related to the mouse usage, like RSI ones (Repetitive Strain Injury).

This tutorial will show the most useful shortcuts for IntelliJ IDEA, the JetBrains IDE for Java, Groovy, and other languages.

For this tutorial, we will use:

- IntelliJ IDEA 2016.1.2 Community Edition (free and open source).
- Linux Mint 17.03.

You can download the IDE from the official JetBrains website, available for Linux, Windows and MacOS.

7.1 Editing

Basic code completion: Ctrl + Space

For completing keywords, variables, methods, etc.

Smart code completion: Ctrl + Shift + Enter

This is how IntelliJ IDEA defines its smart code completion:

Smart Type code completion filters the suggestion list and includes only those types that are applicable to the current context.

That is the reason of considering it "smart".

Let’s see how it works with an example:

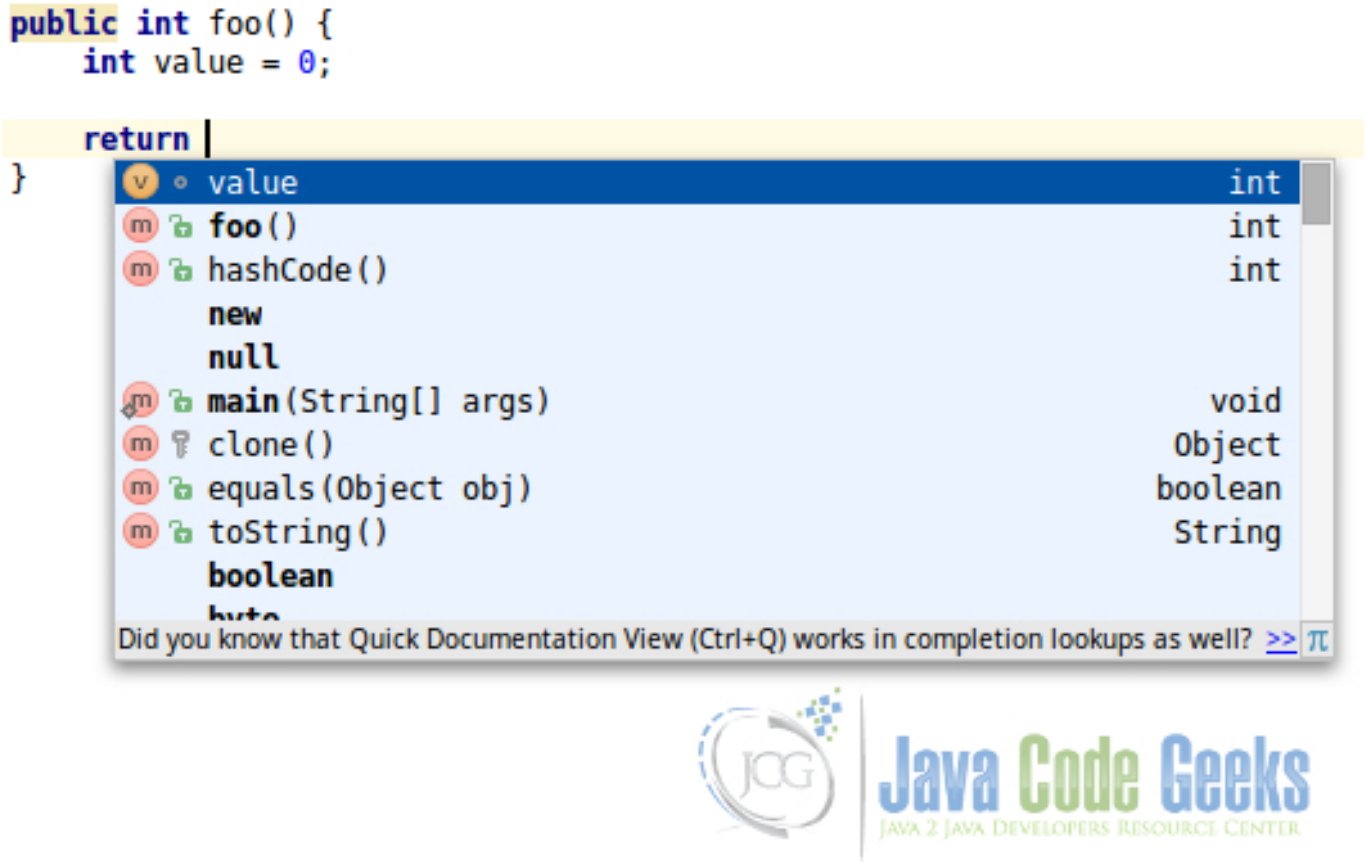


Figure 7.1: Using the smart code completion

That is, using this shortcut, in a return statement in this case, the IDE knows that value will probably be the `value` to complete the statement with, even before starting typing the word.

Comment/uncomment line/selection with line comment: Ctrl + /

For example, if we would select the following lines:

```
public static void main(String[] args) {
    System.out.println("Hello world!");
}
```

This shortcut would comment each line with `//`:

```
//public static void main(String[] args) {
//    System.out.println("Hello world!");
//}
```

Comment/uncomment line/selection with block comment: Ctrl + Shift + /

Exactly the same as the previous one, but this for block comments surrounded with `/* */`.

Delete line at cursor: Ctrl + Y

Also works for the selection, but this is usually done with backspace key. Pressing these keys will delete the line where the caret is placed.

Duplicate current line/selection: Ctrl + D

Just for generating duplicates below the line or selection.

