IntelliJ IDEA Essentials

Develop better software fast with IntelliJ IDEA

Jarosław Krochmalski
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 "Squash'em – The Debugger"
- A synopsis of the book’s content
- More information on IntelliJ IDEA Essentials

About the Author

Jarosław Krochmalski is a passionate software designer and developer who specializes in the financial business domain. He has over 12 years of experience in software development. He is a clean-code and software craftsmanship enthusiast. He is a Certified ScrumMaster and a fan of Agile. His professional interests include new technologies in web application development, design patterns, enterprise architecture, and integration patterns. He likes to experiment with NoSQL and cloud computing.

Jarosław has been working with IDEA since its first release and has observed the IDE grow and mature. He has been designing and developing software professionally since 2000 and has been using Java as his primary programming language since 2002. In the past, he worked for companies such as Kredyt Bank (KBC) and Bank BPS on many large-scale projects such as international money orders, express payments, and collection systems. He currently works as a consultant for the Danish company 7N and writes software for the Nykredit bank. You can reach him via Twitter at @jkroch or by e-mail at jarek@finsys.pl.
I would like to thank my wife, Marylka, and my two boys, Wojtuś and Mati, for being patient and letting me finish the book. Without their help and understanding, this book would not have been possible.

I would like to thank all the people at Packt Publishing, especially Govindan K, Aman Preet Singh, Ellen Bishop, Richard Gall, and Sam Wood—you've made the entire writing and publishing process very smooth and straightforward. A special thanks to all the technical reviewers and proofreaders for providing me with valuable feedback from which I have learned a lot. Thank you.

Greetings to my friends at 7N, Nykredit, Kredyt Bank, and Bank BPS—I hope you enjoy reading the book as much as I enjoyed writing it.
IntelliJ IDEA Essentials

The first version of IntelliJ IDEA was released in January 2001. It is a mature, integrated development environment (IDE), designed to help you in the coding process, and supports a large number of different frameworks, tools, and targets. It works with multiple programming languages. It now includes full support for Java 8 and Java EE 7.

The key objective of IntelliJ IDEA is to increase and assist developer productivity. Whether you develop in Java, Scala, or PHP, or make the frontend using HTML and JavaScript, IntelliJ IDEA's smart and relevant suggestions and code completion, on-the-fly code analysis, and respectable refactoring tools will support you in every step.

When you are migrating from NetBeans or Eclipse, you will quickly see that IntelliJ IDEA is different because it understands the context. The IDE knows where you are in the editor and reacts accordingly; you will be surprised at how smart IntelliJ IDEA behaves.

This tool is a generic workhorse rather than a strict Java IDE. In this book, you will learn how to make IntelliJ IDEA work for you and get your job done in the most efficient and pleasant way.

What This Book Covers

Although the book describes the latest version of IntelliJ IDEA - 14, most of the concepts will also work on the previous revision of the IDE.

Chapter 1, Get to Know Your IDE, Fast, is a very concise note on editions comparison, requirements and installing IntelliJ IDEA in Windows, OSX, and Linux. This chapter guides you through the main workspace and show you ways to customize it for different tasks, presenting briefly the most useful plugins, IDE settings, and configuration tips.

Chapter 2, Off We Go—To the Code, describes the process of setting up a new project or importing an existing one. The chapter explains terminology differences with NetBeans and Eclipse and presents the concept of modules and artifacts.

Chapter 3, The Editor, describes the core of IntelliJ IDEA—the editor. In this chapter, you use state-of-the-art code completion, templates, and other great IntelliJ IDEA features. This chapter shows how to set up the editor and gives you some productivity tips.

Chapter 4, Make It Better—Refactoring, presents the powerful refactoring toolset of IntelliJ IDEA. You are guided through the most useful refactoring techniques.

Chapter 5, Make It Happen—Running Your Project, covers configuring the runtime environment for your project. We also talk about adding run configurations, either on the
server or standalone. This chapter focusses not only on Java, but on other technologies such as Node.js as well.

Chapter 6, Building Your Project, focusses on building a project. You use IntelliJ IDEA's own build system, and Maven and Gradle integration as well.

Chapter 7, Red or Green? Test Your Code, is all about unit testing in IntelliJ IDEA. We focus on setting IntelliJ IDEA up specifically to run tests. You create JUnit and TestNG run configurations and then run and debug the tests. Then, you are given a brief overview of the test runner windows, useful settings, and option suggestions.

Chapter 8, Squash'em – The Debugger, focusses on the IntelliJ IDEA debugger. You get familiar with the debugger tool window and debugger options. We look under the hood—evaluating expressions, using watches, conditional breakpoints, and other debugger features. We also talk briefly about remote debugging.

Chapter 9, Working with Your Team, This chapter is all about version control, and managing change lists and tasks. There is a brief description on how to set up VCS integration, with the main focus on Git. This chapter describes integration with popular bug trackers, such as JIRA and YouTRACK.

