Getting started with Intellij IDEA

Exploit IntelliJ IDEA's unique features to rapidly develop web and Java Enterprise applications

Hudson Orsine Assumpção

[PACKT]
PUBLISHING

Getting Started with IntelliJ IDEA

Exploit IntelliJ IDEA's unique features to rapidly develop web and Java Enterprise applications

Hudson Orsine Assumpção

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Getting Started with IntelliJ IDEA

Credits

Author
  Hudson Orsine Assumpção

Reviewers
  Scott Battaglia

  Tomasz Nurkiewicz

Acquisition Editors
  Edward Gordon

  Rubal Kaur

Lead Technical Editor
  Mohammed Fahad

Technical Editors
  Adrian Raposo

  Gaurav Thingalaya

Copy Editor
  Tanvi Gaitonde

Project Coordinator
  Romal Karani

Proofreader
  Amy Johnson

Indexer
  Rekha Nair

Production Coordinator
  Kirtee Shingan

Cover Work
  Kirtee Shingan

About the Author

Hudson Orsine Assumpção is a Brazilian software engineer with a bachelor's degree in Information Systems from Universidade Estadual de Montes Claros (Unimontes). He has experience in developing applications in Java EE and ActionScript 3. He also has practical knowledge in Data Warehouse, OLAP tools, and BPMN. He is a certified ITIL V3 foundation professional. Currently, he is developing, with a friend, a web-based ERP system using the Java EE platform.

About the Reviewers

Scott Battaglia is a senior Software Development Engineer for Audible.com, the leading provider of premium digital spoken audio information, currently focused on Android development. Prior to that, he was an Identity Management Architect and senior Application Developer with Rutgers, The State University of New Jersey. He actively contributes to various open source projects, including Apereo Central Authentication Service and Inspektr, and has previously contributed to Spring Security, Apereo OpenRegistry, and Apereo uPortal. He has spoken at various conferences, including Jasig, EDUCAUSE, and Spring Forward, on topics such as CAS, Identity Management, Spring Security, and software development practices. Scott holds a Bachelor of Science and Master of Science degree in Computer Science from Rutgers University and a joint Master of Business Administration and Master of Public Health—Health Systems and Policy—degree from Rutgers University and the University of Medicine and Dentistry—School of Public Health. Scott is on the advisory board for Blanco's Kids, a group focused on providing public health services to extremely poor communities in the Dominican Republic. In his spare time, he enjoys photography, running marathons, learning new programming languages, and sponsoring random Kickstarter projects.

Tomasz Nurkiewicz is a Software Engineer with 7 years of experience, mostly developing backend with JVM languages. He is a Scala enthusiast. Presently, he is working on leveraging the power of functional programming in the global banking industry. Tomasz strongly believes in automated testing at every level. He claims that a functionality not tested automatically is not trustworthy and will eventually break.

He is always happy to implement monitoring and data visualization solutions and has reviewed the book *Learning Highcharts, Packt Publishing* in the past. He is a data analysis apprentice and a technical blogger (`http://nurkiewicz.blogspot.com`), speaker at conferences, and trainer. Most importantly, he has been a proud user of IntelliJ IDEA for many years.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Developing codes that are correct in a fast manner is a difficult task, even for senior developers. In conjunction with the abilities of the developer, great tools are involved in providing a simple, flexible, and fast way of developing software. IntelliJ IDEA is one of the most powerful IDEs for Java developers, extending facilities you can find in common IDEs and providing features you've probably never seen before.

In this this book, you will learn how to extract the maximum from IntelliJ IDEA 12, beginning with understanding the basic concepts and applying features in real development challenges. You will see your development speed improve naturally and understand why IntelliJ IDEA is considered to be the smartest IDE in the world.

What this book covers

Chapter 1, *Getting Started with IntelliJ IDEA 12*, presents some features in a superficial way. The intention here, is to show you the features you can expect in IntelliJ IDEA. The chapter begins by showing common features, then it shows you what is new in this version of the IDE. You will also see the differences between the Community Edition and Ultimate versions, and how to install and configure this software. At the end of the chapter is a section where you will see what you can expect in the future versions of IntelliJ IDEA.

Chapter 2, *Improving Your Development Speed*, will provide a wide range of functionalities that will improve your development speed. In the beginning, it presents to you the main visual interface of the IDE. After this, it shows you how to improve your productivity using the features of IntelliJ IDEA such as the productivity guide, live templates, and navigation usabilities. More development speed techniques are discussed later in the chapter, such as organizing tasks using your favorite issue tracker and how to use TODO marks. At the end of the chapter, a small list of diverse plugins is presented.

Chapter 3, *Working with Databases*, explores diverse database facilities. Different from the previous two, this chapter uses a practical approach to show the features of IntelliJ IDEA that could make your work with databases even simpler. This chapter begins by showing the Database Tool; here, we will create a simple database using this tool. We will use visual elements to create, edit, and populate a table, and visualize data in a table and filter it. In the end, we will understand how to work with ORM technologies, such as Hibernate, extracting entities from the database we created.

Chapter 4, *Web Development*, continues using a practical approach to develop a simple web application. In this chapter, some features that are available in all kinds of projects are used, such as refactoring techniques. We start this chapter by creating a web module and configuring the application server. After this is done, we configure the web module to use the frameworks we need such as JSF and Spring. Then, we correct the dependencies of the module using IntelliJ IDEA facilities and develop our code. At the end of the chapter, we develop a simple web service that will be used in the next chapter.

Chapter 5, *Desktop Development*, will show you how to develop a simple desktop application. In the beginning, we will create a desktop application module and discover the visual editor. While we explore this tool, we will create the GUI of our application and see how the visual editor is integrated with the whole IDE. Then, we will consume the web service we created in the previous chapter and, finally, configure the data binding of our application.

What you need for this book

In some chapters in this book, we will use some external software that needs to be installed in your computer; these are as follows:

- MySQL version 5.5
- Apache Tomcat 7
- We also need Java Development Kit (JDK) 7 or later.

Who this book is for

If you are a developer, experienced or not, and want to improve your development speed or just understand what IntelliJ IDEA can offer you, this book is directed to you. You just need basic background knowledge of Java development to follow this book; however, some technologies used in this book can be advanced and will not be explained in depth.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Mount the downloaded `.dmg` file as another disk."

A block of code is set as follows:

```
private void showMessage(Status status) {
  //TODO: something should be done here.
}
```