Generate methods (constructor, getters & setters, etc.): Alt + Insert

With this combination, IntelliJ IDEA will prompt the following window:

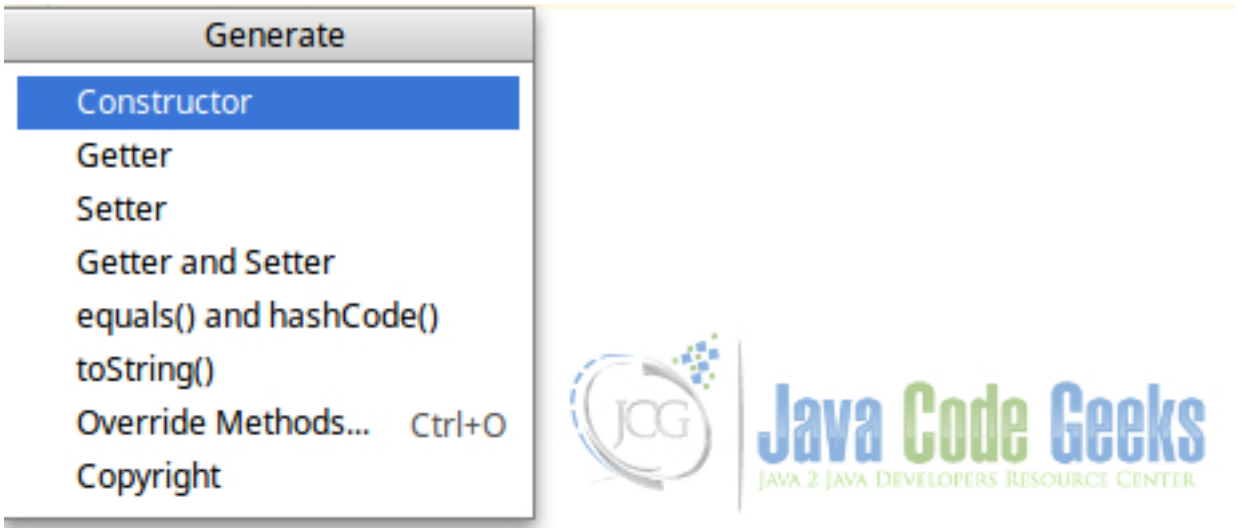


Figure 7.2: Methods available to be generated automatically

The getters and setters, and also the constructor, are suggested based on the class attributes.

Override parent methods: Ctrl + O

Similar to the previous one since the `Object` methods also appear (`hashCode`, `equals`, etc.), but is also applicable for methods of extended classes.

Surround line/selection: Ctrl + Alt + T

This allows to surround the code automatically with control structures such as `if`, `for`, `try/catch`, etc. and also with `synchronized` or `Runnable` blocks.

Indent line/selection: Ctrl + Alt + I

The selection will be indented with the specified configuration in Settings/Editor/Code Style, for the language you are working with, in the "Indent" setting.

Reformat code: Alt + Ctrl + L

If the previous one was for indenting the code, this one is for reformatting the code in all of its aspects, for the whole file, again based on the settings defined in Settings/Editor/Code Style.

For example, with this shortcut, we could convert the following disordered code:

```
public static
void main( String[] args ){System.out.println(  "Hello world!"  )  ; }
```

To:

```
public static void main(String[] args) {
    System.out.println("Hello world!");
}
```

Depending, of course, on the indentation level, spaces before and after braces, etc.

Rename: Shift + F6

This is applicable to any identifier (class names, methods, attributes, variables, etc.). This is done "safely", i.e., IntelliJ IDEA won't break anything in the code.

Find & replace: Ctrl + R

This combination shows the typical tool window with two search boxes, one to find the term to replace, and another for the replacement. IntelliJ IDEA also allows to use regular expressions for the finding.

7.2 Searching/navigating

In this section we will see how to make searches, and also how to navigate through the files.

Find text in current file: Ctrl + F

The classical shortcut for many programs.

Find next coincide: F3 or Enter

Set the selection of the entered text to find in the following coincidence.

Find previous: Shift + F3

The same as the previous one, but for the previous coincidence.

Go to class: Ctrl + N

With this shortcut you can find a go to any class, receiving suggestions, like in the following image:

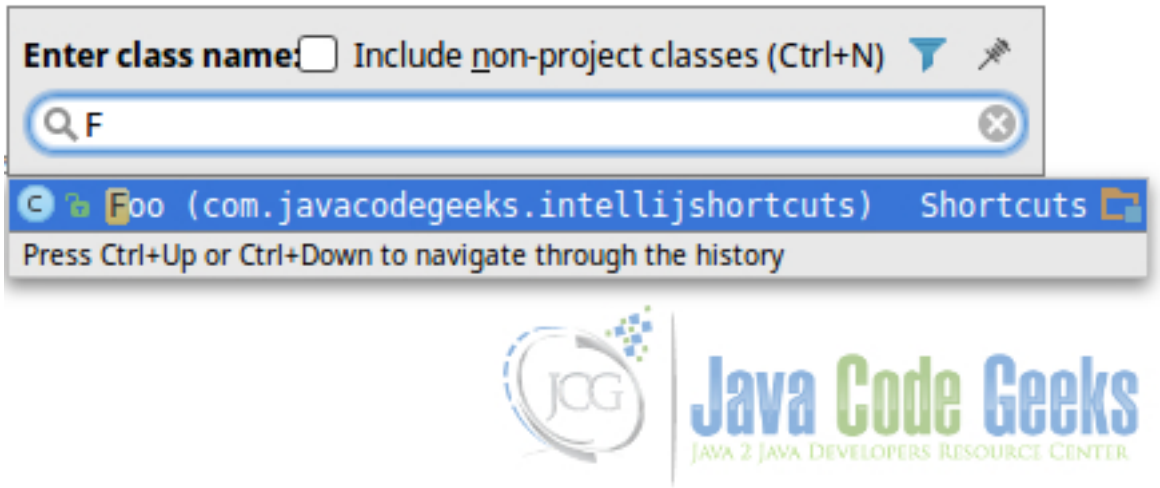


Figure 7.3: Navigating between classes

Go to any file: Ctrl + Shift + N

The same as the previous one, but applicable to any file, not only classes.

Go to declaration: Ctrl + B

In this case, we have to place the cursor in element (object, method, attribute, parameter, etc.), and the IDE will navigate to the place of its declaration, even if it is in another file.

