Eclipse Cheat Sheet

(version 2.0)
Updated Mars, 2021

©Pr. Olivier Gruber
Université de Grenoble-Alpes

This document is to kick start you using Eclipse and give you my personal tricks and tips that boost my productivity, simplifying Java development with Eclipse. Many of those tips are not specific to Java.

This document is to help you work faster. Working faster means more time for fun but also means learning far more and far better. Eclipse can really help you work faster, simplifying many of the repetitive tasks of programming. So use it. But Eclipse is also a complex environment that takes some time to master. This document will fast track you to some of the really cool features of Eclipse.

For your information, NetBeans usually has very identical features, so if you are a NetBeans user, search for them and improve your coding experience. Probably other tools have similar features.

Get Started

To get started, rely on the Web. There are just countless tutorials on the Web, including of course the Eclipse tutorial from the Eclipse community.

http://help.eclipse.org/

Just make sure that you are looking at the documentation and tutorials for your Eclipse version.

Installing Eclipse, see the web for that.
Still, which version of Eclipse should you install? Usually, the latest is the best, but if you are updating to a new version of Eclipse, you might give it a try before jumping in. But if you hit some bugs, try out an earlier version. It has happened in the past that Eclipse shipped new versions with major bugs, but it has been rare. Overall, the Eclipse team and the plugin teams are doing a fantastic job. Thanks guys! And it is free, you cannot argue against that.

Yet, you have different initial installs to choose from. I would suggest to the get plain Java version. It contains only Java tools, which are all used internally by the Eclipse teams... so I found it to be always the most stable version. Then, if you are a C developer, you can always later install the CDT plugin for C/C++ development tools. You may also add later other tools and other programming languages such as JavaScript, Python, etc.

If you install the full Web install, it will be huge... but at some point, you may need to do Web development. I would suggest to have two Eclipse installed. One for Java/C developments. One for Web development.

Updating Eclipse, see the web for that.
Eclipse is a platform with plugins, as such, it offers you the ability to install new plugins or updates existing ones. Installing from the Eclipse market has not been a satisfactory experience for me so far. I prefer a more “controlled” approach. Look at the help menu

Help → Check for updates
Help → Install new software

You will have a much better control on what is available and what you want to install.

Glitch with Eclipse and the openJDK

Sometimes, you will hit bugs with the openJDK (mostly on Linux installs). I would suggest that you install the Oracle JDK. Use that JDK to run Eclipse itself. Setup that JDK as your JAVA_HOME and add it to your PATH:

```
$ export JAVA_HOME=/your/path/to/your/jdk
$ PATH=$PATH:$JAVA_HOME/bin
```

Of course, edit the path for your JDK in the lines above.

ALWAYS INSTALL A JDK – DO NOT INSTALL A JRE!

Even if you do not hit bugs, you shall install a JDK from Oracle. With a JDK, you will get the sources for all the JDK classes, it will prove really usefull, trust me.

Note: on many distribution installers, the sources are available from a different package than the package for the JDK.

Also, with a JDK, you will get the JVisualVM tool, again, that tool will prove really usefull. The tool shows you everything you need to know about the currently running Java Virtual Machines on your system. As a beginner, it is unlikely you will use it, but really rapidly you will. Trust me.

See below, but also make sure that your Eclipse knows about the JDK:

Windows → Preferences → Java → Installed JRE

Select your JDK as the default Java environment.

Also setup the Java compliance level for your Java project,
so that it matches your installed Java JDK:

Windows → Preferences → Java → Compiler

So for instance, install a JDK 1.8, Standard Edition, and set your compiler setting to Java 1.8. If your Eclipse is old, it may not support Java 1.8, in which case you should either upgrade your Eclipse or downgrade your installed JDK.

Notes for C/C++ developers:

You may want to install the CDT plugin for C/C++. The Eclipse C/C++ tools are pretty good, yet not as good as the Java ones. But it certainly beats text editors.

However, you may want to also have DDD installed as a front-end to GDB for C debugging. Eclipse CDT does include gdb support, and for most tasks, it is very acceptable. However, they made the mistake of hidding the gdb console. I guess the Eclipse team felt that an all-gui experience was necessary, I disagree. DDD gives you the gdb console and the console is a must if one is doing serious C/C++ debugging.

Which Eclipse tutorials should I read?

Get started on the Web, there is everything you need available. No excuse whinning that you had no lectures on Eclipse. Learning Eclipse and other advanced tools will benefit you. Yet, since the amount of tutorials and documentation on Eclipse is large, which may be intimidating, here is a fast-track path to help you focus on the important stuff first. Start reading the user guides on the basic Eclipse concepts on the Web:

Concept of Workspace.
Concept of Perspective and views
Concept of Project.