Chapter 10, Not Enough? Extend It, describes briefly the plugin architecture of IntelliJ IDEA. We talk about possibilities and develop a simple plugin, so that you have knowledge of how to extend the IDE. You are also presented with useful links and resources to develop your knowledge even further.
Squash'em – The Debugger

Debugging is the process of finding and reducing the number of bugs in your application. In the past, it was a fairly complicated process, sometimes requiring external tools. Today, most programming environments have an integrated debugger, an easy and enjoyable way to find errors in your code. In this chapter, we will focus on the integrated debugger. Out of the box, IntelliJ IDEA supports debugging for Java, Groovy, and JavaScript applications. The debugging functionality is incorporated in IntelliJ IDEA; you only need to configure its settings. Depending on the enabled plugins, the IDE can also support debugging for other languages, for example, Scala or PHP. We will begin by reviewing the debugger settings and options. Setting up the Java and JavaScript debugger will give you a general idea of how to set up the debugger for the language of your choice. Next, we will look at the debugger tool window and then go to the debugging process itself. At the end of the chapter, we will summarize some essential keyboard shortcuts that are valuable for debugging.

After reading this chapter, you will know how to use breakpoints, watches, and how to evaluate expressions. Let's start with the debugger settings.

Debugger settings

At the beginning, you will have to configure the roots, dependencies, and libraries to be passed to the compiler before the debugging process starts. This can be done in the **Project Structure** dialog box. We described the project structure in *Chapter 2, Off We Go – To the Code*. Refer to this chapter to recall how to set up the project structure.

The debugger-related configuration can be found in the **Settings** dialog box, in **Build, Execution, Deployment**. Launch the **Settings** dialog box by picking **Settings** from the **File** menu or just use the Ctrl + Alt + S (PC) or cmd + , (Mac) keyboard shortcut.
The important option from the debugging perspective is to make IDEA generate debugging info for the compiled classes. This option is enabled by default; you can find it by navigating to **Java Compiler | Generate Debugging Info**.

The debugger itself can be configured in the **Debugger** section. As you may remember, you can quickly find a specific option by entering search keywords in the search field of the **Settings** window:

![Debugger Settings Window]

Depending on the number of enabled plugins, you will find language-specific debugger options here. Most of the options are configured properly out of the box to enable convenient debugging. You can tweak them according to your liking. Let's look at the available common options in detail now.

In the root page of the debugger settings, there are some general debugger options. The **Transport** section will define the connection method of the process being debugged. If you are using Windows, the available values for the transport will include **Shared memory**:

![Transport Options]

The **Shared memory** section will be faster, but the **Socket** transport has an advantage—the debugger will use the same universal debugging protocol on the local and remote machines. When you deploy your application to a remote server, the only evident configuration change will be the IP address.
If you happen to work on some legacy code and must use the old JDK, checking the **Force classic VM for JDK 1.3.x and earlier** option might be a good option. This is a rather rare situation, given the fact, that JDK 1.3 was put in an end-of-life state in 2007. However, if you need it, debugging using the classic VM is much faster than debugging with HotSpot under the old JDK.

Disabling JIT will basically pass `Djava.compiler=NONE` at runtime when the application is launched. This will affect the JIT compiler; if checked, the JIT compiler will be disabled.

**Hide debug window on process termination** will make the **Debug** tool window disappear when the debugged application terminates. It's good to have this option checked; the **Debug** window is useless if the debugger is not running, anyway.

If **Focus application on breakpoint** is selected, on hitting a breakpoint IntelliJ IDEA will show the source code containing the breakpoint in the editor. Again, I believe it's good to have it checked. After hitting the breakpoint, it will be easier to comprehend what is going on and evaluate expressions or create watches.

In the **Data Views** section, you can customize the way the data is displayed in the debugger. The options have very intuitive names:

- **Value tooltips delay (ms)**: This will control the tooltips that show the value of the variable when you hover the mouse cursor over the **Variables** tab or, in the editor, when your application is paused on a breakpoint.

- **Sort alphabetically**: This will show the nodes in alphabetical order.

- **Enable auto expressions in Variables view**: This option will force the debugger to analyze the source code near the breakpoint when stopped. It will read one statement before and one statement after the line containing the breakpoint. If there are no method invocations in the surrounding lines, the debugger will try to pick up any expression from these lines and put them in the **Variables** view. We will describe the **Variables** view later in this chapter.

In the **Java** section of **Data Views**, selecting **Autoscroll to new local variables** will make the IDE automatically scroll the list for new variables that appear in scope when stepping. **Auto tooltips for values** will make IntelliJ IDEA show the values of the variables when you hover the mouse cursor over them. The tip will present the value in an alternative (and sometimes more readable) way.

The **Show** subsection describes how the object and its properties will be presented in the debugger. This allows you to toggle between showing and hiding various elements such as static fields or object IDs, for example.
In the following screenshot of the debugger preferences, you can specify which classes should be presented by executing their `toString()` method:

![Debugger Preferences Screenshot]

By default, IntelliJ IDEA will execute `toString()` on all classes that override the standard `toString()` method. You can customize this action by specifying your own list; use the Add class button to open the **Choose Class** dialog box or **Add Pattern** to open the Filter prompt:

![Choose Class Dialog]

When defining the class filter, you can use wildcards such as `*`.

In the **Java Data Type Renderers** section, you can tweak the way different objects are displayed in the debugger. Instead of relying on the object's `String` representation, any expression can be assigned to display the object instead.

To add a new data type renderer, click on **Add**, specify its name, and define the type of objects to be affected by the renderer. This should be the fully qualified name of the class. Click on the ellipsis browse button to display the good old **Find Class** dialog box, where you can choose the desired type from the list. As always, start entering the search keywords to narrow down the list.