Go to implementations: Ctrl + Alt + B

If we place the cursor in a declaration (interfaces, abstract classes and methods), and we press this combination, IntelliJ IDEA will show a list of the places where it is being implemented to navigate to it, like shown in the following example:

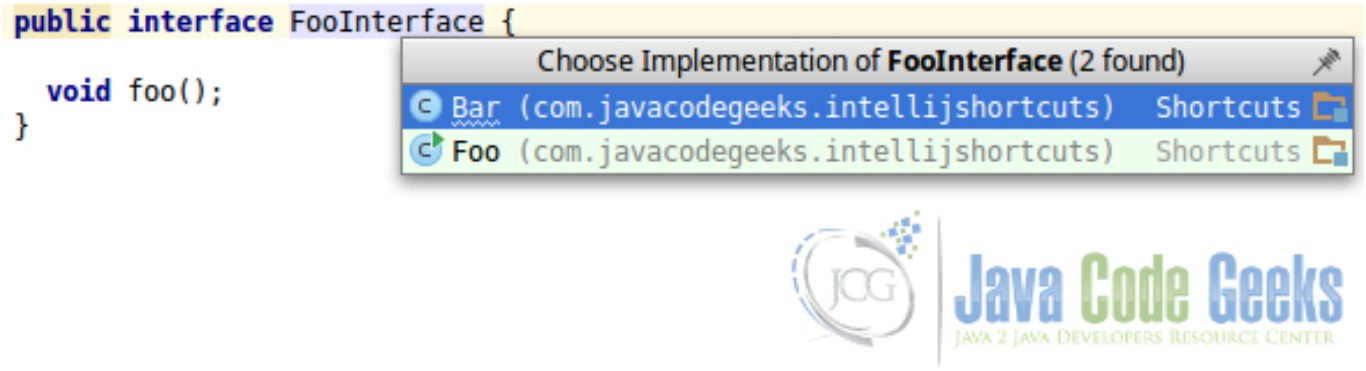


Figure 7.4: Navigating to implementations

If there's only one implementation, the IDE will navigate directly to it.

Go to super-method/class: Ctrl + U

This is the same as the previous one, but reverse: navigating to the parent definition of a method or class.

Go to line & column: Ctrl + G

This shortcut will show a window where we can specify the line and column of the current file, to navigate to it. The format is `line:column`. But if we can also specify only the line, if the column is not necessary.

Switch between tabs: Ctrl + Tab

Just like, e.g., in web browsers.

7.3 Compiling and running

Compile project: Ctrl + F9

This just makes the project, without running it.

Select configuration and run: Alt + Shift + F10

This allows to configure the project before running it, selecting the main class, setting parameters for the JVM and the program, etc.

Run project: Shift + F10

Runs the project with the configuration set.

Select configuration and debug: Alt + Shift + F9

As same as selecting the configuration and running, but for debugging.

Debug project: Shift + F9

Equal to running the project, but debugging it.

7.4 Other shortcuts

Find action by name: Ctrl + Shift + A

A very useful shortcut, specially when you are not familiarized with other shortcuts yet. This allows to find any available action or option in IntelliJ IDEA, by the name, like shown in the following image:

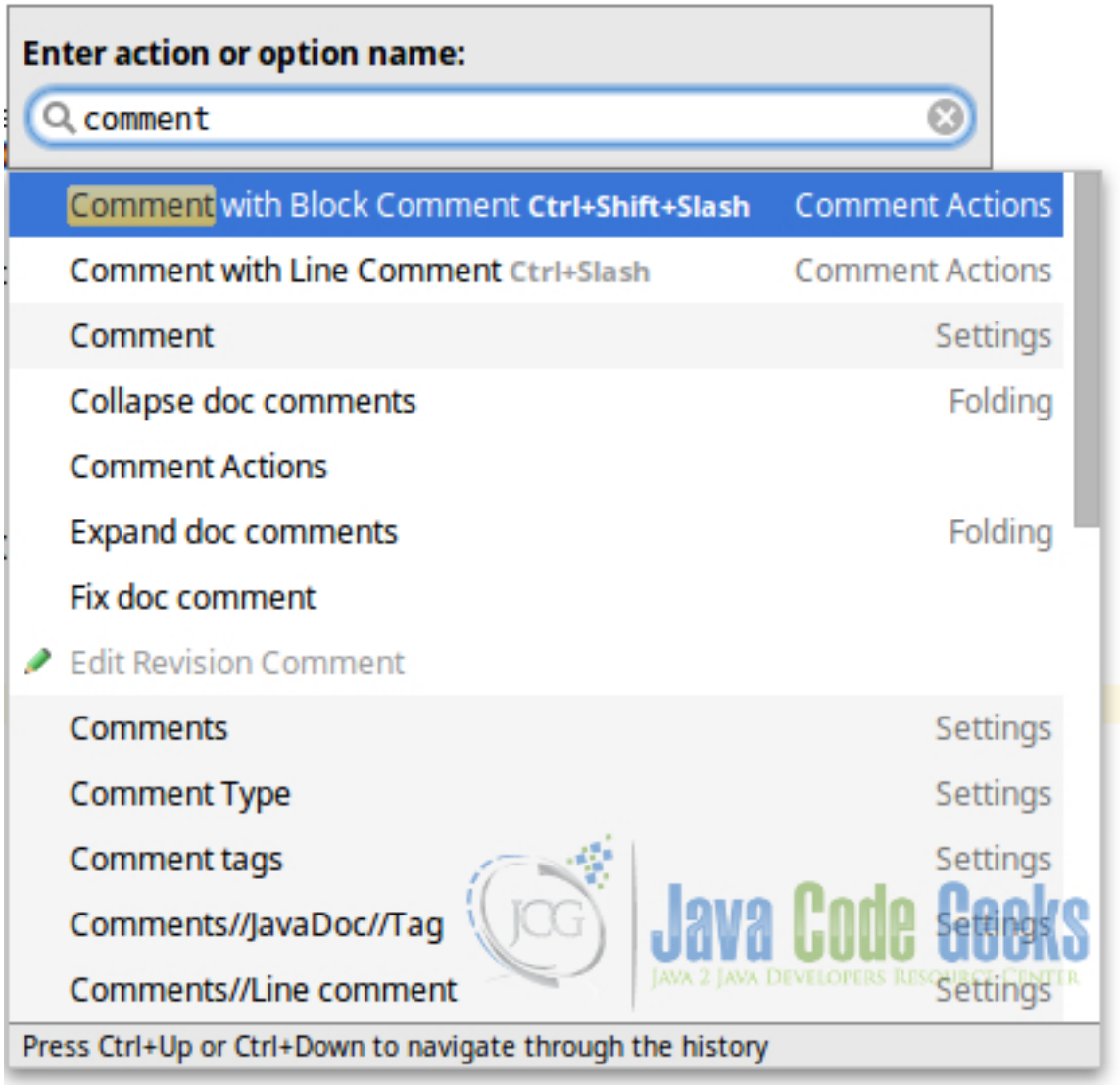


Figure 7.5: Finding available actions by name

Open settings: Ctrl + Alt + S

This will open the settings menu (File/Settings).