And also about how to update/install software in Eclipse. Do not use the Eclipse market, I was not really pleased with it so far, but you may have better luck. Otherwise, look at the Help menu, with options to install

new software or check for updates.

With that, you will already have a good start to navigate your way around Eclipse.

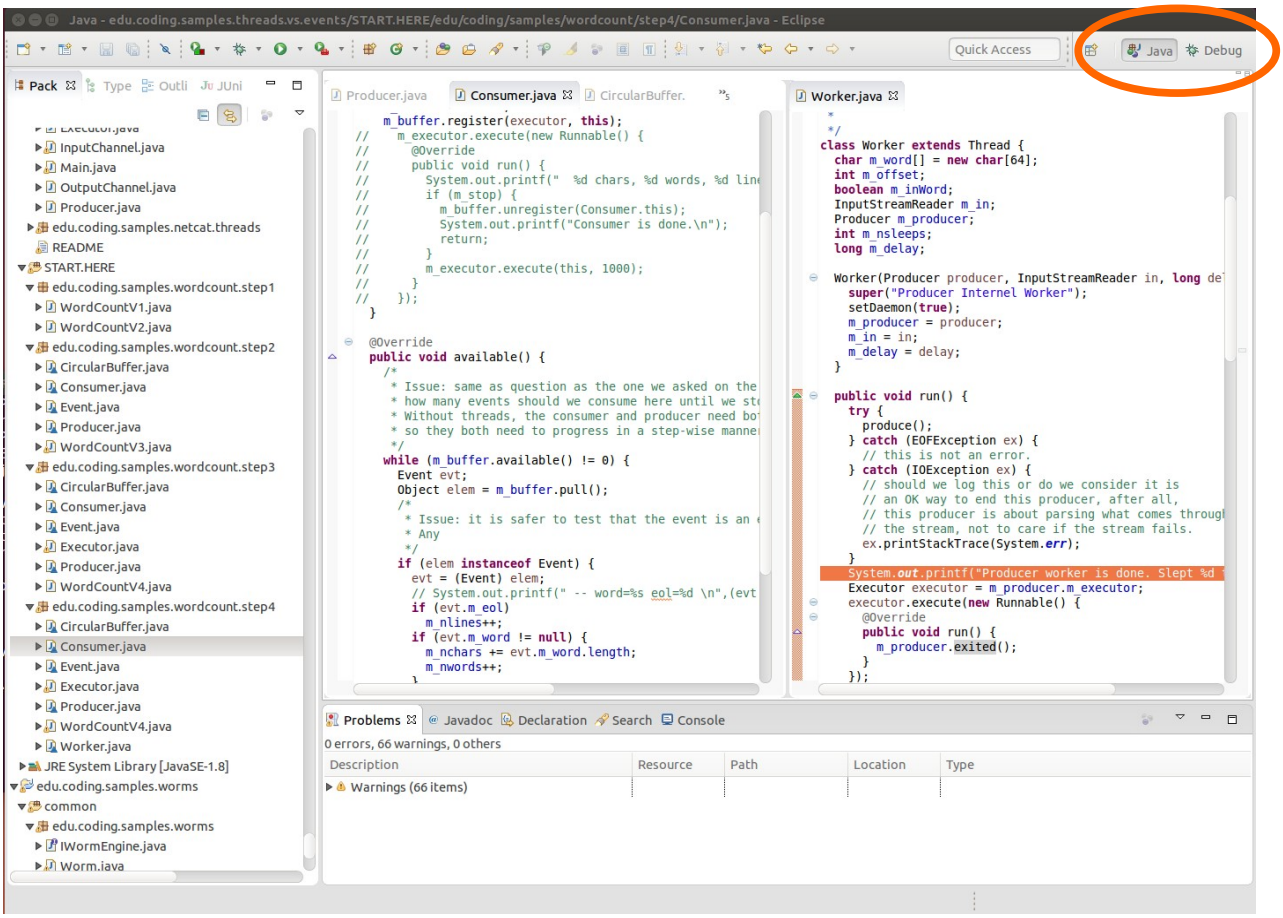
Key Points on Perspectives/Views:

Here is a minimal summary, with a few highlighted points that will be very important for you. Eclipse's core concepts are the Workbench, with perspectives and views, and the Workspace with projects. The Workbench is the graphical user interface in a sense and the Workspace is where your projects with your code are.

The Workbench experience is based on perspectives and views. A perspective corresponds roughly to a theme-oriented interface, such as a Java perspective for developing in the Java programming language or a C perspective for developing in the C programming language. A perspective can be shared, such as the debugger perspective that enables you to debug your programs written in various programming languages.