The data type renderers are executed twice on rendering a node in the debugger and on expanding the node, when the children information is presented. The dialog box shown in the following screenshot shows the two cases in separate radio groups:
Instead of using the default renderer, you can provide your own expressions in the **Use following expression** field. In the expression fields, you can use an object's properties, constants, and String methods to construct the output.

Normally, expanding a node in the debugger lists the object's member variables by using the renderer to correct their object types. Editing the **When expanding a node** section allows you to overrule that behavior and select a single expression or a series of expressions to render the appearance. The optional **Test if the node can be expanded** field accepts a Boolean expression, that, if true, will make the IDE display the expandable nodes for the defined objects; otherwise, no nodes will be displayed. When editing the expressions, all of the code completion features are at your disposal.
When editing the expression in the data type renderer, use the keyword
**this** to refer to the instance to which the renderer is going to be applied.

In the next section, **Stepping**, we can configure stepping behavior, as shown in the following screenshot. Basically, we choose what the debugger should ignore while stepping. By choosing what to skip, we can improve the debug stepping speed.

The **Skip synthetic methods** will convert stepping into methods generated by the compiler. The **Skip constructors**, **Skip class loaders**, and **Skip simple getters** option names are pretty self-explanatory; they will make the debugger ignore constructors, class loaders, and access methods accordingly.

In the **Do not step into the classes** list, you can add custom classes that should be ignored. The list of classes contains two types of entries: fully qualified class names and class patterns.

You can include or exclude specific packages by modifying the checkboxes selection. By default, the list contains some standard Java SDK and IntelliJ IDEA runtime class patterns, so you can save some of your time by not stepping onto them. Use the checkboxes in the list to disable/enable particular patterns. Use the Add Class, Add Package, and Add Pattern buttons to manage the list.
Stepping can be configured for JavaScript as well.Tick the corresponding checkboxes to force IntelliJ IDEA to ignore the JavaScript library and other specified scripts when stepping, as shown in the following screenshot:

Use the Hotswap page to manage the behavior of the Java HotSwap mechanism. The components of the Hotswap page are explained in more detail in the following points:

- If Make project before reloading classes is turned on and you select Reload Changed Classes form the Run menu, the make process will be performed first.
- If you check the Enable "JVM will hang" warning checkbox and try to perform the HotSwap operation while the application is suspended, IntelliJ IDEA will produce a warning about the possible freezing of the Java Virtual Machine. It's better to have this option checked.
- Select Reload classes in background to reload classes and their process in the background; all progress messages will be displayed in IntelliJ IDEA's status bar.
- The section Reload classes after compilation controls how the HotSwap mechanism should behave. Always will reload classes automatically, Never will basically turn HotSwap off, and Ask will make IntelliJ IDEA prompt you whether to reload the changed classes or not.

Hot swapping is doable only if a method body is altered. If a method or class signature has changed, the class reload will not be possible. There is a commercial plugin named JRebel that allows us to reload almost every change at runtime.

We covered the setting up of the Java debugger. In the next section, we will focus on the JavaScript debugger.
Setting up the JavaScript debugger

Web applications are probably the most popular application type nowadays. If you work on the frontend, JavaScript debugging can come in handy. Of course, the JavaScript Developer Tools included in Chrome or Firefox's Firebug are very, very good. However, having a common IDE with a powerful editor, refactoring tools, and keyboard shortcuts to debug the backend and frontend at the same time will boost your productivity a lot without question. IntelliJ IDEA comes equipped with a fully featured JavaScript debugger. It's bundled as a JetBrains plugin and is enabled by default. If you happen to disable it, refer the Picking your plugins section in Chapter 1, Get to Know Your IDE, Fast, to enable it again.

You can install additional plugins to support JavaScript libraries such as AngularJS to get the code completion and hints in the editor.

The JavaScript debugger in IntelliJ IDEA communicates with the browser; it can be either Chrome or Firefox. Before you start the debugger, you will need to install the extension for these browsers to enable this communication. For Chrome, head to Chrome Web Store and search for the JetBrains IDE Support extension, as shown in the following screenshot, and then install it:

![JetBrains IDE Support](image)

When it comes to Firefox, if you start the debugging session for the first time and have configured Firefox as a browser of your choice in the run/debug configuration profile, IntelliJ IDEA will display a prompt for you to install the add-on for the browser, as shown here:

![Install Extension](image)

Click on OK and then accept the request to install third-party add-ons in the browser. Firefox will restart and communicate with the IDE.
During a debugging session, IntelliJ IDEA will use the port specified in **Built-in server port** to communicate with the Chrome extension or Firefox add-on. It's provided by default, but you can specify another value in the **Debugger** settings as shown here:

![Built-in server port](image)

If the port is already taken, the IDE will find the closest available port and display its value, as shown in the following screenshot:

![Waiting for connection](image)

You can also specify the port value manually. In both cases, you will need to adjust the port number in the browser extension settings; otherwise, the IDE will not be able to communicate with the browser and will report the following error:

![Error](image)

The **Selecting Show DOM** properties window will make the IDE display the DOM properties in the **Variables** pane of the **Debug** tool window if you use Firefox. We will describe the **Variables** tab later in this chapter. If you are using Chrome, the DOM properties will be displayed in the **Elements** tab.