Toggle full screen editor: Ctrl + Shift + F12

For entering/exiting the full screen in the editor.

7.5 Vi/Vim fan? This is for you!

This is not exactly related to the IntelliJ IDEA shortcuts, but it's worth mentioning.

If you are a former Vim user and you miss its commands, or you still use it and you don't want to learn other shortcuts, or even if you want to learn to use it but using IntelliJ IDEA, there is a plugin for Vim emulation.

7.5.1 Installation

- Open settings menu (Ctrl + Alt + S).

- Go to plugins menu.
- Click the "Install JetBrains plugin. . . " button.
- Type "IdeaVim" in the search box.
- Click the "Install" button, and restart the IDE.

7.5.2 Usage

To use the Vim emulator, the only thing we have to do is to ensure that Tools/Vim Emulator has a tick. If checked, the editor will be in Vim mode. We can activate/deactivate whenever we want.

7.6 Summary

In this tutorial we have seen the IntelliJ IDEA shortcuts for code edition, navigation between files and compilation and execution of the project. Apart from that, we have also seen the Vim plugin for IntelliJ IDEA, that allows to have a better experience for Vim users.

Chapter 8

IntelliJ IDEA Create Test Tutorial

IntelliJ IDEA is one of the most used Java IDE. If we want to create tests for our IntelliJ project, it may not be very intuitive, so, in this tutorial we will see how to create them.

For this tutorial, we will use:

- IntelliJ IDEA 2016.2.2 Community Edition (free and open source).
- Linux Mint 18.

You can download the IDE from the official JetBrains website, available for Linux, Windows and MacOS.

8.1 Project creation

If it is the first time you open IntelliJ IDEA, you will see that it suggests you to create a new project. Otherwise, create a new file selecting "File/New/Project".

In the new window, you have to select "Java" as the project type, and then select the SDK for the project, as shown in the image below.

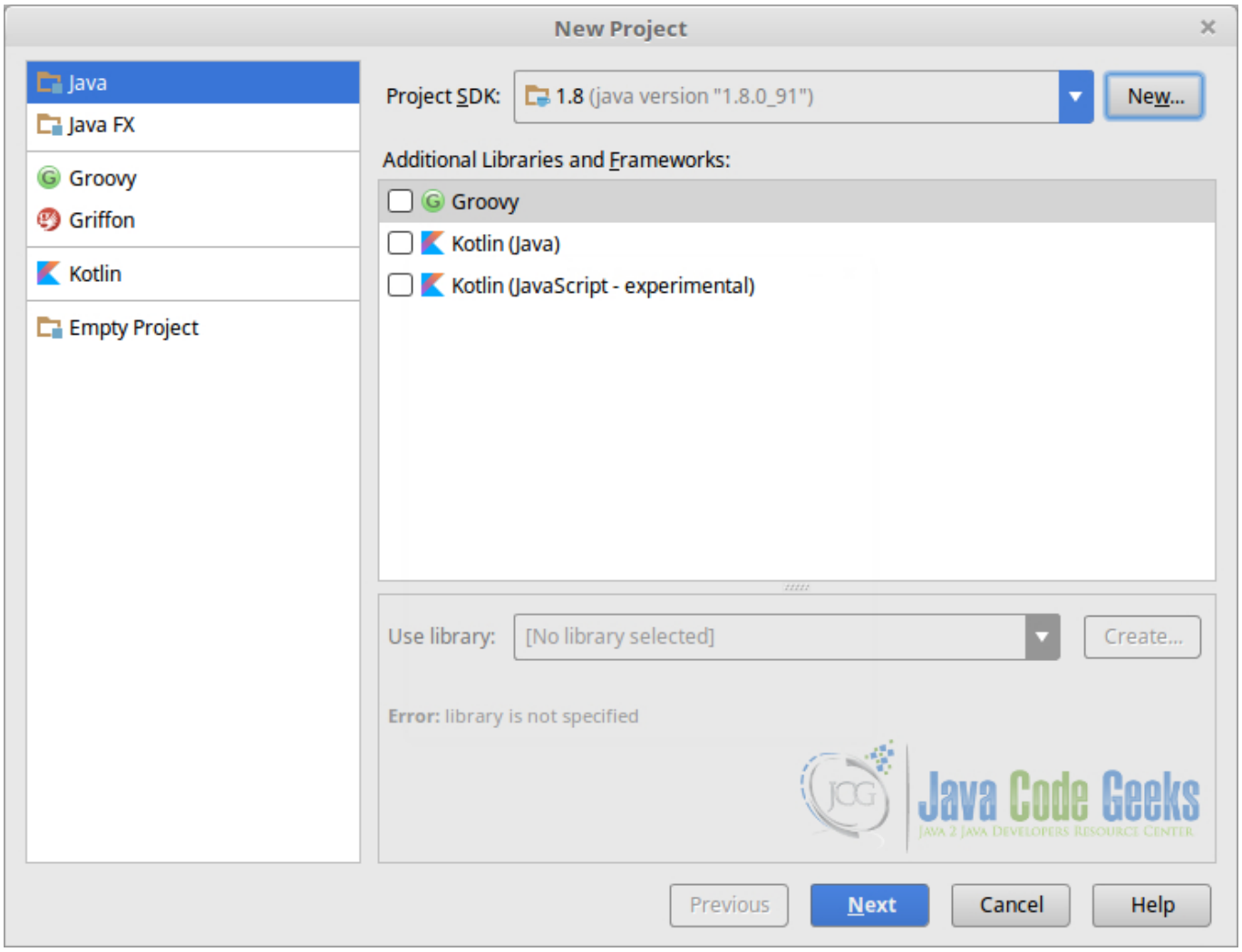


Figure 8.1: Setting the SDK for the project

Then, just select the name for the project, and finish the creation.