Perspectives are composed of views, a view being a user interface components that display useful information. For instance, the Java perspective has a Navigator view for looking at your workspace in a Java-centric way. It also has editor views, along with an outline view for example. A debugger perspective would have a stack view, a variable view, and an editor view to view the sources. Of course, the editor view in both the Java and Debug perspectives are the same. In fact, you can open any view in any perspective.

Below a Workbench, on the Java perspective, but with a debug perspective in the background. Notice that you can have multiple classes open at the same time, even have two editors side by side. Drag and drop views where ever you like on the screen, you can reorganized them as you please.

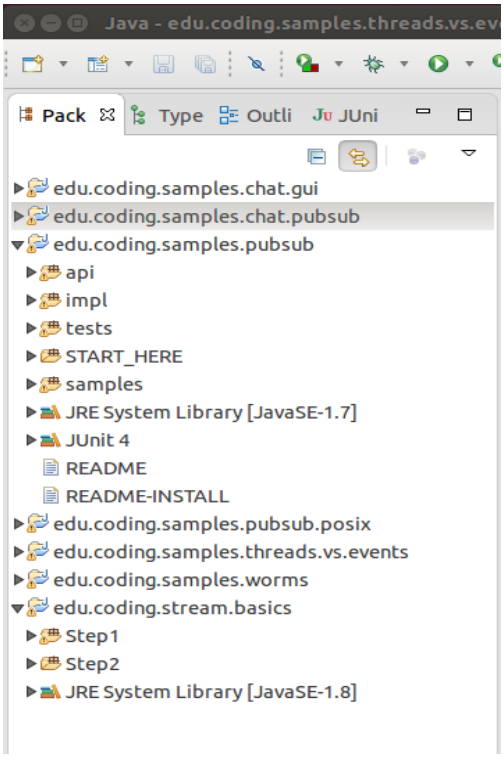


The Workspace experience is based on the concept of projects. A Workspace lives essentially in a directory, where immediate sub-directories are projects. Here we have a multiple workspace layout, each workspace with several projects.

```
~/Workspaces/
        MyWorkspaceA/
                ProjectA/
                ProjectB/
        MyWorkspaceB/
                ProjectA/
                ProjectB/
                ProjectC/
```

You can open multiple Eclipse instances on different workspaces, or you can toggle workspaces within a single Eclipse instance.

Notice the double arrow at the top of the Package view, it synchronizes the editor and the Package view. In other words, as you change between edited Java classes, the Package view will automatically scroll up or down to the right project, package, and class. So you will always know where the class you are editing is.



Java Developers

Eclipse has an excellent Java support, since all the Java tools are used in-house by the Eclipse developers. Sometimes, Eclipse can be a little hard to master, but it will deliver for Java developers.

Look on the Web for tutorials on :

Creating a Java project, building it, and launching it.

Read also about the Java debugger and how to use it.

Creating a Java project is the easiest thing in Eclipse, just use the Eclipse wizard

Menu: File → New → Java Project

By default, Eclipse will create a source folder (src) for holding your classes and a class folder (bin) for holding your compiled classes (classfiles). Make sure you switch to the Java perspective (it should happen automatically though) so that you get the package viewer on the left-hand side of your Eclipse window.

Although Eclipse manages your project entirely and you can do everything from within Eclipse, it is usefull to understand how Eclipse lays your project on your file system. One of the main reasons is to be able to grab exiting sources and integrate them into an Eclipse project for faster browsing, compiling, and debugging.

Default Setup: when creating a new Java project, if you choose all the defaults, you will have the following typicaly layout on the file system:

```
MyWorkspace/
        MyProject/
                src/
                bin/
```

Under the src directory, you will have a tree of directories matching your Java packages. Within these directories, you will find your classes (.java files, your sources). The class files (.class files) generated by the Java compiler (javac) will be found under the bin/ directory, using a similar layout. Let's assume we have two classes (Main and Toto) in the Java package edu.uga:

```
MyWorkspace/
    MyProject/
        src/
            edu/
                uga/
                    Main.java
                    Toto.java
        bin/
            edu/
                uga/
                    Main.class
                    Toto.class
```

To import sources to a project:

Create a Java project, in your workspace. Go back to your host operating system, Linux or Windows, and just create the proper hierachy under the source directory (src), using any Linux/Windows tool. You may unzip sources from a given zip archive for example.