By selecting **Show function values**, you force the IDE to display the values of functions under the **Functions** node in the **Variables** tab. Showing only the user-defined functions will limit the list to your own functions only.

If you want IntelliJ IDEA to show certain object properties in the **Variables** tab, add them to the **Show the following properties for an object node** list. The **Variables** tab will display a label with the values of the listed properties. A good candidate to add to the list is Angular’s $id property, for example.
If you want the IDE to ignore specified scripts or libraries during the debugger stepping, add them to the list using the **Do not step into library scripts** and **Do not step into scripts** checkboxes in the **Debugger** section of the **Settings** dialog box.

Now, if we have the JavaScript debugger set up and are able to communicate with the browser, we can move on. From now on, we will debug in Java, but most of the topics we are going to explain will be the same for all the supported languages: setting breakpoints, adding watches, and evaluating the expressions. Let's place some traps in the code—the breakpoints.

### Managing breakpoints

There are a couple of breakpoint types in IntelliJ IDEA: the line, exception, field, and method breakpoints. Let's start with the most common type: the line breakpoint.

The line breakpoints are placed on the gutter. We described the gutter in the very first chapter. To position the breakpoint, simply click on the gutter where you want the debugger to stop. You can also use the $\text{Ctrl} + F8$ (PC) or $\text{cmd} + F8$ (Mac) keyboard shortcut. The selected line will be shaded in red and the big red dot on the gutter will represent the breakpoint as shown here:

```java
public void readData(JsonObject user) {
  try {
    currentUser = objectReader.readValue(user.encode());
  } catch (IOException e) {
    logger.error("Error reading user data", e);
  }
}
```

Line breakpoints can be set on executable lines only. Comments, declarations, and blank lines are not suitable locations for line breakpoints.

Another type of breakpoint is the method breakpoint; it lets you follow the program flow at the method level. To set the breakpoint on the method, just click on the gutter near the method signature as shown in the following screenshot:
Be warned, however, that the method breakpoint will slow down the debugging process a lot. IntelliJ IDEA will warn you about this as shown here:

![Method breakpoints may dramatically slow down debugging](image)

The reason for this slowdown is that the Java runtime has to add checks on every method entry to check whether it matches the breakpointed method signature. Also, the method inlining (an optimization performed by the Java Just-In-Time (JIT) compiler) becomes impossible; small methods will run 10 to 100 times more slowly.

If you notice that the debugging session is very slow, you may have some method breakpoints set up by accident. You can review, disable, or delete active breakpoints in the Breakpoints dialog box. We will discuss this dialog box in a minute.

If you would like the debugger to stop at a method, consider placing the ordinary line breakpoint on the first line of the method instead of using the method breakpoint.

![To move a breakpoint, just drag a line breakpoint to the needed line. The field/method breakpoint can be dragged as well, but to another field/method declaration only.](image)

To delete the breakpoint, simply click on it or use the Ctrl + F8 (PC) or cmd + F8 (Mac) keyboard shortcut again.
To change the breakpoint state from active to disabled, put the caret on the line with the breakpoint and select **Toggle Breakpoint Enabled** from the **Run** menu, as shown here:

Alternatively, you can right-click on the breakpoint with your mouse, uncheck the first checkbox in the pop-up dialog box, and click on **Done** as shown here:

The breakpoint icon on the gutter will turn green. This means that the breakpoint is now disabled. The disabled breakpoint will no longer stop the debugger, as can be observed from the following screenshot:

You can toggle the breakpoint state between enabled and disabled by clicking on it using the mouse with the **Alt** key pressed.
Sometimes you would like to stop once and investigate the problem without being bothered with the active breakpoint set up in the code. There is a nice feature in IntelliJ IDEA named temporary breakpoints. A temporary breakpoint will be removed when hit. To place a temporary breakpoint, use **Toggle Temporary Line Breakpoint** from the **Run** menu. Alternatively, you can place an ordinary line breakpoint, then click on it with the right mouse button, select **Remove once hit**, and then click on **Done**:

![Temporary Breakpoint](image)

The temporary line breakpoint will be represented with a red icon containing the number 1 inside as shown here:

![Temporary Breakpoint Icon](image)

If you want the debugger to stop if a specific instance variable field is being accessed or modified, use **Field Watchpoint**. This is a special kind of breakpoint that will pause the execution if any access or modification to the instance variable is being made.

To create **Field Watchpoint**, place the caret in the line with the field definition and place the breakpoint in the usual way, either by clicking on the gutter or by using the Ctrl + F8 (PC) or cmd + F8 (Mac) keyboard shortcut. The Field Watchpoint will be represented with a slightly different gutter icon as shown here:

![Field Watchpoint](image)

You can enable or disable a Field Watchpoint the same way as the other breakpoints, by clicking on it using the mouse with the **Alt** key pressed. Disabled Field Watchpoints will also turn green as shown here:

![Disabled Field Watchpoint](image)
The line/method breakpoints and Field Watchpoints can be set from the editor. If you want to tweak their behavior or define breakpoints of different types, you will need to open the Breakpoints dialog box. To do this, use View Breakpoints from the Run menu or use the Ctrl + Shift + F8 (PC) or cmd + Shift + F8 (Mac) keyboard shortcut. The output is shown in the following screenshot:

You can also access the Breakpoints dialog box by right-clicking on the breakpoint and selecting More. The keyboard shortcut for this (shown in the following screenshot) is the same as the global View Breakpoints command; that is, Ctrl + Shift + F8 (PC) or cmd + Shift + F8 (Mac):
Chapter 8

On the left side of the **Breakpoints** dialog box, you can see all breakpoints currently defined in the project. For each individual breakpoint in the list, you can view and change its properties as required. Do this by selecting the breakpoint and adjusting its options on the right side of the dialog box.