Any command-line input or output is written as follows:

```
mysql -u root -p
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: " In this initial window, you can open the **Settings** window".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

[4]

1
Getting Started with IntelliJ IDEA 12

This introductory chapter will present some features of IntelliJ IDEA (later referred to simply as IntelliJ) and how you can install and configure it. You will be presented with some of the great features of IntelliJ IDEA and will notice for yourself that, even if you've already seen some of the features presented here in other IDEs, none of them are so focused to provide facilities to the work of the developer.

After you've seen the features, you will learn how to install and configure IntelliJ. Despite these being really simple tasks, you will probably see great value in this information, mainly because the organization and nomenclature of some elements may differ from that of the IDE you used before. So, what is provided here is to help you remove some barriers you may encounter while using IntelliJ.

Presenting features

As the name suggests, IntelliJ IDEA tries to be an intelligent **Integrated Development Environment** (**IDE**). To do this, it applies techniques that permit the automation of some tasks and suggests actions that may facilitate the developer's work. Allied with this, IntelliJ provides out-of-the-box integration with popular frameworks used by Java developers. This native integration drastically reduces the existence of bugs and provides great synergy between the different technologies used to construct your project.

Here, I will show you some of the great features that exist in IntelliJ IDEA 12.

Smart code completion

Code completion is a set of techniques used to analyze and propose pieces of code to the developer. Like other IDE, IntelliJ also presents code completion; but, as compared to other techniques, code completion in IntelliJ is much smarter.

- It can detect and differentiate between a big quantity of languages, such as Java, Scala (using a plugin), Groovy, SQL, JPQL, JavaScript, and so on, even when they are injected in another sentence as shown in the following screenshot (for example, a SQL string in Java code):
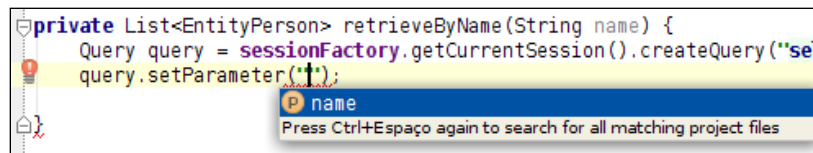
- Beyond suggested variables, classes, and method names, it can suggest methods, parameters, properties, filenames, resources, Spring Framework's beans, database tables, and so on, even when you are working with non-Java files.

- Smart code completion suggests sentences based on the context and the user needs. So it can suggest, for example, chain completion like `getModule().getProject()` that fit correctly in the context as shown in the following screenshot:
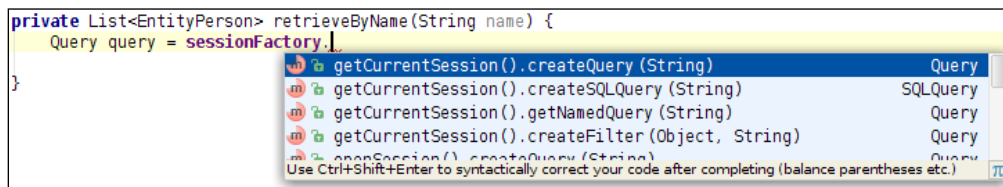
- IntelliJ can propose completion based on middle name search and camel case abbreviation as shown in the following screenshot:
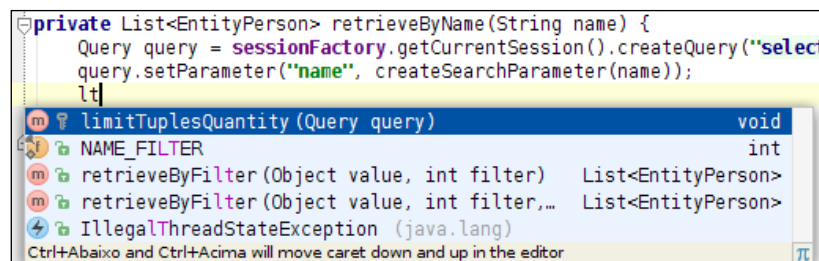
On-the-fly code analysis

An IDE isn't useful if it can't identify whether or not an expression is invalid and will bring compilation errors to the developer. Instead of only identifying these problems, IntelliJ tries to help you improve your code. While typing your source code, you will notice that IntelliJ will analyze your code in the background. In these analyses, IntelliJ will use its configurations to identify errors and possible improvements.

The kind of analyses that IntelliJ does in the code are fully configurable; these configurations are called inspections and are one of the most powerful features of this IDE. With inspections, IntelliJ can find a large range of existing errors, possible bugs, or improvements, such as identifying if a variable is never used, suggesting a simplified sentence, identifying thread problems, and more, as you can see in the following screenshot:
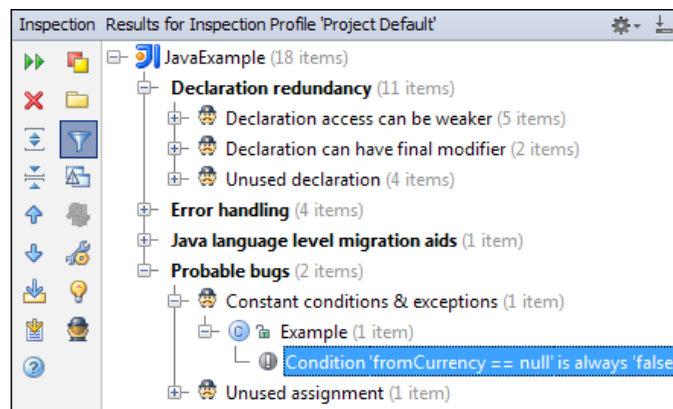
Advanced refactoring

Sometimes, the refactoring process is very tiring because you need to make a lot of modifications to a lot of files. Even when you do the refactoring carefully, sometimes compile errors appear, for example, because you forgot to change a method usage. Fortunately, IntelliJ can help us do these tasks.

IntelliJ has a big range of refactoring options that help the developer refactor his/her code faster and in a more secure manner. Depending on the section of code you are working on, IntelliJ will automatically suggest the refactoring. In case it doesn't propose any refactoring or gives you an option you don't want to use, you can select an option from the refactor menu. The following are some of the refactoring options available:

- **Type migration**: With this, the refactoring technique can be useful in changing a member type in a safe way because IntelliJ will find and modify all the usages of the member. You can use this refactoring in a bunch of situations, such as in method return type, local variable, and parameter.

- **Extract class**: This refactoring option enables the developer to extract one class from another. The developer can choose which methods and variables he/she wants to transfer to the new class. This is very useful when you have a very huge and complex class and want to simplify it. There is a similar option that creates a superclass based in the current class being edited and another that extracts an interface in the same way.
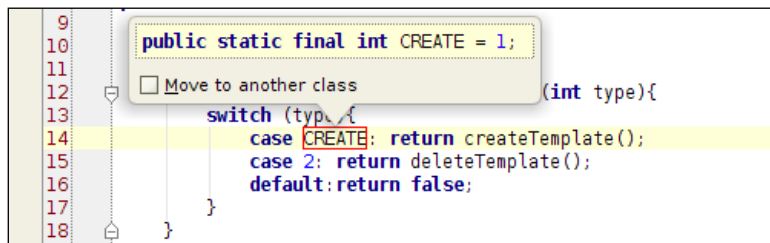
- **XML-aware dedicated refactoring**: There are more than 50 refactoring options exclusive to XML files. For example, you can rename tags and attributes, wrap and unwrap tags, replace attributes with tags and vice versa, convert tags to attributes, safely add and remove subtags and attributes, and so on. If you use some framework that works with XML files, such as Spring, you will see that you can, for example, refactor a bean parameter's name directly in XML.

- **Drag-and-drop items**: Sometimes you look at your project tree and think that it would be better to move some classes and packages. In IntelliJ, you can drag-and-drop the elements from one place to another in your project, and it will automatically change the imports and usages for you.

- **Change method signature**: You have probably already experienced a situation like this: you created a method and, after doing a lot of work, you've decided to change the method signature. Using IntelliJ, it is possible to minimize the effects of a method signature change. IntelliJ can remove, reorder, and add parameters to the signature; it is also possible to change the return type and the parameter types. In case you are adding a parameter, IntelliJ will insert a default value in all the usages of the method.

- **Introduce constant**: Sometimes, while developing, we don't see that it is possible to use a constant instead of repeating the same value all throughout the class. You can easily create a constant based on the repeated values present in your code. Once the constant is created, IntelliJ will detect the entries and replace them as shown in the following screenshot:

```
 9
10     public static final int CREATE = 1;
11
12   □ Move to another class                (int type){
13         switch (type){
14             case CREATE: return createTemplate();
15             case 2: return deleteTemplate();
16             default:return false;
17         }
18   }
```

Navigation and search

One of the great characteristics you will find in IntelliJ is the ease of finding and navigating to a resource. You can run a search in practically all the controls that exist in the IDE, even when you can't see a search field. You can also use different elements to navigate to a specific code that you want.

The following features can be highlighted to navigate through the source code:

- A dialog that shows the opened files and IDE windows can be used to open the desired element without using the mouse
- You can navigate to a declaration from a usage
- You can navigate to a usage from a declaration
- A breadcrumb helps to access files that are in the current path or someplace near the actually used file, as shown in the following screenshot:

```
entities ⟩ src ⟩ org ⟩ example ⟩ dao ⟩ C PersonDao ⟩

C MainForm.java ×   C PersonDao.ja  I IPersisterDao   ctory.ja

38                                  I IRetrieverDao
39   private void configurePhonesA   C PersisterDao  rson p
40       for(EntityPhone phone:pe                    ())
41           phone.setPersonByPers   C PersonDao
42       for(EntityEmail email: pe   C RetrieverDao  d())
43           email.setPersonByPersonid(person);
```
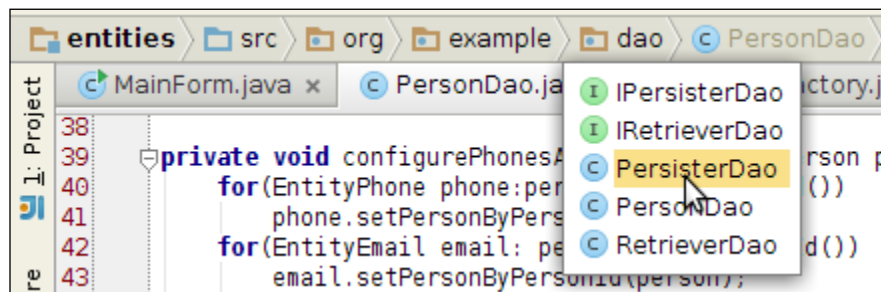
- Small symbols are available on the left side of the code editor for navigation. Clicking on these icons will focus the cursor in the desired code as shown in the following screenshot:

```
@Override
public void edit(T entity) throws Exception {
    sessionFactory.getCurrentSession().merge(entity);
    managedEntity=entity;
}
```
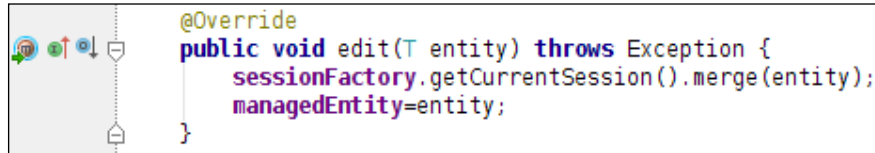
- It is possible to jump to the last change the developer did, no matter if the file is in the current module or closed

For running a search, the following features can be highlighted:

- You can find the usages of a piece of code by just clicking on one menu

- Structural searches can be used to find and replace sentences that can't be represented by a simple string pattern, such as finding only the values of field initializers

- It is possible to find elements in places where you can't see any search field, for example, you can search for a class in the Project Structure tree

- You can use a regular expression for find and replace

Tools and frameworks support

IntelliJ provides support for different tools that help you to develop your application using it. The integration between the tools and the IDE helps developers to focus and reduces some reworks, mainly because the changes that are made by one tool reflect in another tool and, sometimes, starts a refactoring process. A lot of frameworks have advanced support in IntelliJ; to cite some of them: Spring, Play, Grails, JSF, Struts, Flex, JPA, Hibernate, and so on. The support for these frameworks, in conjunction with the integration of different tools, makes the development very simple and productive.

- IntelliJ has a **Unified Modeling Language** (**UML**) tool that you can use to create your classes or just to see how your classes are organized. The UML tool uses refactoring techniques, so you can change a class in the UML editor and be secure that the code will be changed too.

- Basically, you won't need to use another program to create databases, tables, keys, and even to store procedures. No matter which database you are using, IntelliJ will guess the correct dialect for the database; in the rare case it doesn't choose well, you can configure the SQL dialect of the project, file, or directory. There is a powerful graphical table editor available, so you can do the majority of your work in a visual environment, as shown in the following screenshot:

- IntelliJ supports the most known **Version Control Systems** (**VCS**) such as Git, subversion, CVS, Mercurial, and more. With this tool the user can work with different VCSs in a similar way. You can view the differences, changes, and history, manage change lists, integrate with your favorite bug tracker, and more.

- IntelliJ is also fully integrated with the following most commonly used build tools: Maven, Ant, Gradle, and Gant. They provide good integration for unit tests and not only to Java code, but also to JavaScript, Flex, Android, PHP, and others too.

What is new

The Version 12 of IntelliJ comes with good enhancements and new features, such as the following:

- **JavaFX 2**: Now it is possible to use code completion, navigation, refactoring, and CSS facilities in JavaFX 2 projects

- **Java 8**: In this version, the next generation of the Java platform is fully supported with code assistance for the new syntax, such as Lambda expressions, type annotations, default methods, and method references

- **Android UI Designer**: IntelliJ provides a full-featured UI Designer for Android; this way, you can build advanced layouts for Android mobile devices by simply dragging elements

- **Spring Frameworks**: This framework was already supported, however, new improvements were made that bring performance enhancements and better code assistance for a wider range of Spring Frameworks, such as Security and Batch

- **Flex Flash and AIR**: It provides support for Adobe Gaming SDK, including AIR SDK, Starling, Feathers, and Away3D

- **Drools Expert**: The rule-based declarative language from JBoss has code assistance and refactoring tools available

- **Cloud Tools**: You can deploy, run, and monitor your applications in Cloud Foundry and CloudBees

Installing

Now that you know some features of IntelliJ, we need to install it. On IntelliJ's website `http://www.jetbrains.com/idea/`, you will find two editions of IntelliJ: Community and Ultimate.

As you've imagined, IntelliJ IDEA Community Edition is a free version of the IDE, but, unfortunately, it has lots of limitations. For example, in the Community version you won't have UML and database tools, nor the frameworks' ease for Spring, JPA, and so on, neither will you have support for languages like JavaScript, CSS, and others. You can see the differences between the versions in the following table; notice that all the things the Community Edition supports are also supported by the Ultimate Edition.

Community Edition	Ultimate Edition
Features	
Android support	Database tools
Build tools	UML designer
Unit testing	Code coverage
Issue tracking integration	Dependency structure matrix
Local history	Maven dependency diagram
Context management	Structural search and replace
Eclipse project interoperability	Code duplicates detection
Swing GUI designer	
Code spell checker	
OSGI	
Frameworks and technologies	
	Spring Framework
	Play Framework
	Java EE 6
	Google Web Toolkit
	Java Persistence API, Hibernate
	Struts
	Adobe Flex, AIR
	Grails
	Griffon
	Sass, LESS
	Rails, RubyMotion
	Django
	Node.js
	Cucumber for Java Groovy
	AspectJ
	JBoss Seam
	Tapestry

Application server	
	Tomcat
	TomEE
	Glassfish
	JBoss
	WebLogic
	WebSphere
	Geronimo
	Resin
	Jetty
	Virgo
Version Control	
Git, GitHub	Team Foundation Server
Subversion	ClearCase
Mercurial	Perforce
CVS	Visual SourceSafe

For more details about the features supported by each version, go to `http://www. jetbrains.com/idea/features/editions_comparison_matrix.html?IU`.

In this book, we will work only with the Ultimate Version 12.1.x. Don't worry, you can download an evaluation version and use it for 30 days. You can run IntelliJ on three different operating systems: Microsoft Windows, GNU Linux, and Apple Mac. Independent of the OS, you will need the following configuration to run IntelliJ:

- 1 GB RAM minimum, 2 GB RAM recommended
- 300 MB hard disk space plus at least 1 GB for caches
- 1024 x 768 minimum screen resolution
- JDK 6 or higher

You can run it on the following operating systems:

- Microsoft Windows 8/7/Vista/2003/XP (including 64-bit)
- Mac OS X 10.5 or higher, up to 10.8 (Mountain Lion)
- GNU/Linux with GNOME or KDE desktop

The installation of IntelliJ is so simple that it can be done in the following way:

On Windows, the steps are as follows:

1. Run the downloaded file.
2. Follow the steps in the wizard.

On Mac, the steps are as follows:

1. Mount the downloaded `.dmg` file as another disk.
2. Copy IntelliJ IDEA to your `Applications` folder.

On Linux, the steps are as follows:

1. Unpack the `.tar.gz` file.
2. Run `idea.sh` from the `bin` subdirectory.

Configuring

The first time you open IntelliJ, an initial configuration wizard will be shown to you. This wizard will ask you which technologies you want to enable or disable, then it will pick plugins or a group of dependent plugins. It is recommended that you just enable the plugins that you will use because the more plugins you have enabled, the more time is spent in starting the IDE. Also, you can enable or disable any plugin when you want, in the settings dialog.

The majority of the time, you won't need to configure anything in IntelliJ. For example, if your project uses a Git repository, when you open the project in IntelliJ, it will configure its VCS tool automatically based on the Git's files present in the `Git` folder. Even if it is necessary to configure something, you will probably see a balloon on the upper-right corner indicating the event and a link which will direct you to the place where you can take action.

Anyway, all configurations are done in the settings dialog. Even when you see options in the context menus or in balloons, they are just shortcuts to sections of the settings dialog. You will also see them in some tabs and buttons used to configure it, but they are shortcuts too.