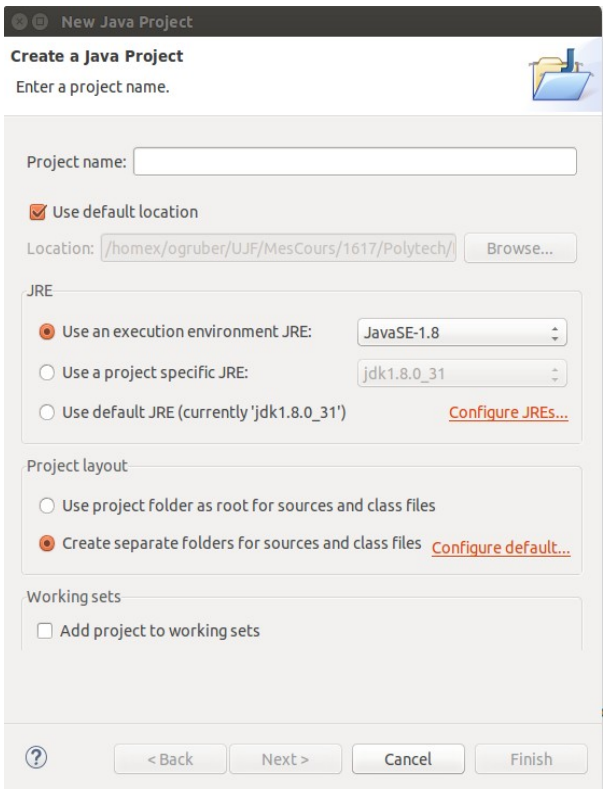
Then, go back to Eclipse and **refresh your project** (right-click on the project name and select refresh or select your project and hit F5). Refreshing tells Eclipse that the file system has changed and it needs to rescan the source directory for added or removed classes and packages.

In some rare cases, you may have to clean your project (Menu: Project → Clean) before it rebuilds correctly. This is extremely rare, so in general, just refresh and ask for an immediate rebuild (Ctrl-B).

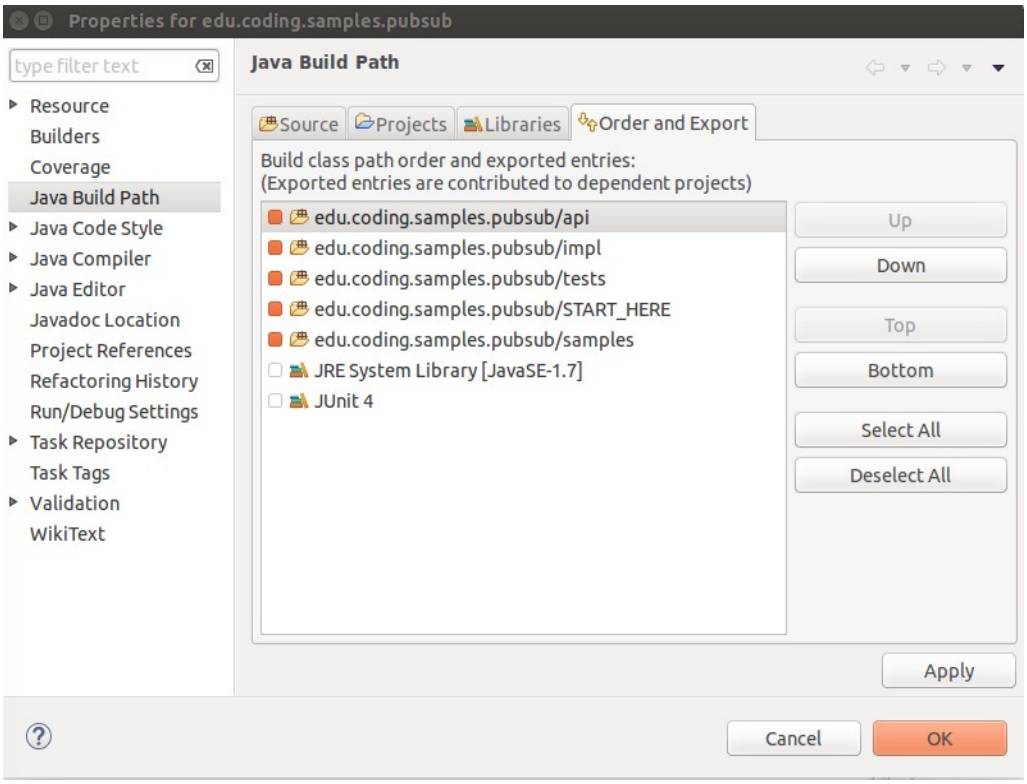
A few very important point about creating a Java project:

1. Choose the version of Java you want to write your code in. Java has evolved, from its version 1.0 to its current version 1.8.
2. Make sure your project is configured to compile to that version of Java. See your project properties for that.
3. Make sure your project runs on a JDK that is compatible with the version of Java you chose.