To navigate to the breakpoints source code from the Breakpoints dialog box, double-click on the desired breakpoint, or press the **F4** keyboard shortcut and close the window using **Esc**.

For every defined breakpoint, you can configure the suspend policy, as shown in the following screenshot. It defines whether the application should be suspended on hitting the breakpoint:

![Suspend Options](image)

If you select **All**, all threads will be suspended when a breakpoint is hit. On the other hand, if **Thread** is selected, only the thread where the breakpoint is hit will be suspended. If the **Suspend** checkbox is not selected, no threads will be suspended.

The actions that can be performed on hitting the breakpoint include logging the message or logging the evaluated expression to the console. To log the message, mark the corresponding checkboxes. Even if the expression field is just a single line, it's equipped with all the benefits of IntelliJ IDEA's editor such as hints and parameter completion, as shown in the following screenshot:

![Logging Options](image)
By switching off suspension and logging the message or the expression to the console, the debugger can provide you with some useful debugging information, without even stopping at the breakpoint. This is particularly useful for remote debugging situations, where you may not change the source code to add log statements to the code.

At any time, you can hover the mouse pointer in the editor over the defined breakpoint to quickly get the information about the suspend policy and actions to be executed when the breakpoint is hit, as shown in the following screenshot:

In the **Condition** field, you can enter a Java Boolean expression that should be valid in the line where the breakpoint is set. Again, the expression field contains a fully featured, syntax-aware editor. The expression is evaluated every time the breakpoint is reached. The expression can include calls to the methods that return Boolean values. If the evaluation result of the expression is **true**, the actions you selected will be performed. Otherwise, if the result is **false**, this breakpoint will not produce any effect and the debugger will just skip it.

Apart from providing the **Condition** expression, another way to create a conditional breakpoint is to select other breakpoints from the **Disabled until selected breakpoint is hit** drop-down list as shown here:
Picking any breakpoint from the list will disable the breakpoint conditionally. The breakpoint will become enabled when any other one is hit. If the breakpoint is hit, it can be disabled conditionally again or left enabled, as shown here:

![Conditional Breakpoint Icon]

The breakpoint defined this way will be presented with the conditional icon in the gutter as shown here:

```java
@override
public Auditor getCurrentAuditor() {
    return Auditor.getSystemAuditor();
}
```

You can limit breakpoint hits only with particular object instances using their IDs. The instance ID is the Java object ID that uniquely identifies an object in the target JVM and is reported by the JVM itself. IntelliJ IDEA uses it to display variables with the `@` character. For `foo.bar.MyClass@418`, the instance ID will be `418`, for example. By turning on **Instance filters** and providing ID values, you can make this breakpoint active only for specific object instances.

To filter breakpoint behavior with regard to a particular class, select the **Class filter** checkbox. Use the **Class Filters** dialog box (shown in the following screenshot) to configure class filters that determine which classes a specific breakpoint will be triggered in and in which classes it should not:

![Class Filters Dialog]

You specify classes and class patterns to be included on the left pane, and classes and class patterns to exclude on the right pane. To add a class to either of the lists, press the Add class button to open the **Choose Class** dialog box.
The filter specified through a class name points at the class itself as well as at all its subclasses.

Alternatively, to add a class pattern use the Add Pattern button. IntelliJ IDEA will show the prompt for the class pattern. The pattern may start or end with an asterisk (*), which stands for any number (including zero) of characters. A filter specified through a class pattern points at the classes whose fully qualified names match the pattern. The subclasses are selected only if their fully qualified names also match the pattern.

The class patterns are matched against fully qualified class names.

The last option, Pass count, especially useful when debugging long loops, allows you to specify the counter. On each breakpoint hit, the counter will be decreased; after the specified number of passes, the execution will be suspended. In other words, this defines the number of times a breakpoint is reached but ignored. While this may be occasionally useful, it can be emulated by using the breakpoint condition expression, which is faster to use than digging through a lot of nested dialog boxes. The Pass count option is available only if Instance Filters and Class Filters are not marked as active.