As you finish the initial configuration, IntelliJ will open the initial window where you can do some actions, such as creating a project or configuring the IDE. In this initial window, you can open the **Settings** window (or by pressing *Ctrl + Alt + S* on Windows or *command + ,* on Mac when in the main IntelliJ window). In the **Settings** window, you will see various items, some of which are specific for the plugins you have enabled, others that are used to configure the entire IDE, and others for project configuration. They are divided in two big sections: **Project Settings** and **IDE Settings**.

The **Project Settings** section is used to change configurations in the IDE that are specific to the current project, such as the compiler, deployment, and VCS. On the other hand, the **IDE Settings** section is the place where you will manage the configurations that will impact all current and future projects you have, such as **Plugins**, **Notifications**, and **Keymap**. Navigating through the items of the settings window is really simple, but some of them are hidden in a tree structure, which makes it difficult to find them when we need. Fortunately, this can be solved using the search box, as shown in the following screenshot. As I mentioned before, searches are available in practically all IDE elements.
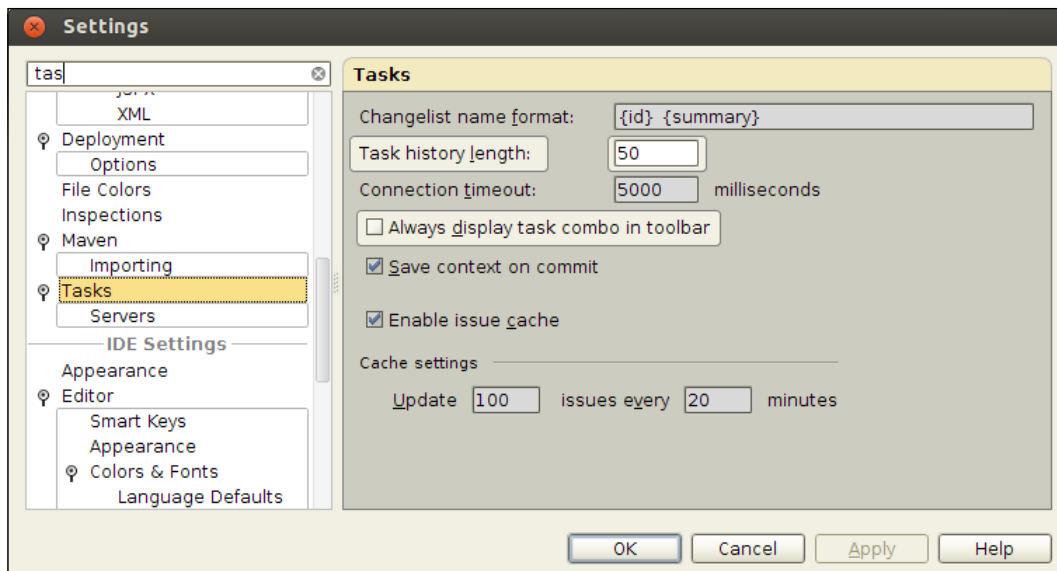
Project Structure

This dialog is used to configure the structure of the project and can be opened by clicking on the **Project Structure** button present in the main window or using the shortcut *Ctrl + Alt + Shift + S* (or *command + ;* in Mac). It's divided into the following five categories:

- **Project**: This section is used to configure the general settings for the project, such as project name, the Java version used, the language level, and the path for the compiled output.

- **Modules**: This module is a discrete elementary unit of the functionality which you can compile, run, test, and debug independently. In this section, you can configure settings that are specific to each module in the project, such as the dependencies, package's prefix, and paths.

- **Libraries**: This section is used to manage the libraries used in the project. Even if the libraries are managed by an external tool, such as Maven, they can be managed in this category.

- **Facets**: Facets represent various frameworks, technologies, and languages used in a module. Here, you can manage the facets of the module. This way, you can define if the support for a particular framework is enabled for it (for example, Spring). Once a facet is activated in a module, IntelliJ will provide facilities to work with that facet.

- **Artifacts**: An artifact is a draft that defines the layout of the project output. It could be an archive (JAR, WAR, EAR) or a directory. In other words, an artifact defines what the final output of the project will be. In this section, you can configure the properties for the artifact of the project.

The following sections in the chapter are two other categories related to **Platform Settings**: **SDKs** and **Global Libraries**.

Virtual machine options

Perhaps someday you will notice that IntelliJ may be running slow; this may happen, for example, when your project has lots of classes. To avoid this, we could increase the available memory used by IntelliJ. Simply edit the file `INTELLIJ_FOLDER/bin/idea.vmoptions` (or `INTELLIJ_FOLDER \bin\idea.exe.vmoptions` on Windows). What you discover upon opening this file are the virtual machine options used to run IntelliJ; so, if you want to change other things related to Java VM, which executes IntelliJ, this is the place to do so.

You should take care while changing or adding options in this file. Depending on the options you use here, the speed of IntelliJ could get worse. For example, setting values very high for the arguments `-Xms` and `-Xmx` can make garbage collector decrease its performance. On the other hand, very small values could create an OutOfMemoryError in IntelliJ. You can use the memory usage indicator to help you to configure these arguments. It is localized in the bottom-right corner of the IDE. If the bar indicates that it is using almost the total reserved memory (for example, using 235 MB of 256 MB), it is recommended that you increase the memory reservation argument by at least 500 MB, as shown in the following screenshot:
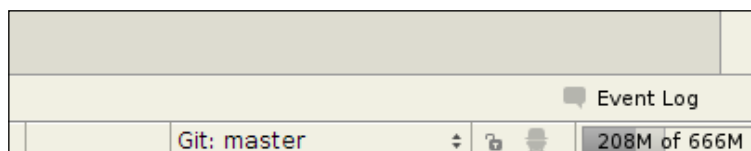
Migrating from another IDE

If you are a user of another IDE (for example, Netbeans or Eclipse), you will notice that some differences exist in IntelliJ that could make you go crazy when using this IDE.

- The nomenclature—the name of a feature in another IDE is probably different from that in IntelliJ (for example, project in Eclipse is the same as module in IntelliJ).

- IntelliJ saves the files automatically, so if you try to close the file to get the old content, it won't work; in this case you can use the **History** tab.

- The shortcuts are different. You can change the keymaps (the relation between the shortcut and the action), although there are predefined keymaps for Eclipse and Netbeans users. A better approach is learning from scratch.

- IntelliJ IDEA compiles files only when it's needed to save system resources for other tasks. IntelliJ is smart enough to know which files have changed, so it recompiles just what is needed. However, if you still want to make IntelliJ compile on every save, you can use the EclipseMode plugin.

- It doesn't have the deploy-on-save feature. However, you can use the **On frame deactivation** option available in the dialog or, perhaps, you can use other options, such as the JRebel plugin for IntelliJ.

- In the other IDEs the user can open multiple projects in one window; in IntelliJ a window is tied with only one project.

The JetBrains website provides complete lists of FAQs. for Netbeans at `http://confluence.jetbrains.com/display/IntelliJIDEA/IntelliJ+IDEA+for+NetBeans+Users` and Eclipse's users at `http://confluence.jetbrains.com/display/IntelliJIDEA/IntelliJ+IDEA+for+Eclipse+Users`, which can help you perform the transition to IntelliJ more easily.

What to expect in the near future

At the time this book is being written, Jetbrains is developing a new release of IntelliJ. The new version will show improvements mainly in Android development. If you don't know, IntelliJ IDEA is the base of Android Studio, an IDE focused exclusively on Android applications development. The new features and facilities that are present at this time, in Android Studio will be available in IntelliJ IDEA 13, with the addition of other tools and features already present in this IDE. We can highlight some features that will probably be present in the next release:

- Support for importing the configuration of Android projects from `build.gradle` and building them through Gradle

- External annotations (`@Nullable`, `@MagicConstant`) for Android APIs
- Folding of Android string resources in the Java editor to display the actual string value
- Many new Lint Checks
- Initial support for editing the RenderScript code
- Support for working with fragment resources
- Support for control variations in the palette (for example, multiple variations of TextField with different default settings)
- Integrated 9-patch editor
- New UI for viewing the list of connected devices in the Android Tool Window

Summary

In this chapter, some features were presented to you just to let you know what you can expect while using IntelliJ. As you've already seen, the tools are totally integrated with the IDE. This will give you the confidence to make the changes you need without worrying about the time you would spend on them. The fact that almost all configurations are centralized in the same dialog, added to the search options available in basically every control, gives you facilities to find what you need in a fast way.