Notice on the right hand side, when creating a Java project, the Java Wizard asks which JRE. Here, you see the version that is selected, a JavaSE-1.8, that is, a Java Standard Edition in version 1.8. This is the runtime in which your classes will be loaded and executed.



On the figure below, you are looking at the Properties of a Java project. You can see many different property domains on the left panel: Java Build Path, Java Compiler, etc. This is the place where you control all aspects of your Java projects. This is a place that you want to explore and understand.



We are looking at the Java Build Path, which controls how you classes are built. The Order_and_Export tab is a summary tab. We can see that this project is setup for JavaSE-1.7 (look at the JRE System Library towards the end of the list). We can see that it uses multiple source directories (api, impl, tests).

Using the Source tab, one could see all the source directories and the output directory for the classes.

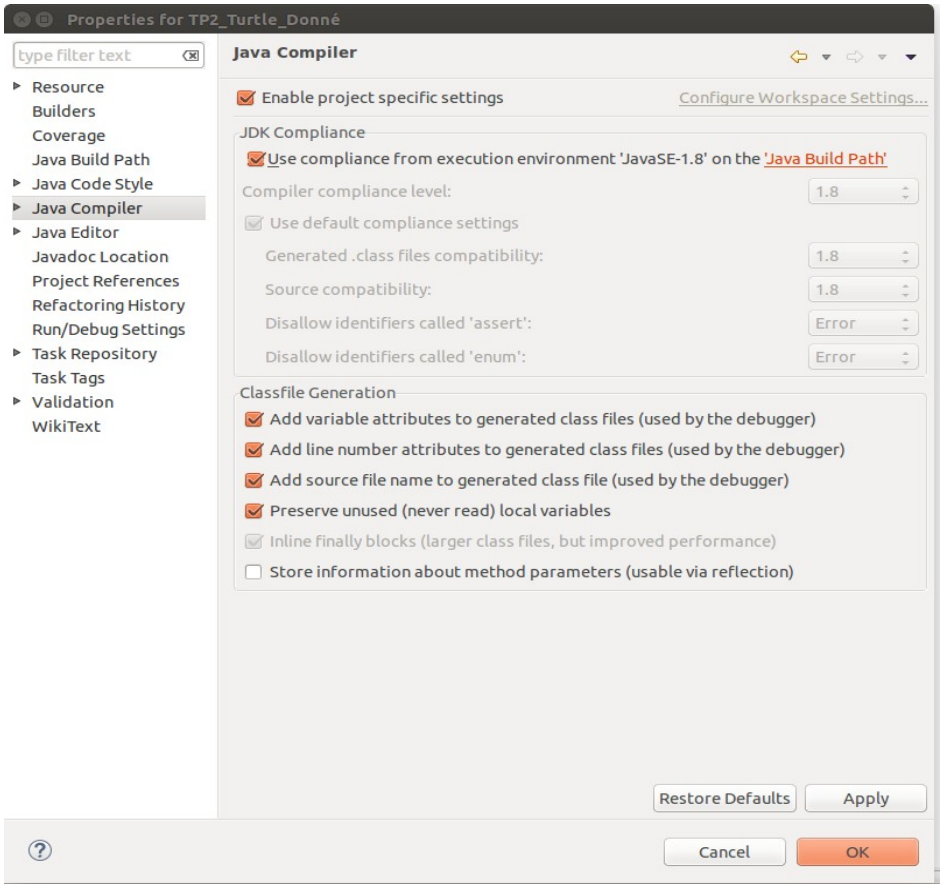
Using the Libraries tab, one could add jar files as libraries to this project.

Using the Projects tab, one could add dependencies to other Java projects, meaning that your classes in your project could use classes from other projects in your workspace.

From there, it is important to check your compiler setttings

Select the Java Compiler properties. You can see that you can either use the global settings from your workspace or have project specific settings.

With this project, we have a Java-SE-1.8 JDK compliance, this means that your classes will be compiled for Java 1.8. This requires to run on a JRE-1.8. If you are setup up with a JRE-1.7 as we were in the previous page, you will get errors when executing your code.