All the breakpoints in the Breakpoints dialog box (shown in the following screenshot) are grouped by their type. You can group them additionally using switches in the upper toolbar. Grouping by package, class, or file is possible. To use specific grouping, just left-click on the icon.
Yet another type of breakpoint you can define is **Java Exception Breakpoint**. The debugger will stop if the exception of a specified type is thrown. This kind of breakpoint is not related to a specific source code line, but applies globally to the runtime environment. To add an exception breakpoint, click on the green plus icon in the upper toolbar, or as usual use the Alt + Insert keyboard shortcut to display a list of breakpoint types you can create, as shown in the following screenshot:

Select **Java Exception Breakpoints** from the list or use the Alt + 3 keyboard shortcut. The well-known **Select Exception Class** dialog box will pop up, allowing you to specify the exception class from the project classpath or project. As usual, start typing to narrow down the list, as shown in the following screenshot:
To define a **Field watchpoint** using the **Breakpoints** dialog box, again use the $Alt + Insert$ keyboard shortcut and select **Java Field Watchpoints** from the list this time. In the next dialog box, type the fully qualified name of the class containing the desired instance variable. Alternatively, you can use $Shift + Enter$ to search for the class using the well-known class finder dialog box. When the name of the class is filled, type the name of the instance variable or press $Shift + Enter$ again to display the list of instance variables to choose from, as shown in the following screenshot:

![Field watchpoint dialog box](image)

If you click on **OK**, by default the field will be monitored for modification only. You can change this behavior and make IntelliJ IDEA watch for access fields as well; just select the appropriate checkbox as shown here:

![Watch checkbox](image)

Field watchpoints can be created from the **Debug** tool window as well. We will discuss this process later in this chapter.
The list of breakpoints is also visible in the Favorites tool window. If you want the list to be visible all the time, just pin the tool window, as we described in the first chapter. The icons in the tool window are the same as in the gutter to easily distinguish the breakpoint's type. To remove the breakpoint or field watchpoint, select it from the list and click on the minus icon; or use Delete from the context menu.

In the editor, locate the line with the watchpoint or breakpoint to be deleted and click on its icon in the gutter. You can also remove the breakpoint using the Breakpoints dialog box. Additionally, you can remove all breakpoints of a certain type; just select the whole group and execute the Delete action.

Now that we have our breakpoints explained and set, let's see how they work in real life. Let's start the debugger session.

**Starting the debugger**

To begin the debugging session, you will need the runtime/debug configuration defined for the project. We described this process in detail in Chapter 5, Make It Happen – Running Your Project. This time, though, instead of running the defined configuration, use the Shift + F9 (PC) or Ctrl + D (Mac) keyboard shortcut; or click on the Debug icon on the toolbar, as shown here:

When you start the debug configuration, two things will happen. First, IntelliJ IDEA will analyze the breakpoints you set up and, if they are valid, will mark them with the valid breakpoint icon, as shown here:
Otherwise, if the breakpoint is invalid, the IDE will mark it with the invalid breakpoint icon and thus ignore it. The breakpoint can be invalid because it is placed on the line with the comment or in a block with an opening or closing brace, for example, as shown here:

```
}  
System.out.println(1);   
System.out.println(a);
```

Hover over the invalid breakpoint to see the reason why IntelliJ IDEA decided not to take it into account:

The validity of the breakpoints is presented only when the debugger session is started and active.

The second thing that will happen is that the **Debug** tool window becomes available; let’s focus on it as shown in the following screenshot:
Chapter 8

The Debug tool window

If you are debugging multiple applications at once by executing multiple debug configurations, each one will display its output in a separate tab. The tab will be named after the corresponding debug configuration is executed.

By default, the Debug tool window will switch itself to the Console tab and display the output generated by your application. If the debugger stops, the most interesting stuff can be performed on the first tab: the Debugger tab. Let’s take a closer look at it now.

You will find three nested tabs here that present the current state of the suspended application: Frames, Variables, and Watches as shown here:

You can hide unnecessary tabs by using the hide icon in the tab itself as shown here:

The hidden tab can then be restored using its icon in the Hide/Restore toolbar, available in the upper-right corner of the Debug tool window. You can toggle the additional Threads tab as well, as shown in the following screenshot:
Let’s discuss the tabs in detail. The **Frames** and **Threads** tabs are very similar. They both give you access to the list of frames and threads in your application. A frame corresponds to an active method or function call. A frame stores the local variables of the method or function called, its arguments, and the code context that enables evaluation of the expressions.

In the **Frames** pane, you select the thread you are interested in from the drop-down list as shown here:

To navigate between frames, use the up and down arrow buttons on the toolbar or the up arrow and down arrow keys on the keyboard.

The **Threads** tab, for a change, shows all the threads of a process as a tree view, presenting the name, ID, and thread status, as shown in the following screenshot:
If you expand a single thread branch and click on it (in the Frames tab) or double-click (in the Threads tab) on the method name, the corresponding source code of the method will be opened in the editor. At any time, the thread view can be customized a little by right-clicking anywhere in the Frames or Threads tab and selecting Customize Threads View from the context menu, as shown here:

In Customize Threads View you can turn on some additional information to be shown, such as displaying thread groups or source file names as shown here:

The currently suspended thread is marked with a red tick icon in the call stack. You can interrupt it by selecting Interrupt from the context menu. If you select the method currently being executed (and paused at the breakpoint), you can also drop a frame from the call stack using the context menu, as shown here:
Squash'em – The Debugger

By using the **Drop Frame** functionality, you can "fall back" on the previous stack frame, in a way going back in time. You need to be aware that changes that were already made to the global state (such as static variables) and variables outside the stack frame will not be reverted. Only local variables will be reset. Dropping the frame is a useful feature to explore different paths of execution, without having to restart the application, or a particular lengthy process that led to the current stack.

Dropping a frame is very helpful to re-enter a method if you missed a critical spot.