You will probably suffer the first few times you use it; this is because IntelliJ uses a different approach to solve the same problems. However, when you're familiarized with it's way of work, you probably won't want to work any other way.

In the next chapter, you will learn more details about the organization of the IDE and how to explore specific features to improve your development performance.

2
Improving Your Development Speed

What all developers want is to do their job as fast as they can without sacrificing the quality of their work. IntelliJ has a large range of features that will reduce the time spent in development. But, to achieve the best performance that IntelliJ can offer, it is important that you understand the IDE and adapt some of your habits.

In this chapter, we will navigate through the features that can help you do your job even faster. You will understand IntelliJ's main elements and how they work, and beyond this, learn how IntelliJ can help you organize your activities and the files you are working on. To further harness IntelliJ's abilities, you will also learn how to manage plugins and see a short list of plugins that can help you.

Identifying and understanding window elements

Before we start showing you techniques you can use to improve your performance using IntelliJ, you need to identify and understand the visual elements present in the main window of the IDE. Knowing these elements will help you find what you want faster. The following screenshot shows the IntelliJ main window:
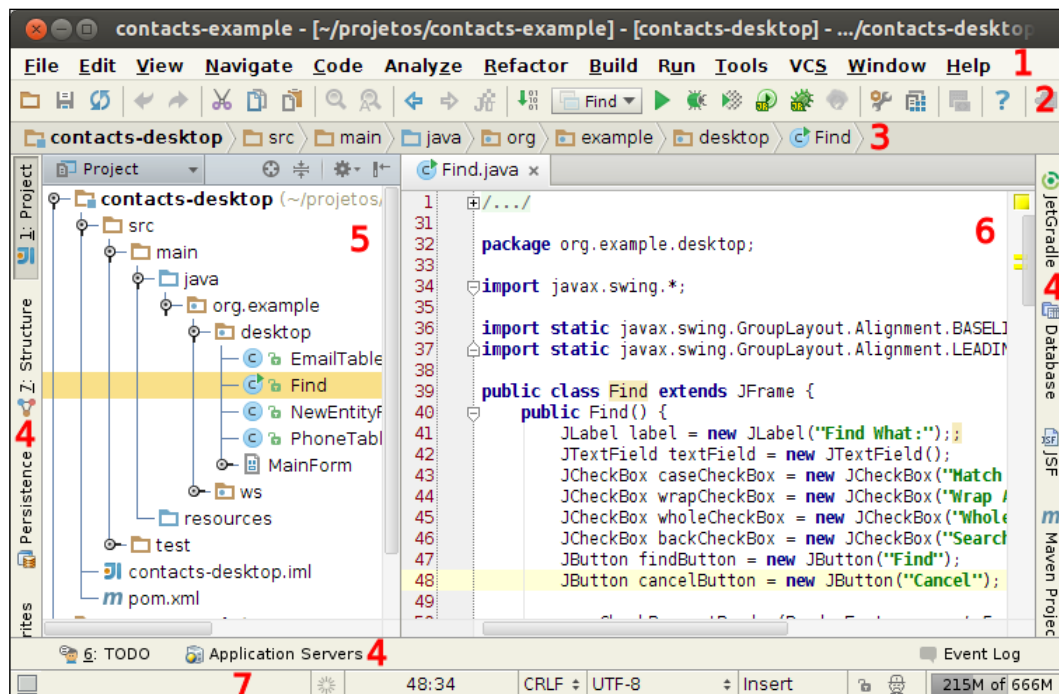
The main window can be divided into seven parts as shown in the previous screenshot:

1. The main menu contains options that you can use to do tasks such as creating projects, refactoring, managing files in version control, and more.
2. The main toolbar element contains some essential options. Some buttons are shown or hidden depending on the configuration of the project; version control buttons are an example of this.
3. The Navigation Bar is sometimes a quick and good alternative to navigate easily and fast through the project files.
4. Tool tabs are shown on both sides of the screen and at the bottom of IntelliJ. They represent the tools that are available for the project. Some tabs are available only when facets are enabled in the project (e.g. the **Persistence** tab).

5. When the developer clicks on a tool tab, a window appears. These windows will present the project in different perspectives. The options available in each tool window will provide the developer with a wide range of development tasks.

6. The editor is where you can write your code; it will be better covered in the next section.

7. The Status Bar indicates the current IDE state and provides some options to manipulate the environment. For example, you can hide the tool tabs by clicking on the icon at the bottom-left of the window.

In almost all elements, there are context menus available. These menus will provide extra options that may complement and ease your work. For example, the context menu, available in the tool bar, provides an option to hide itself and another to customize the menu and toolbars.

You will notice that some tool tabs have numbers. These numbers are used in conjunction with the *Alt* key to access the tool window you want quickly, *Alt + 1*, for example, will open the **Project** tool window.

Each tool window will have different options; some will present search facilities, others will show specific options. They use a common structure: a title bar, a toolbar, and the content pane. Some tool windows don't have a toolbar and, in others, the options in the title bar may vary. However, all of them will have at least two buttons in the rightmost part of the title bar: a gear and a small bar with an arrow. The first button is used to configure some properties of the tool and the second will just minimize the window. The following screenshot shows some options in the **Database** tool:
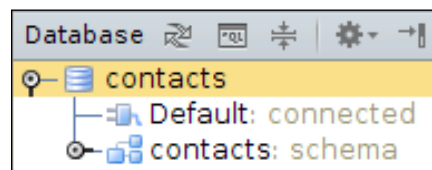
The options available under the gear button icon generally differ from tool to tool. However, in the drop-down list, you will find four common options: **Pinned**, **Docked**, **Floating**, and **Split** modes. As you may have already imagined, these options are used to define how the tool window will be shown. The Pinned mode is very useful when it is unmarked; using this, when you focus on code editor you don't lose time minimizing the tool window.

Identifying and understanding code editor elements

The editor provides some elements that can facilitate navigation through the code and help identify problems in it. In the following screenshot, you can see how the editor is divided:
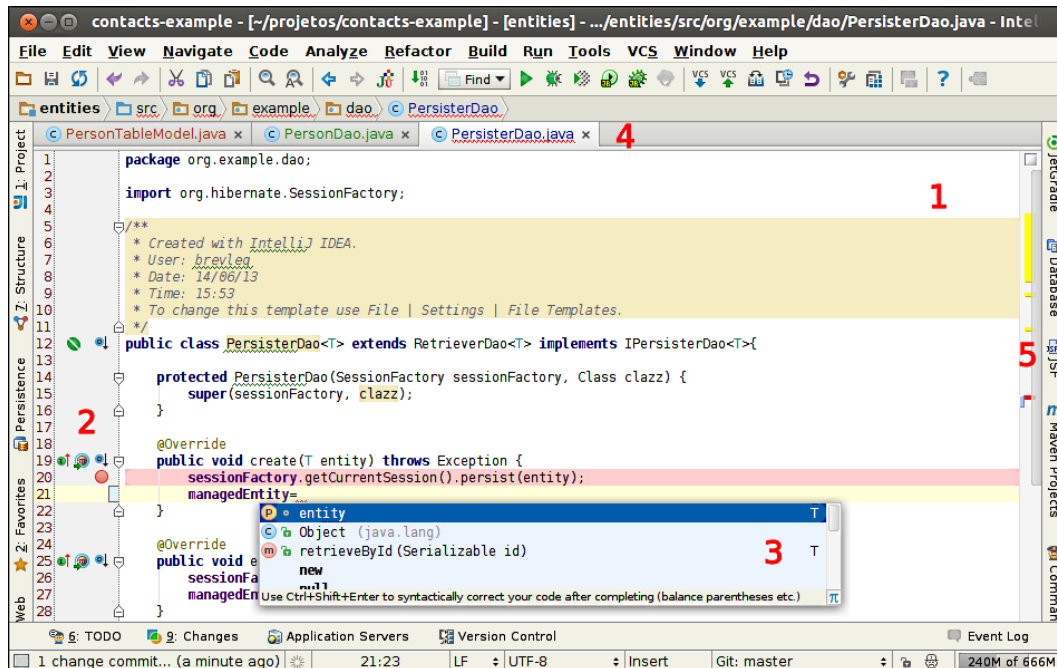
1. The editor area, as you probably know, is where you edit your source code.

2. The gutter area is where different types of information about the code is shown, simply using icons or special marks like breakpoints and ranges. The indicators used here aren't used to just display information; you can perform some actions depending on the indicator, such as reverting changes or navigating through the code.

3. The smart completion popup, as you've already seen, provides assistance to the developer in accordance with the current context.

4. The document tabs area is where the tabs of each opened document are available. The type of document is identified by an icon and the color in the name of the file shows its status in version control: blue stands for "modified", green for "new", red for "not in VCS", and black for "not changed". This component has a context menu that provides some other facilities as well.

5. The marker bar is positioned to the right-hand side of the IDE and its goal is to show the current status of the code. At the top, the square mark can be green for when your code is OK, yellow for warnings that are not critical, and red for compilation errors, respectively. Below the square situated on top of the IDE this element can have other colored marks used to help the developer go directly to the desired part of the code.

Sometimes, while you are coding, you may notice a small icon floating near the cursor; this icon represents that there are some intentions available that could help you:

indicates that IntelliJ proposes a code modification that isn't totally necessary. It covers warning corrections to code improvement.

indicates an intention action that can be used but doesn't provide any improvement or code correction.

indicates there is a quick fix available to correct an eminent code error.

indicates that the alert for the intention is disabled but the intention is still available.

The following figure shows the working intention:
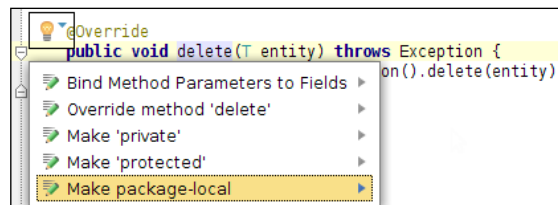
Intention actions can be grouped in four categories listed as follows:

1. Create from usage is the kind of intention action that proposes the creation of code depending on the context. For example, if you enter a method name that doesn't exist, this intention will recognize it and propose the creation of the method.
2. Quick fixes is the type of intention that responds to code mistakes, such as wrong type usage or missing resources.
3. Micro refactoring is the kind of intention that is shown when the code is syntactically correct; however, it could be improved (for readability for example).
4. Fragment action is the type of intention used when there are string literals of an injected language; this type of injection can be used to permit you to edit the corresponding sentence in another editor.

Intention actions can be enabled or disabled on-the-fly or in the Intention section in the configuration dialog; by default, all intentions come activated. Adding intentions is possible only after installing plugins for them or creating your own plugin. If you prefer, you can use the *Alt + Enter* shortcut to invoke the intentions popup.