Once the project is created, in the explorer (left part), right click the root folder, the one with the folder name.

Select "New/Folder".

Give the folder the name you want; "test" is the standard name for test directory.

Now, right click the tests folder, and select "Mark directory as/Test Sources Root".

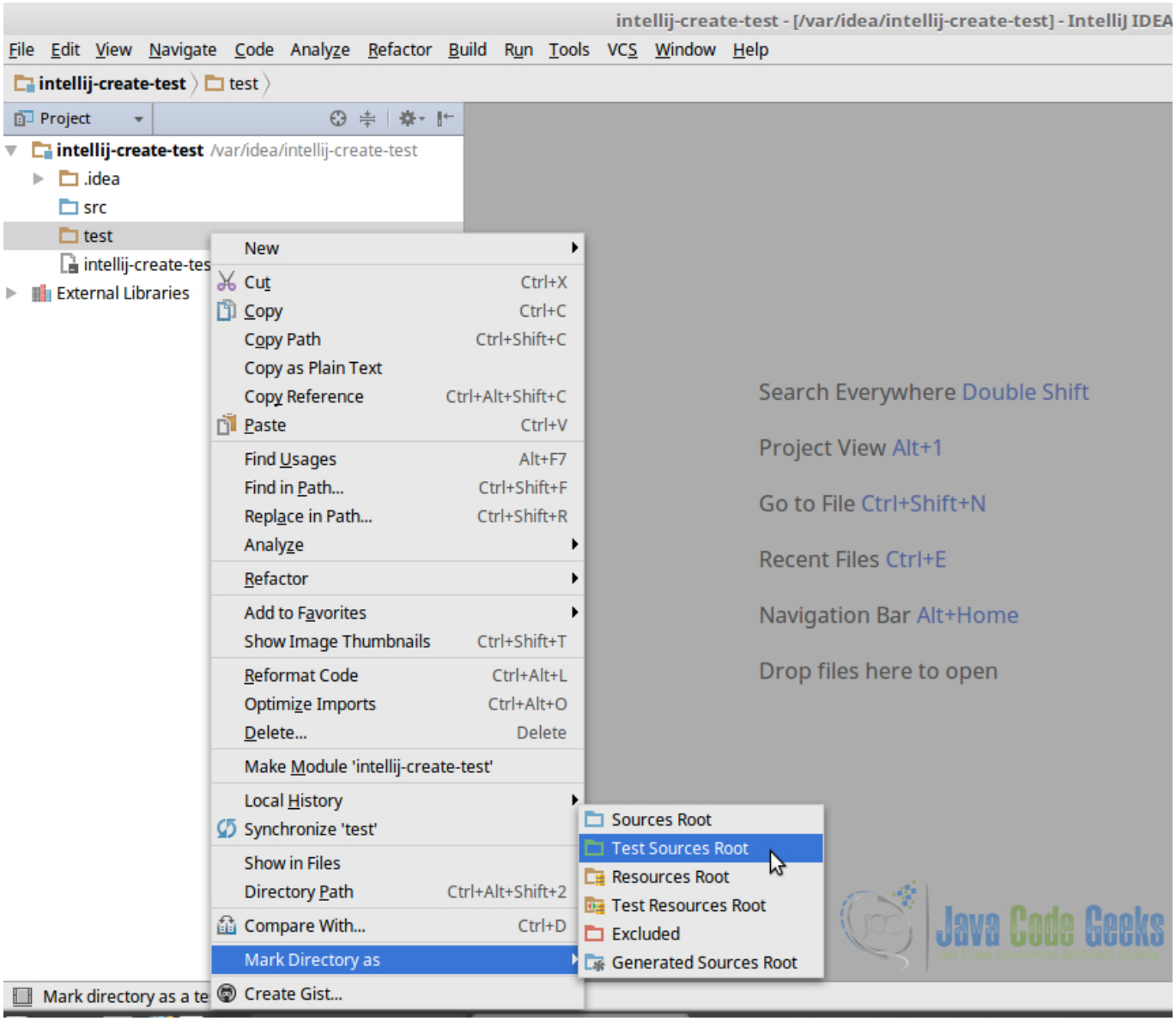


Figure 8.2: Mark directory for tests

Now, the test directory should appear green.

8.2 Base code

We will code a simple class in order to test it later:

SimpleCalculator.java

```
package com.javacodegeeks.intellij.test;

public class SimpleCalculator {

    /**
     * n1 + n2.
     *
```

```
     * @param n1 First number.
     * @param n2 Second number.
     * @return n1 + n2.
     */
    public float add(float n1, float n2) {
        return n1 + n2;
    }

    /**
     * n1 - n2.
     *
     * @param n1 First number.
     * @param n2 Second number.
     * @return n1 - n2.
     */
    public float subtract(float n1, float n2) {
        return n1 - n2;
    }

    /**
     * n1 * n2.
     *
     * @param n1 First number.
     * @param n2 Second number.
     * @return n1 * n2.
     */
    public float multiply(float n1, float n2) {
        return n1 * n2;
    }

    /**
     * n1 / n2.
     *
     * @param n1 First number.
     * @param n2 Second number (divisor).
     * @return n1 / n2.
     * @throws ArithmeticException If the divisor is 0.
     */
    public float divide(float n1, float n2) throws ArithmeticException {
        if ((int)n2 == 0) {
            throw new ArithmeticException("Cannot divide by zero.");
        }

        return n1 / n2;
    }

}
```

8.3 Creating tests

Finally, after having a class to test, we are going to see how we can create tests for it. There are two ways: manually, and automatically.

8.3.1 Manually

In this case, we have to manually create the test class, which in this case is `SimpleCalculatorTest`:

SimpleCalculatorTest.java

```
package com.javacodegeeks.intellij.test;

public class SimpleCalculatorTest {

    @Test
    public void testAdd() {

    }
}
```

The `@Test` annotation will appear in red, because the symbol cannot be resolved, since JUnit is not added to the path. To add it, we have to place the caret in the annotation, and press Alt + Return. A window with several options will appear; we have to choose the first, the one saying "Add JUnit4 to classpath".

After this, we have to choose to reference JUnit to the distribution included with IntelliJ IDEA; or to add it locally to the project, in a directory.