During the debugging session, the most used functionality is inspecting variables and evaluating expressions. Let's cover that process now in more detail.

**Inspecting variables and evaluating expressions**

During the debugging session, IntelliJ IDEA will present the values of the variables in the editor itself, next to the variable usage, as shown in the following screenshot:

![Screenshot of variables](image)

While this may be useful as a quick overview for the current state, the **Variables** tab gives you the opportunity to examine the values of the variables in your application in more detail. When a stack frame is selected in the **Frames** tab, the **Variables** tab displays all the variables within its scope, such as method parameters, and local and instance variables. Variables here are listed with their unique internal IDs we mentioned earlier; you can use this ID to define the breakpoint instance filter. Each variable in the tab has a context menu when you right-click on it as shown in the following screenshot:
The **Set Value** option allows changing the variable value. **Inspect**, available for fields, local variables, and reference expressions, will display and track its reference in its own window:

If you need to examine several references in detail, you can open an inspection window for each one of them. A separate window is created for each reference and all of its child references. All changes of the references are immediately reflected in the inspection window.
The inspection window is not modal and you can open as many as you want.

Most of the options in this context menu are self-explanatory. You can copy a variable's value to the clipboard, compare the value with the clipboard, or set the variable value. If you want to label the selected variable with a meaningful name, use the Mark Object command.

Pick Customize Data Views from the context menu as shown here:

You will be presented with a dialog box containing two tabs, Data Views and Data Type Renderers, as shown in the following screenshot:

The contents of the two tabs will be the same as we described in the Debugger settings section at the beginning of this chapter.

If you want to evaluate a number of variables or expressions in the context of the current frame and view all of them simultaneously, you can create watches for them. The values of the expressions are updated with each step through the application, but are only visible when the application is suspended.

By using the Add to Watches command, you can send the selected object to the Watches tab. You can achieve the same effect by right-clicking on the variable in the editor and selecting Add to Watches from the context menu, as shown in the following screenshot:
There is one important difference, though: the Variables tab is active only when the debugger session is running. On the other hand, the context menu in the editor is active whether the debugger session is active or not.

All the variables you added to watches will show up in the Watches tab, as shown in the following screenshot:

To add an item to the Watches pane, you can also click on the green plus icon in the toolbar; or, as always, use the Alt + Insert keyboard shortcut. To change the expression represented by a watch, right-click on the desired watch and select Edit in the context menu.

The expression field is, again, a fully featured advanced source code editor with code completion and hints available, as shown here:
To remove a watch, select it in the Watches pane and choose Remove Watch from the context menu or use the Delete key.

The Watches tab context menu has very similar functionality to the one present in the Variables tab. Additionally, you can use it to edit the watch, remove the watch, or remove all watches, as shown here:

Watches are persisted as part of a project. They will be preserved if you close the project.

Apart from creating watches, you can also define field watchpoints in the Variables tab. Just select the variable and choose Add Field Watchpoint to have a breakpoint defined as shown in the following screenshot:
Probably the most often used feature of the debugger is the expression evaluation. Apart from having values of the variables presented in the Variables or Watches tab, you will often evaluate expressions on-the-fly. The IDE enables you to evaluate an arbitrary expression from the context of the stack frame currently selected in the Frames tab. The following modes are available: Expression Mode to evaluate single-line expressions and Code Fragment Mode to evaluate short code blocks, including declarations, assignments, if/else constructs, and loops.

IntelliJ IDEA provides a way to quickly evaluate an expression at the caret or a selection. If the debugger stops on the breakpoint, select the variable or expression you want to evaluate and start the Evaluate Expression dialog box (shown in the following screenshot) by choosing the Evaluate Expression command from the Run menu or using the Alt + F8 (PC) or option + F8 (Mac) keyboard shortcut:

Expression mode is active by default; if you want to evaluate a code fragment, click on the Code Fragment Mode button, the result of which is shown in the following screenshot:
As you will quickly find out, the expression field comes with all the benefits of the IntelliJ IDEA source code editor. You will find syntax highlighting, code completion, hints, and every powerful feature the IDE has to offer in terms of editing code. You can even use live templates here, as we described in Chapter 3, The Editor.

A method can be invoked within the Expression Evaluation dialog box only if the debugger has stopped at a breakpoint, but has not been paused manually.

Take note that, if a method invoked within Expression Evaluation has a breakpoint inside its body, this breakpoint will be ignored.

If the specified expression cannot be evaluated, IntelliJ IDEA will describe the reason in a few words in the Result pane of the dialog box as shown here:

If you are using the mouse during the development process, you can quickly evaluate the value of any expression by placing the mouse cursor over the expression during the debugger session. The value of the expression will be shown as a tooltip, as shown in the following screenshot:

If an expression contains children, clicking on the green plus icon will expand the hint in the object tree dialog box and display all of the children.
Clicking on the variable in the editor with the left \textit{Alt} key pressed will show the dialog box with the expression value evaluated.

Apart from the Hide/Restore toolbar, the \textbf{Debug} tool window has two toolbars of great importance: the \textbf{Debug} toolbar and the \textbf{Stepping} toolbar. We will explore them now.

\section*{Debugger actions}

All of the actions available in the toolbars mimic the actions present in the \textbf{Run} menu during the debugging session. I believe that using the toolbars is more convenient than using the menu (apart from using keyboard shortcuts, of course). Let's take a look at them now.