Command Line Tools:

Sometimes, you may wish to compile and execute manually, although it should be a rare situation. To compile manually, you will use the Java compiler (javac). To know how to use javac, you can use the Linux on-line manuals

```
$ javac -help
Usage: javac <options> <source files>

where possible options include:

  -g                         Generate all debugging info

  -g:none                    Generate no debugging info

  -classpath <path>          Specify where to find user class files and annotation processors

  -cp <path>                 Specify where to find user class files and annotation processors

  -sourcepath <path>         Specify where to find input source files

  -d <directory>             Specify where to place generated class files

  -s <directory>             Specify where to place generated source files

  ...
```

For compiling a class "*Main*" in a Java package "*edu.uga*"

```
$ cd my-workspace/my-project
$ mkdir bin
$ javac -d ./bin -s ./bin -sourcepath ./src src/edu/uga/Main.java
```

Notice that we tell the java compiler (javac) to read sources from ./src and to output class files in ./bin. Notice that we need to give the correct full path to the Java source we want to compile, including the src prefix (**src**/edu/uga/Main.java).

You may also compile all the classes in the "edu.uga" package

```
$ javac -d ./bin -s ./bin -sourcepath ./src src/edu/uga/*.java
```

You may have already compiled classes, in jar files for example, in which case you will use the "-cp" option to specify where to find those extra class files.

Of course, once you compiled your Java sources, you may want to execute your program. From the same place (the *src* directory)

```
$ java -cp ./bin edu.uga.Main
```

Notice that we use the Java name this time for the class (edu.uga.Main) and not the file name (bin/edu/uga/Main.class). Don't we love those idiosyncrasies!

WHY SHOULD YOU USE A DEBUGGER?

Why should you invest a few hours of your time learning a debugger? Because you cannot call yourself a serious software developer if you do not know how to use a debugger. The debugger should be the first tool you master right after the compiler.

A debugger will help you learn Java programming faster.

A debugger will help you fix bugs faster.

The Eclipse Java debugger is just awesome.

So go read on the Web the Eclipse tutorial on Java debugging. Then, go down below and read in this document the section on more advanced features of the Eclipse Java debugger.
One of the main reasons for using Eclipse... the Java world is extremely well supported and in particular Java debugging. One would need hours to cover all the great useful productive stuff one can do with a debugger. Here some starters, be inquisitive, and you will find new stuff regularly.

First and foremost, ****USE**** the debugger. If you are not doing performance measurements, **you must always run your programs under the debugger**. **Why?** Because that way, whenever a bug occurs, you can debug it right there and then, without having to worry about how the heck did you make that bug surface and to be able to re-produce it.

Right-click on a class that defines a main method, and you can launch it under debugger. Once you have launched it, you can do so again from the “bug” icon (the one icon that looks like a round bug with legs).

Note: if you click on the right-hand side of the bug icon, on the arrow, you get a pull-down menu that allows you to choose which debug configuration you want, but also you can go and edit the configurations.

Debug configurations are very important. You can have many, with many different names, allowing you to start the same Java program with different settings, such as different arguments for your application and for the underlying JRE.

Note: you can also create or edit debug/run configuration from the project properties (Alt-Enter), under Run/Debug Settings.

Note: you can debug multiple programs at once with Eclipse. This is especially useful when debugging distributed code across multiple JREs.

A debugger will help you getting your code off the ground quickly. Just execute step by step and catch all the minors bugs and mistakes one does when writing some new code. You know, null pointers, almost-correct string manipulations, forgotten field assignments, etc.

Notice that you can run incomplete code in Java, so you can start writing and start debugging, even though all your code is not there. Nice to try out stuff quickly.

After single-stepping, the most important functionality is setting breakpoints. A breakpoint is a place where execution will stop. Just left-click on the right-hand-side of any source line and you will set up a breakpoint on that line. Under the Debug perspective, you can look at the breakpoint (open the breakpoint view: Window → Show View → Other → Debug → Breakpoints).

IMPORTANT: You can setup breakpoints on exceptions. From the breakpoint view, select the ! icon, enter the name of an exception class, and choose caught/uncaught, and that's it. For instance, **you should always have a breakpoint set for NullPointerException, caught and uncaught**.

IMPORTANT: you can also set a condition on a breakpoint, like only breaking the execution when the index of a loop reaches a certain value... To set a condition, go in the "Breakpoints" view, select your breakpoint, and add the condition in the properties of your breakpoint.

In the call stack, you have monitor information also that are extremely useful when debugging multi-threaded applications. In particular, this always to see deadlocks and understand them. Don't forget to “pause” your threads (the || icon on the debugger bar).

Dynamic Software Update (DSU) is also a powerful tool... while you debug, you can find and understand a bug, fix it and recompile, the JRE, while still under the debugger, will dynamically update your loaded classes in your running JRE. They are limitations, but in many cases, you can just keep going with your debugging session. Key limitations to know about:

- Changing the structure of a class, especially adding fields does not well.
- Changing polymorphism also does not work well in many cases.
- If you change a method body that is currently on the stack, most of the times it works, sometimes it does not.