Regardless the option we choose, now the annotation should not be marked in red, having now every JUnit symbols available. So, we could write the following test:

SimpleCalculatorTest.java

```
package com.javacodegeeks.intellij.test;

import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class SimpleCalculatorTest {

    private SimpleCalculator calculator;

    @Before
    public void setUp() {
        this.calculator = new SimpleCalculator();
    }

    @Test
    public void testAdd() {
        float n1 = 1;
        float n2 = 2;
        float expected = 3;
        float actual;

        actual = this.calculator.add(n1, n2);

        assertEquals(expected, actual, 0.000);
    }
}
```

Which should pass if we run it (Ctrl + F5).

8.3.2 Automatically

IntelliJ IDEA is able to generate test classes automatically, in a very fast and comfortable way, specially for those classes with many methods.

To do this, we just have to place the caret, in the class file, in any point of the class declaration statement (i.e. `public class SimpleCalculatorTest`), and press Alt + Return, to show a list of options, where "Create Test" should appear.

If we select it, another window will be shown, in this case, to select the methods we want to test:

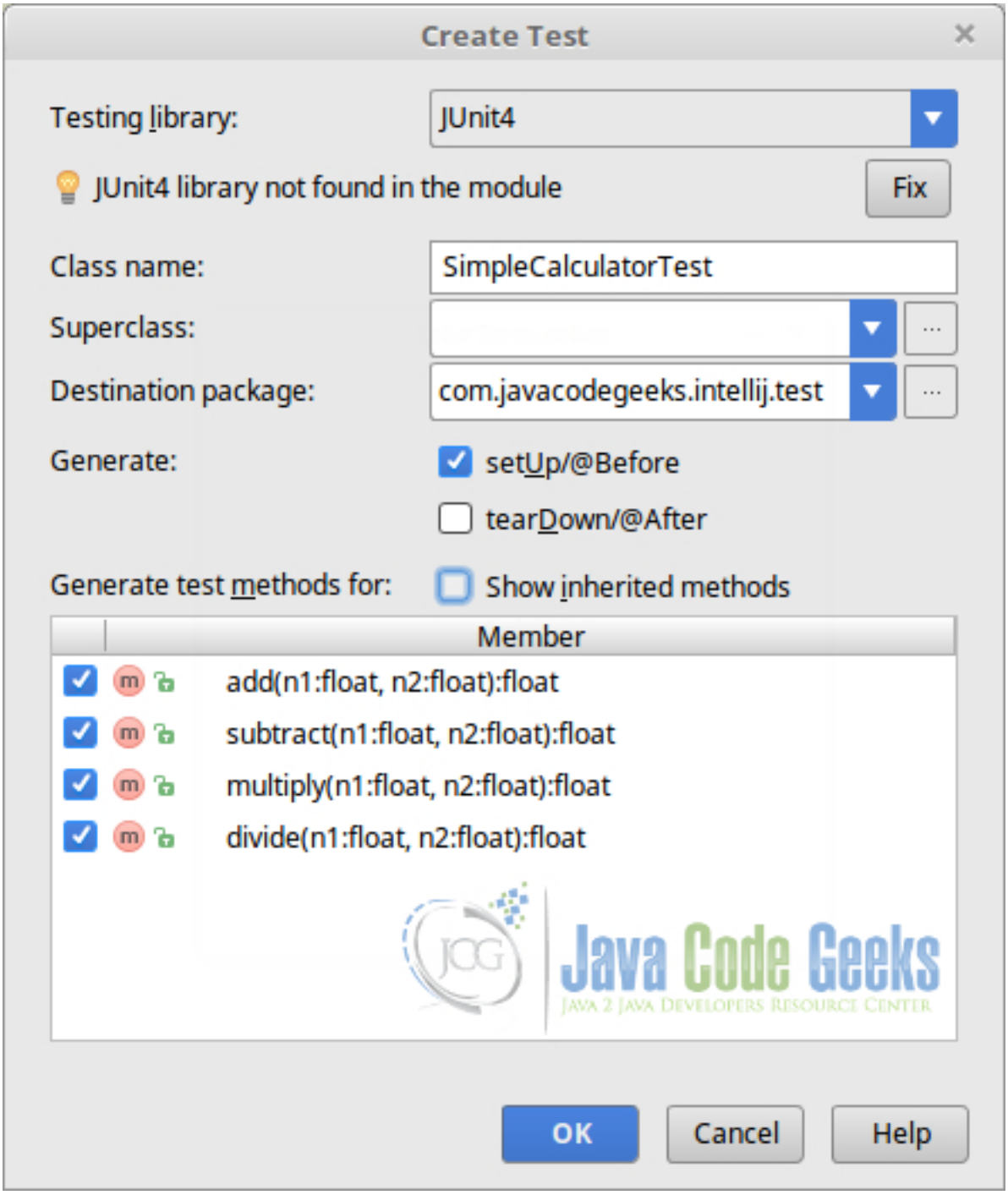


Figure 8.3: Selecting the methods to test

As you can see, IntelliJ IDEA shows every method of the class for which the test methods can be generated. And, also, allows to generate `setUp()` and `tearDown()` methods.

If we haven't JUnit added to the project, IntelliJ IDEA will warn us. If we press the "Fix" button, we will see the same window as when adding JUnit manually, asking about adding the library locally to the project, or using the distribution of IntelliJ IDEA.

Finally, IntelliJ IDEA would politely generate the following test case for us!

SimpleCalculatorTest.java

```
package com.javacodegeeks.intellij.test;

import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class SimpleCalculatorTest {
    @Before
    public void setUp() throws Exception {

    }

    @Test
    public void add() throws Exception {

    }

    @Test
    public void subtract() throws Exception {

    }

    @Test
    public void multiply() throws Exception {

    }

    @Test
    public void divide() throws Exception {

    }

}
```

Note that, for every test, IntelliJ IDEA has added a `throws Exception` clause. This is how IntelliJ is configured by default, and we can change it if we want.

To change this, go to "File/Settings" (Ctrl + Alt + S), go to "Editor/File and Code Templates", navigate to the "Code" tab, and find the "JUnit4 Test Method". Once selected, edit the template of the editor displayed in the right part of the window, removing the `throws Exception` clause, and save the changes.