The Debug toolbar contains actions to manage your debugging session. You can restart the debugging session by using the \textit{Ctrl} + \textit{F5} (PC) or \textit{option} + \textit{F5} (Mac) keyboard shortcut. If the session is stopped, this command will turn in to the Debug button, which will start the debugging session again, as shown here:

When an application is paused, you can click on the Resume Program button, or use the \textit{F9} (PC) or \textit{cmd} + \textit{Option} + \textit{R} (Mac) keyboard shortcut, to resume the program execution, as shown here:
Squash’em – The Debugger

Click on the Pause button to suspend the execution and the Stop button to stop the debugging session, as shown here:

![Pause Program](image)

There is a useful button you can use to temporarily disable all the defined breakpoints: the Mute Breakpoints button (shown in the following screenshot). Use this to change the status of the breakpoints from enabled to disabled and vice versa. With breakpoints muted, the program will execute without stopping.

![Mute Breakpoints](image)

The Get thread dump button will allow you to review the thread dump in a convenient and readable way. The tab is divided into two parts: the left one displays all the threads and the right one displays the stack trace for the selected thread, as shown here:

![Get thread dump](image)

The stepping toolbar contains commands to navigate the execution flow during the debugging session. The first one is Show Execution Point, available with the `Alt + F10` (PC) or `option + F10` (Mac) keyboard shortcut, as shown in the following screenshot:

![Stepping toolbar](image)
When you go away from the point where the debugger stopped, either by switching the editor's windows or navigating to other files or methods, Show Execution Point will quickly get you back to the point of interest, which is the current execution point where you can continue stepping.

The Step Over command (the F8 keyboard shortcut) will make the debugger run until the next line in the current method or file. If the current line is the last one in the method, execution shifts to the line executed right after this method.

The Step Into command (the F7 keyboard shortcut) will make the debugger step inside the method called at the current execution point. If the method is set to be skipped in the Stepping page in the Debugger settings dialog box, as we described at the beginning of this chapter, you can use the Force Step Into command or the Alt + Shift + F7 (PC) or option + Shift + F7 (Mac) keyboard shortcut. The Force Step Into command permits you to dig into a class from the list of classes not to be stepped into; for example, a standard Java SDK class. Clicking on the Step Out command (the Shift + F8 keyboard shortcut), will make the debugger step out of the current method, to the line executed right after the execution of this method. One feature I find especially useful is Run to Cursor, shown in the following screenshot:

This action will resume program execution and pause until the flow reaches the line at the cursor position in the editor. You don't need to define the breakpoint for this. It's like a temporary breakpoint defined implicitly. Be aware that, if the cursor is positioned on the line that has already been executed, the execution flow will just be resumed, and you will have no chance to go back.

When you have stepped too deep into the method's sequence and need to step out of several methods at once, use the Run to Cursor feature.
Now you know how to navigate within the debugger. Let’s make it even faster and more effective by summarizing keyboard shortcuts worth using while debugging.

**Keyboard shortcuts summary**

The following table summarizes debugger actions and associated shortcuts:

<table>
<thead>
<tr>
<th>Action</th>
<th>PC shortcut</th>
<th>Mac shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug</td>
<td>Shift + F9</td>
<td>control + D</td>
</tr>
<tr>
<td>Step over</td>
<td>F8</td>
<td>F8</td>
</tr>
<tr>
<td>Force step over</td>
<td>Alt + Shift + F8</td>
<td>option + Shift + F8</td>
</tr>
<tr>
<td>Step into</td>
<td>F7</td>
<td>F7</td>
</tr>
<tr>
<td>Force step into</td>
<td>Alt + Shift + F7</td>
<td>option + Shift + F7</td>
</tr>
<tr>
<td>Run to cursor</td>
<td>Alt + F7</td>
<td>option + F9</td>
</tr>
<tr>
<td>Evaluate expression</td>
<td>Alt + F8</td>
<td>option + F8</td>
</tr>
<tr>
<td>Resume program</td>
<td>F9</td>
<td>cmd + option + R</td>
</tr>
<tr>
<td>Toggle breakpoint</td>
<td>Ctrl + F8</td>
<td>cmd + F8</td>
</tr>
<tr>
<td>View breakpoints</td>
<td>Ctrl + Shift + F8</td>
<td>cmd + Shift + F8</td>
</tr>
<tr>
<td>Show execution point</td>
<td>Alt + F10</td>
<td>option + F10</td>
</tr>
</tbody>
</table>

**Summary**

As you can see, debugging in IntelliJ IDEA is very handy. After reading this chapter, you know how to set up the debugger, place and manage the breakpoints, and look under the hood (that is, inspecting variables, adding and tracking watches, and evaluating expressions). Searching for bugs will be easy and effective.

You now know how to set up a project and unleash the power of the state-of-the-art code editor. You can define a runtime or debug configuration to execute the application. Well, this should be enough to get you going. However, there’s some more—version control. You can probably work on your software in the team and use a version control system, such as SVN or GIT. Even if you work alone, having version control set up is a great idea. IntelliJ IDEA provides first-class support for a version control system such as SVN or GIT. We are going to cover this subject in the next chapter.
Where to buy this book

You can buy IntelliJ IDEA Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.