Miscellaneous:

Multi-project setup: you can use a multi-project setting, separating your own code in different projects, or leveraging existing projects and extending/using them from other projects. At first, there is no need to do so, when beginning to code in Java. But ultimately, you will get to a situation where you will need to use multiple projects.

Again, go to your project properties (Alt-Enter) and search for Java buildpath pane. Notice that you have a project tab, next to the source or library tabs. You can add other projects, which makes the classes of those projects visible in yours. You can even create cycles across several projects, although Eclipse will ask you to confirm that this is indeed what you want.

To rename a project on disk: Sometimes, you will need to rename a project, like anything else gets renamed one in a while. When renaming classes or packages, just hit F2, type in the new name, and Eclipse will manage everything for you.

But unlike for packages and classes, renaming a project does not change the name of the underlying directory. To do this, you need to rename it in Eclipse, which changes the name by which Eclipse displays it. Then, delete it from your workspace, but KEEPING the underlying resources. Change the underlying directory name in the file system. Then, re-import your project in your workspace.

Pretty printing (Ctrl-Shift-F)
For Java, you can setup pretty printing preferences. Fixing pretty printing is important, it will help you read code and also it is much easier to let Eclipse do the pretty printing that do it yourself manually. Also, working with a team, and using a source control like GIT or SVN, all team developers must use the same pretty printing to avoid false-positive changes when using source control (SVN or GIT)

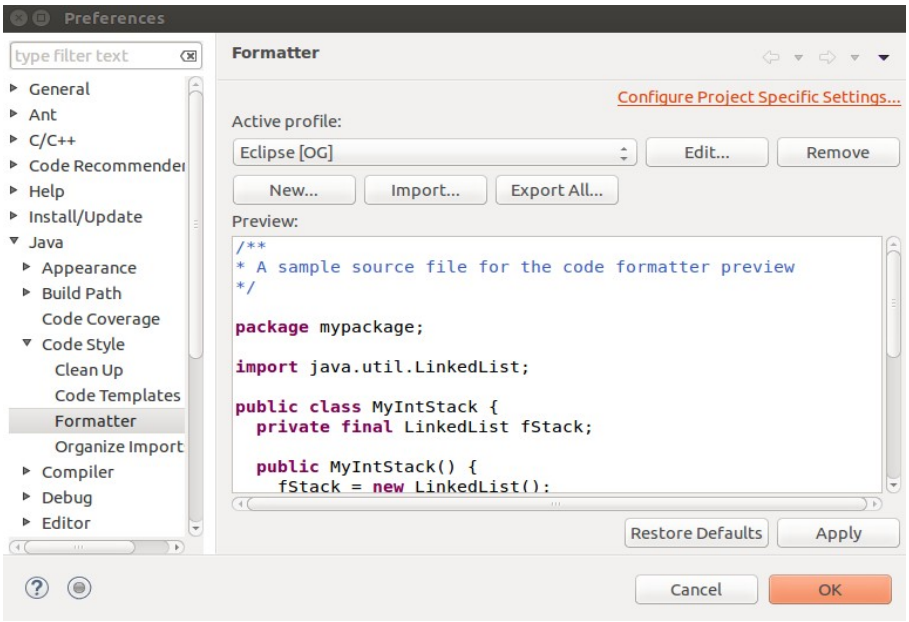
Menus: Windows → Preferences → Java → Code Style

You will need to edit the active profile, changing its name. Notice that I called mine [OG] as opposed to the [built-in] you will see the first time you hit that preference panel.

When you hit the Edit button, you will get another window will very complete preferences on how you want your Java editor to format/present your code.

I usually use 2 spaces instead of tabs of 4 spaces, source code is more compact.

I also remove the auto-formatting of my comments.



Dealing with Eclipse crashes: when your Eclipse will crash, it may refuse to restart, complaining that your workspace is already locked... then you need to remove the lock file in the .metadata directory in your workspace:

```
$ cd workspace
$ rm -f .metadata/.lock
```

If all is lost, you can remove the .metadata directory entirely. Relaunch Eclipse in the same workspace directory and reimport all your projects.

Most useful shortcuts:

Ctrl-F: search (and replace)
Ctrl-space: content assist
Ctrl-Shift-F: pretty-print (reformat your source).
Ctrl-Shift-O: automatic management of your imported classes/packages.