8.4 Summary

This tutorial has shown how to create tests for our Java projects with IntelliJ IDEA. We have seen two ways, manually and automatically, being the second one the most comfortable for almost every cases; seeing also that we have the possibility to choose to use a reference to the JUnit version packaged with IntelliJ IDEA, or to copy the binaries to a local folder of the project.

8.5 Download the IntelliJ IDEA project

This was an example of test creation with IntelliJ IDEA.

Download You can download the full source code of this example here: **IntelliJIDEACreateTestTutorial**

Chapter 9

IntelliJ IDEA Format Code Example

IntelliJ IDEA is currently, probably, the most popular IDE for Java projects. Among many cool features, one of the most useful is that it allows to format our code in a completely customized way. In this example, we will see how it works.

For this tutorial, we have used:

- IntelliJ IDEA 2016.2.2 Community Edition (free and open source).
- Linux Mint 18.

You can download the IDE from the official JetBrains website, available for Linux, Windows and MacOS.

9.1 Project creation

If it is the first time you open IntelliJ IDEA, you will see that it suggests you to create a new project. Otherwise, create a new file selecting "File/New/Project".

In the new window, you have to select "Java" as the project type, and then select the SDK for the project, as shown in the image below.

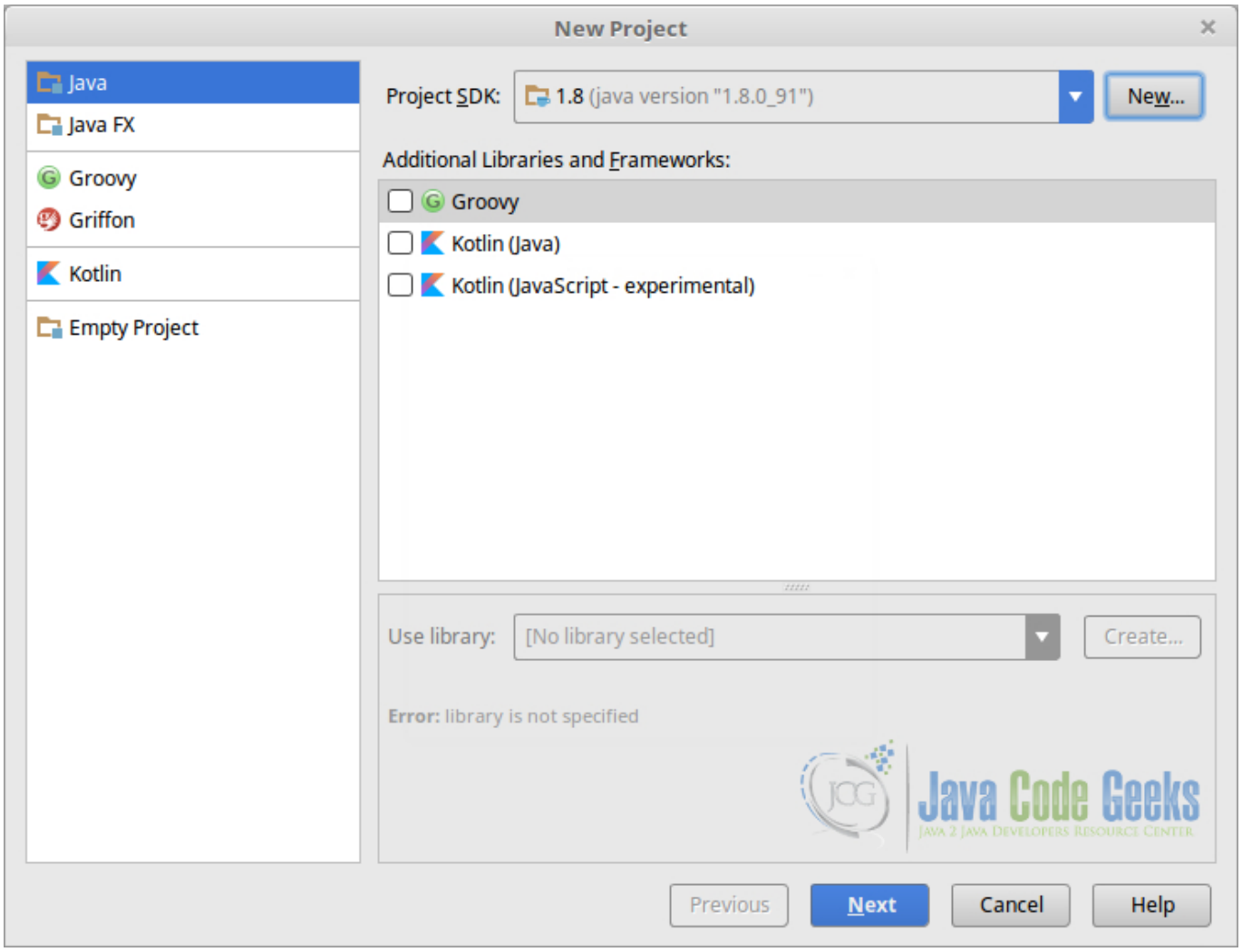


Figure 9.1: Setting the SDK for the project

Then, just select the name for the project, and finish the creation.

9.2 Sample code

Before anything, let's create a sample code. It doesn't even have to make sense, it's just to see how the code formatting works.

For example, we could create a couple of dummy classes like the following:

Bar.java

```
package com.javacodegeeks.intellij.formatcode;

public class Bar {

        public void bar() {
                System.out.println("bar");
        }
}
```

Foo.java

```
package com.javacodegeeks.intellij.formatcode;

public class Foo {

        private Bar bar;

        public Foo(Bar bar) {
                this.bar = bar;
        }

        public void callBar() {
                bar.bar();
        }
}
```

9.3 Formatting the code

The most easy way to reformat the code is with the shortcut: **Ctrl + Alt + L** (or also **Ctrl + Windows Key + Alt + L**, useful for Linux users that have the **Ctrl + Alt + L** shortcut for locking the screen.

So, if we use the shortcut for example, `Bar.java`, it will look now like:

Bar.java

```
package com.javacodegeeks.intellij.formatcode;

public class Bar {

    public void bar() {
        System.out.println("bar");
    }
}
```

Note that IntelliJ IDEA hasn't asked us how it should format the code. It has just formatted it following the default options, which may not be suitable for everyone. Let's see how we can custom the formatting settings.

9.3.1 Format options

The options for formatting the code are defined in "File/Settings/Editor/Code Style". Under this last option, there's a code style scheme for different file types, but we will just look for the Java ones.

The code style options are categorized in the following categories:

- Tabs and indents: the configuration for the indentation, i.e., the indentation of the code blocks. You can also choose to use the tab character, or several spaces.
- Spaces: this section allows to configure the white spaces before and after the control structures and operators, for almost every possible option. For example, to set a white space before the opening curly brace `{`, after commas in method definitions/calls, the assignment operation =, and a large etc.
- Wrapping and braces: similar to the previous one, but in this case wrapping the lines, for control structures curly braces, method calls, etc.
- Blank lines: here we can configure the number of blank lines between different "sections" of the code, for both the minimum blank lines to apply, and for the maximum lines to keep. We can, for example, the configure the number of lines for before and after the imports, the lines between each method declaration, after the class header, etc.

- JavaDoc: for the code documentation. For setting the options for the alignment of the parameter descriptions and blank lines, among others.
- Arrangement: here there are two different sections: a) Grouping rules: how to make groups. We can configure to set together the getters and setters, the overridden methods, and the dependent methods. b) Matching rules: for ordering the fields, methods, etc. depending on their modifiers. An example is shown in the 2nd image.

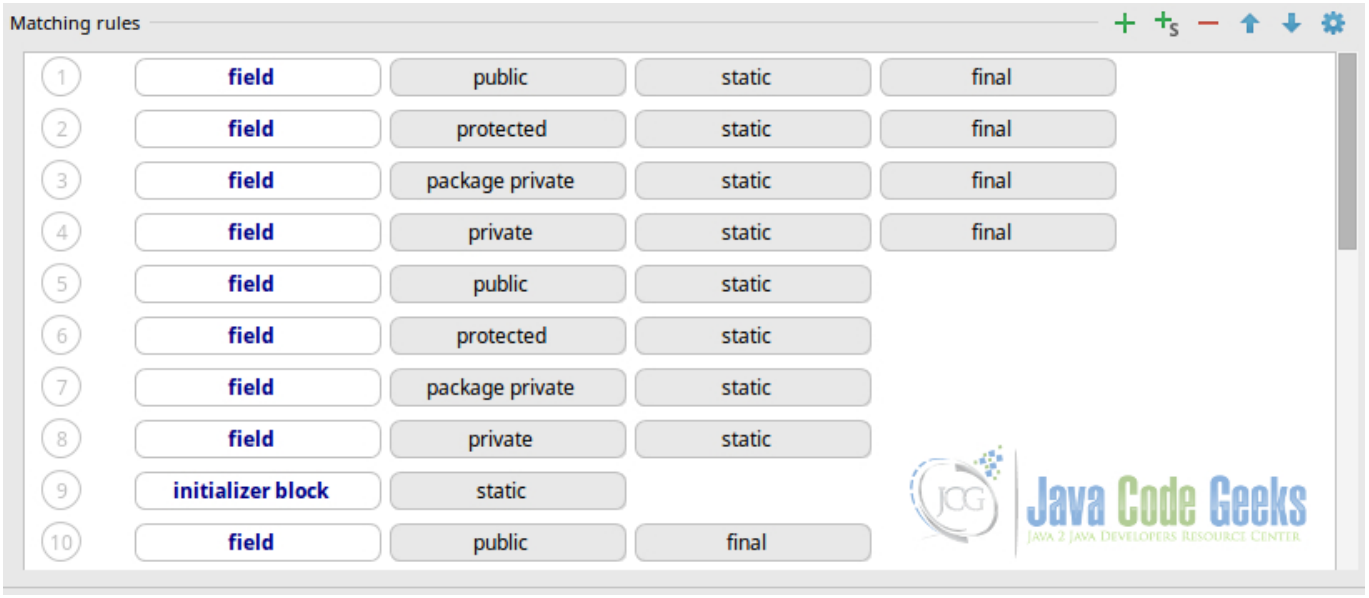


Figure 9.2: Arrangement

This means that, if we reformat the code with this options, inside the class, in the first "position" would go the fields matching `public`, `static` **and** `final` modifiers; in the second, the fields matching `protected`, `static` **and** `final`; and so on.

For rearranging the code, the code formatting shortcut won't work; we have to select "Code/Rearrange code" option. So, for example, if we would have the following code:

Foo.java

```
package com.javacodegeeks.intellij.formatcode;

public class Foo {

    protected static final int b = 2;
    private static final int c = 3;
    public static final int a = 1;
}
```

After rearranging, the resulting code would be:

Foo.java

```
package com.javacodegeeks.intellij.formatcode;

public class Foo {

    public static final int a = 1;
    protected static final int b = 2;
    private static final int c = 3;
}
```

9.3.2 Formatting markers

For some reason, we may want to keep chunks of code away from the IntelliJ IDEA formatter. IDEA allows to ignore code pieces using formatting markers.

To use them, we first have to enable then, in "File/Settings/Editor/Code Style", in "Formatter Control" section. There, we have to choose the markers. By default, the markers are `@formatter:on` and `@formatter:off`, that have to be used in an inline comment.

So, for example, returning to our original `Foo.java` class, using the markers:

Foo.java

```
package com.javacodegeeks.intellij.formatcode;

public class Foo {

private Bar bar;

        //@formatter:off
        public Foo(Bar bar) {
                this.bar = bar;
        }
        //@formatter:on
        public void callBar() {
                bar.bar();
        }
}
```

If we format the code, the result would be:

Foo.java

```
package com.javacodegeeks.intellij.formatcode;

public class Foo {

    private Bar bar;

        //@formatter:off
        public Foo(Bar bar) {
                this.bar = bar;
        }
        //@formatter:on

    public void callBar() {
        bar.bar();
    }
}
```

9.4 Summary

This tutorial has shown how works the IntelliJ IDEA code formatter, seeing how we can format our code with a simple shortcut, and also customizing the code style to apply in a very complete way, from white spaces and line wrappings, to JavaDoc and code arrangement.