Ctrl-O: outline as a popup, quick navigation to Java items in the current source)
Ctrl-Shift-T: search for a class (wildcards do work)
Ctrl-Shift-G: select something, you will see where it is used
for example, select a method and you will see all the places it is called from.
Ctrl-E: select a tab.
Ctrl-Shift-W: close all opened tabs... cool once you have so many opened that you are lost.
Ctrl-B: rebuild all.

F3: select a Java item (class, variable, or method) and navigate to it.
Look for the yellow arrows in the tool bar to have the ability to navigate back from hitting F3.
F4: select a class and see its hierarchy (super-classes) and sub-classes
this is especially useful when wanting to know which classes implement an interface.

Select a class that implements an interface or extends a super class, and for which the compiler complains that some methods are not implemented...

Crtl-1 → Add unimplemented methods
Voilà !

Alt-Shift-R: enables the renaming of the currently selected item (class, method, variable)
Note: once you typed the new value, you can hit escape or return. Return will trigger a global replacement throughout all your classes. Escape will keep the changes to the local file, but it is much faster. So for local changes, such as changing a parameter name, a local variable name, or a private method name, always hit Escape, it is so much faster.

Degug:
F5: step in
F6: step over
F7: step out
F8: go
F11: relaunch last launch (sometimes a bit tricky because it is contextual).

Most useful functionality:

Refactoring: read about it on the web... Eclipse will be your best friend when you master refactoring...

Synchronize your navigation through your code with the Navigator view.
Look for the two-arrow icon in the Navigator view. If selected, whenever you click within a source file, the Navigator view will repaint itself around the corresponding resource. Very useful, but sometimes, it is just better not to be synchronized. See what suits you the best.

Sorting the outline view.
Look the a/z icon to change how items are sorted in the outline view of a class. You can sort by

alphabetical order or you can leave the items in the order in which they are defined in the source.

Building a project (Ctlr-B)
The default is that Eclipse builds your project automatically. But sometimes, you may setup projects to build only manually. Sometimes, you will need to force a rebuild. Sometimes, you will need to force Eclipse to do a complete cleanup before you recompile everything. Most beginners will not have to do this. Look that the Project menu:

Windows → Project → Build / Clean / etc..

Using multiple editors:
By default, Eclipse shows you only one editor, in the center of the Eclipse window. You can use more than one, simply drag a source tab around and you will see that you can dock a new editor in different places.

Also, you can split one tab into multiple regions:

Windows → Editor

You will be able to split vertically or horizontally, with the different panes showing you different regions of your source code.

Comparing files:
You can select multiple classes or files in the navigator view and compare them.

Right-click → Compare

Synchronize editing with the Navigator view:
The idea is to have the navigator view (package or files) to show you the file you are currently editing or not. So as you edit one tab or another, the nagivator view will scroll up or down to show you where the file you are editing is. At the top of the navigator view, upper right corner, there is a dual yellow arrow → synchronize/de-synchronize the navigator with the currently selected source.

Global search → toolbar → the icon with a torch lamp.

Refactoring
You must read about Eclipse refactoring on the web. Guaranteed, you will love it.

Eclipse as built-in support for JUnit:

When developing, you are writing your code, then you are debugging it, usually running some basic tests. Most of us approach testing through ad-hoc small programs... They tend to get lost... and more importantly, as your software evolves, you are not running the same tests systematically... so correcting new bugs might re-introduce bugs in parts of your code that you tested already.

JUnit is useful for two main reasons. First, it gives you a simple framework to structure your tests. Second, it gives you the ability to run all your tests easily and get a report of what is running and what is not.

http://junit.sourceforge.net/doc/cookbook/cookbook.htm
http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2FgettingStarted%2Fqs-junit.htm

Eclipse also as a plugin for coverage:

Now, you wrote a number of tests, you are running them regurlarly, but how do you know that you tested everything? This is where covereage comes in. Run your tests, with coverage, and you will see which lines are code have been tested and which ones have not been tested.

→ Project right-click → Coverage As → Java application or JUnit test
→ Get hightlighted sources and a report coverage.
→ In EclEmma report view, can delete the current session (with the cross button).

http://www.eclemma.org/
http://www.eclemma.org/installation.html

Note that Cobertura is also an interesting coverage framework:

http://sourceforge.net/projects/cobertura/?source=typ_redirect
http://cobertura.github.io/cobertura/

Last but not least:

Oracle JvisualVM is a tool that comes with the Oracle Java JDK. **It is an absolutely fantastic tool.** Of course, it is not really a beginner's tool. Launch it like:

```
$ $(JDK_HOME)/bin/jvisualvm
```

It will launch a window, in which all running Java processes will show up. You can select any one of them and connect to it, seeing everything you wanted to know about the runtime characteristics of your Java program.