Doing things faster

The great advantage in using IntelliJ is the possibility to make your job faster; however, it will only be possible if you know there are some tricks you can use.

The productivity guide

IntelliJ provides a good way to show you what can be done in the IDE to improve your performance. In the **Help** menu, you will find an option called **Productivity Guide**; as you may notice, this is a guide that will help you improve your productivity.

The guide analyzes the usage of IntelliJ and identifies the features you use a lot and the ones you practically don't use, so it can create statistics and configure the **Tip of the Day** window, showing the most relevant tips for when the developer is performing time-consuming operations.

In the productivity guide dialog, as shown in the following screenshot, you will see the quantity of time for which you used a feature and a description of the feature. The description is the same as that used in the **Tip of the Day** dialog and is very useful when you search for features you never used. Even knowing you can identify unused features in this guide is, for a new user, important to keep the **Show Tips on Startup** option (in the **Tips of the Day** window) checked; this is the simplest way to discover the abilities of IntelliJ.
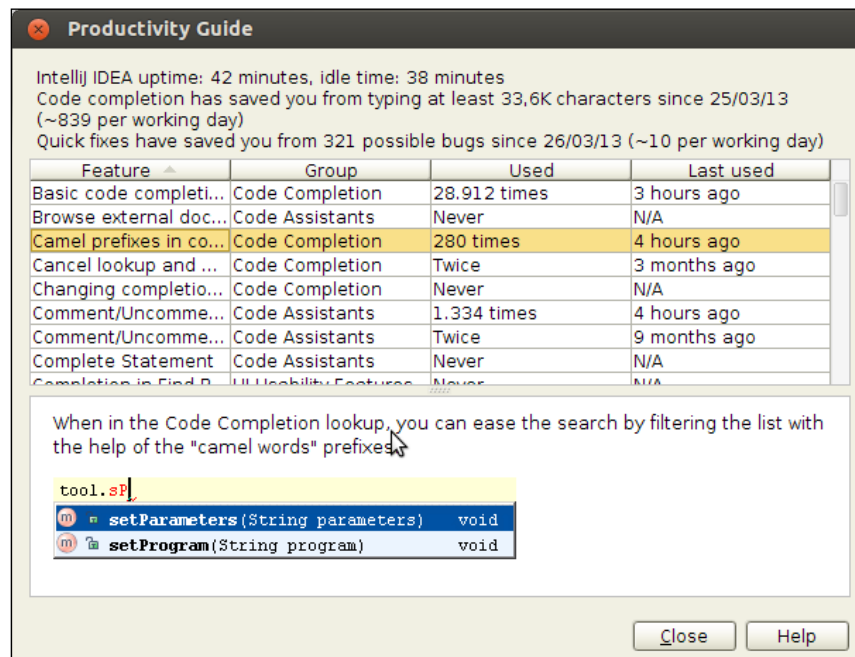
Navigating your source code

The faster you navigate through the source code of your project, the sooner you can get things done. I have already presented, in *Chapter 1*, *Getting Started with IntelliJ IDEA 12*, two ways to navigate the source code: using the **Project** tool window and the Navigation bar; however, there are better ways to navigate directly to what you want.

A faster way to execute some actions in every application is using shortcuts. IntelliJ has a lot of them to make the most different things and some of them are used to navigate:

- *Ctrl + B* (or *Command + B* in Mac) directs the developer to the declaration of methods, fields, constructors, and so on. To use this shortcut correctly, the developer needs to hover the cursor over the element being used.

- *Ctrl + N* and *Ctrl + Shift + N* (or *Command + N* and *Command + Shift + N* in Mac) are used to navigate to a class and a file, respectively; you just need to type the name of the resource you want.

- *Ctrl + Alt + B* (or *Command + Option + B* in Mac) directs the developer to the implementation of an element (class, interface, method, and such). In case the element has more than one implementation, a list with all implementations will appear.

- *Ctrl + E* (or *Command + E* in Mac) can be used so the developer can see the recently opened files, allowing him/her to reopen them easily and fast.

- Ctrl + Tab will display a simple dialog with opened files and available tool tabs.

- *Ctrl + F* (or *Command + F* in Mac) shows a find text field to run a search for sentences in the current file. You can configure regex to find things and use other options available in the find bar.

- *Ctrl + Shift + F* will open a dialog that will allow you to run a search for a sentence through all the files in the project. The developer can filter where the search will be done and define some other options.

The small list of shortcuts above is intended to show you some features directed to navigation; a complete list of shortcuts for your preferred operational system can be downloaded from `http://www.jetbrains.com/idea/documentation/index.jsp`.

In conjunction with shortcuts, IntelliJ provides visual elements that can be used to navigate through and understand the real state of the code. Some of these visual elements can be observed in the gutter area confirming that you are changing your source code. There will be different icons according to the type of code you are editing; these icons are important for two reasons: you can use them to navigate and they will show you if you have correctly done what you wanted. For example, if the Spring icon appears next to a setter, you know the bean property declaration in the XML file worked correctly. As with the gutter area, the marker bar can be used to navigate the source code; however, the marks here just navigate the content of the current file. The following image shows a list of the implementations of a method of an interface when the developer clicks on the Implementation icon:
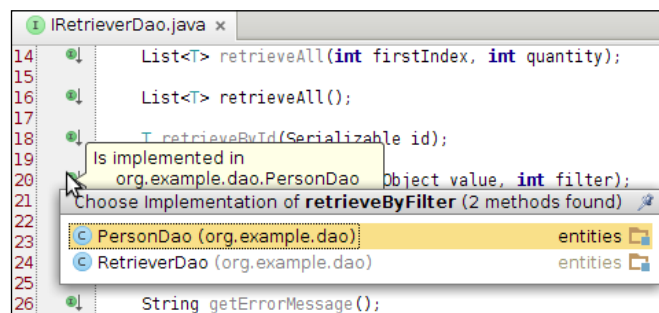
Sometimes, you will need to see the current file open in the **Project** tool tab and perhaps you will see the source tree collapsed. Depending on the quantity of hierarchies and files, it can take some time. To minimize the time spent on this task, you can use the scroll from source button available in the title bar in the **Project** window; this way IntelliJ can expand the source tree and select the current file in use, as you can see in the following screenshot:
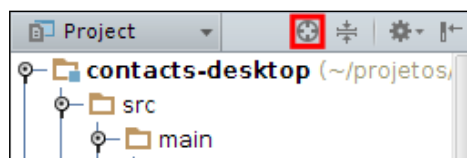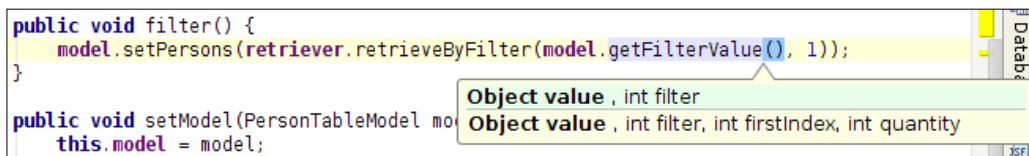
Using code facilities

IntelliJ provides good code facilities; however, sometimes you could under-utilize some of them, probably because you don't know they exist. Generally, the code completion feature is activated when the developer is typing; however, sometimes, the developer should actively force the code completion feature. The following list presents some useful shortcuts when working directly with the code:

- *Ctrl* + space bar key: In the rare case that code completion isn't activated automatically, use this shortcut to active it.

- *Ctrl* + *Shift* + space bar key: Advanced features of code completion are only activated when the developer uses this shortcut. In the case of chained code completion (this improved code completion can suggest code of expected type traversing the method chain), it is only available if you use the last shortcut twice.

- *Shift* + *Delete*: This shortcut is used to cut the entire line; just position the cursor in the line you want cut and it will cut the content to the clipboard.

- *Ctrl* + *D*: Use this shortcut when you need to duplicate a line; position the cursor in the line you want duplicated and use it.

- *Ctrl* + */* and *Ctrl* + *Shift* + */*: Use these when you want to comment on the entire line or a selection, respectively; to uncomment, just do the same action.

- *Ctrl* + *Alt* + *L*: This shortcut can be used to reformat your code; however, if you are using Ubuntu, this shortcut will lock your screen. Don't forget that you can change the shortcuts in the **Settings** dialog.

- *Ctrl* + *Alt* + *O*: This shortcut will optimize the imports.

- *Ctrl* + *Shift* + up arrow key and *Ctrl* + *Shift* + down arrow key: Use these shortcuts when you want to reposition lines, methods, and sentences to a position higher or lower in the code. If the cursor is hovering over a line or selection, these shortcuts will only move it up or down. To move methods, just position the cursor in the method declaration and use the shortcut; you can move a sentence in the same way. In all cases, IntelliJ will analyze if the move is permitted; it won't permit you to move a line from inside a method to the outside, for example.

- *Ctrl* + *O*: This shortcut can be used to override the methods of the superclass, including methods of an interface that aren't implemented. IntelliJ will create a default method body, generally calling the super method.

- *Ctrl* + *I*: This shortcut can be used to implement the methods of interfaces that aren't already implemented.

- *Alt + Insert*: If you need to generate constructors, getters, setters or both, or implement, override or delegate methods, and so on, use this shortcut.

- *Ctrl + Q*: Are you in doubt about some functionality in the code? This shortcut will open JavaDoc for the element with the cursor.

- *Ctrl + P*: Using this shortcut, you can easily see which parameters a method expects. As you can see in the following image, for each method overloading, it will show a line:

```
public void filter() {
    model.setPersons(retriever.retrieveByFilter(model.getFilterValue(), 1));
}

public void setModel(PersonTableModel mo
    this.model = model;
```
Object value , int filter
Object value , int filter, int firstIndex, int quantity

- *Ctrl + Shift + V*: Generally, the applications and the operating system only provide one memory slot to copy and paste. This shortcut will open a dialog with the last five items copied.

You will notice that you can select a suggestion IntelliJ made using the enter key; however, the suggestion will expand and won't override the sentence already present in the line (if this is the case). You may lose some time deleting the characters after the expanded sentence. To avoid this, you can select the suggestion using the tab key and this way the previous sentence will be swapped and the expanded sentence will occupy its space.
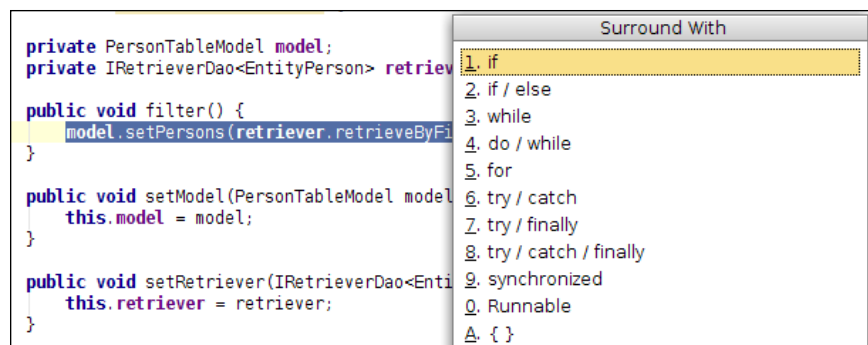
Live templates

To speed up your development even more, IntelliJ provides live templates. They are predefined code fragments to help you insert frequently-used or custom code in your source. Live templates can be of three types, listed as follows:

- **Simple template**: It provides a fixed abbreviation that expands into plain text. The code present in the template is automatically inserted in the source code, replacing the template abbreviation. As an example, the `psvm` abbreviation in the body of a class will create the default main method.

- **Parameterized template**: It has the ability to use variables when the template is expanded. The places where a variable is needed will show an input field where the user may fill the value or accept the one proposed by the IDE. For example, the `fori` abbreviation will create a `for` sentence and position the cursor in places where it needs a variable or parameter.

- **Surround template**: It surrounds the selected text with code before and after it. It is commonly used to surround the selected text with a `try/catch` block.

Live templates can be activated using the *Ctrl + J* shortcut and selecting the desired option from the list or typing the abbreviation and clicking on the template invocation key (by default, it is *Tab*). If you look carefully, the abbreviations follow a pattern; for example, iteration templates start with it, followed by the abbreviation of the group that will be iterated.

To use the Surround template, you first need to select a sentence in the source file; after this, you should use the *Ctrl + Alt + T* shortcut. When you do this, a dialog appears asking which type of surrounding you want; just select one and press enter as shown in the following screenshot:

IntelliJ comes with a default list of live templates; this list covers a lot of possibilities but, if you need one specific template, you can create your own in the **Settings** dialog. By default, the list of live templates will be shown only when you use the *Ctrl + J* shortcut. This way, if you type for in the code editor, the `fori` live template won't be suggested. However, we can change this behavior.

Go to **Help** | **Find Action** (or simply press *Ctrl + Shift + A*) and type `Registry` in the search field; make sure that the option **Include non-menu actions** is checked. When you select the registry result, the **Registry** dialog will appear. Find the **optionshow. live.templates.in.completion** and check it. In the following figure, you will see live templates in the completion list even if you didn't ask for them:
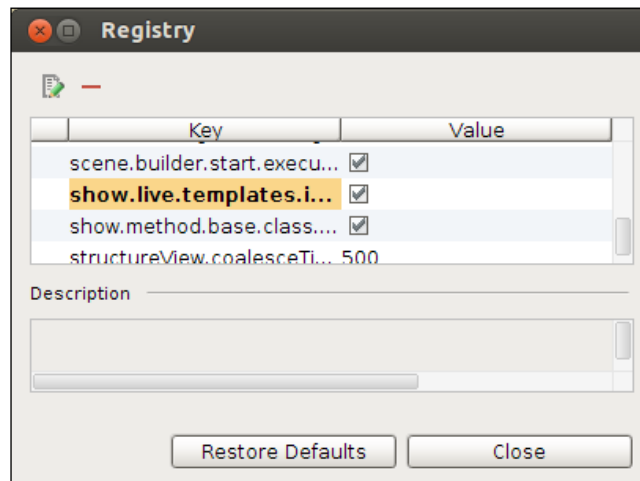
Using refactoring techniques

As you already know, refactoring is a sensitive task that should be performed with extreme care; a simple mistake in a refactoring could generate lots of bugs. IntelliJ provides a large range of facilities to help you in the refactoring process. Even with IntelliJ being so smart, it could make some mistakes and it is your responsibility to avoid this. In the **Refactor** menu, you will find all possible refactor options, some of which will be disabled depending on the position of the cursor. The shortcuts have an important role here as in other parts of the IDE. However, you will notice that the **Refactor** menu has a bunch of submenus that don't have any shortcut; because of this, there is a shortcut that will help you a lot when you don't want to lift your hands off the keyboard—*Ctrl + Alt + Shift + T*. This shortcut will open a pop-up that will show all the refactoring options available for the piece of code that the cursor is hovering over.

Even with that shortcut helping a lot, there are other shortcuts that help the developer and *Shift + F6* is one of them. This shortcut permits us to rename the element that the cursor is hovering over and changes all usages of this element. I should alert you about the usage of the **Rename** refactor in JSF files—this approach runs a search in other JSF files and changes sentences in places that you didn't want to modify. For example, you are using the `var` element in the datatable of an XHTML file and, coincidently, another XHTML datatable has the same value in the `var` element. If you proceed with the refactoring, all usages of that name in other XHTML files will change too. There is a worse side-effect; if you use primefaces datatables, you will notice that some usages of the `var` element in the same datatable won't change (for example, the `rowKey` option). But don't worry, this is the only casein which I had problems using the Rename refactor and, in this case, I used the replace action (*Ctrl + R*).

Even though it is bad practice, sometimes we need to copy a class to use some code that has already been implemented to solve a similar problem. The approach of "copy a class and paste a class" was taking up too much of my time because I needed to use the mouse to select the class file in the project tree. After some time, I discovered that IntelliJ provides a simple utility to do this job; basically, you just need to press *F5* and enter the name of the new class in the dialog that will appear. If you prefer, in this dialog, you can specify where the new file will be stored.

Another important facility is the move refactor; using *F6* you can move a file to another place, even as an inner class. To change the signature of a method, just use *Ctrl + F6* and the options available to add and remove parameters in the change **Signature** dialog that will appear. If you need to migrate a type of method, variable, or parameter, just position the cursor over the element and press *Ctrl + Shift + F6* and a new dialog will be shown to help you change the type of the element.

Delegation is the simple concept of handing a task over to another part of the program. IntelliJ provides an easy way of doing this. First of all, call the delegate option available using *Alt + Insert*, then by clicking on an option under **Generate** or the option **Code | Generate...** in the main menu. In the pop up that appears, select the delegate methods, then the object to be delegated, and click on **OK** in the next dialog. Finally, a new dialog will be displayed to permit you to select the methods you want to delegate.

If you need to transform an element to static, the Make Static refactor will do it in a simple way. The option convert to an instance method does the reverse job when a class is passed as parameter. There are other refactor options available that you can use in specific cases, such as Invert Boolean and Remove Middleman. Invert Boolean will simply invert the return of a method that returns Boolean. Remove Middleman is helpful when you have a class that forwards many of its method calls to objects of other classes; using this option, you can simplify your design.

[33]

Managing your changes

If you are an attentive user, you will notice that, in the majority of dialogs used to manipulate your code, there exists a preview button. It is good practice to analyze all the changes IntelliJ will automatically apply in your code before they take effect, so use the preview button all the time, unless you really trust what IntelliJ will do. After you click on the **Preview** button, the find tool will appear (sometimes it won't, but the results will be there) with all usages it found of the element that will be changed. For example, if you want to rename a method, IntelliJ will find every usage of this method and show them in a list so you can review the changes before you proceed. With this tool, you can include or exclude a usage from participating in the refactoring process and decide whether you will proceed with the refactoring or it will be canceled as shown in the following screenshot:
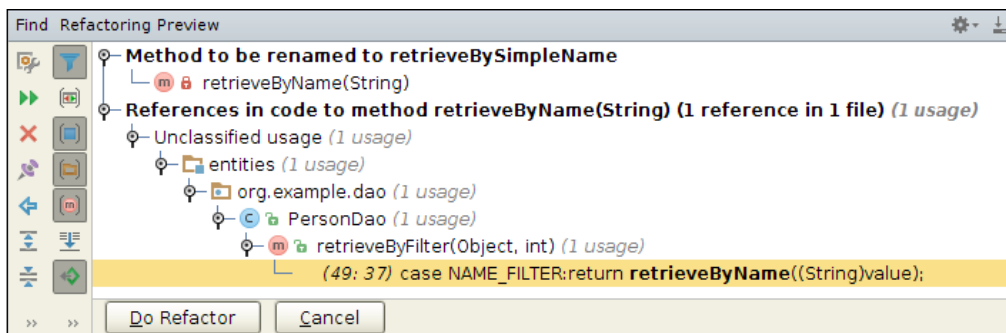
Perhaps you made a modification that didn't please you; for such cases, you have the **Local History** option. Local history is a kind of version control system that is independent of other VCSs, so you don't need any VCS configured to use it and it is active all the time. Different from general VCSs, local history is for personal use only, so you can't share access with other users. It facilitates the development once you can revert your modifications any time you want using the difference window.

To call history to a file, just right-click anywhere in editor (or use the *Alt + ~* shortcut) and you will see the **Local History** option; in that option, you will see three suboptions: **Show History**, **Show History for Selection**, and **Put Label**. The first two options are similar as they show a dialog where the user can see the differences between the versions; however, the **Show History for Selection** option will show only the selected content in the difference dialog. The option **Put Label** is used to define a label to a revision; this way, you will have a more organized history.

In some cases, the local history can be cleared, such as when you install a new version of IntelliJ and when the cache is invalidated. The invalidation of the cache can occur when the cache is overloaded or manually by the user using the option invalidate cache in the file menu. Local history stores a good range of modifications that happen in your code; however, it isn't a full-blown VCS, so if you need real version control, it is better to use a distributed VCS like Git or Subversion.

Organizing your activities

When you are developing, it is important to organize the activities necessary to do your work. Know which tasks should be done and mark something less essential for later. This could help you focus and, consequently, make things easy and fast. IntelliJ provides tools that can be used to organize your activities.

Tasks and context management

IntelliJ provides facilities that can coordinate your work in the IDE; it can be done in accordance with your preferred issue tracker or by creating your own task in the IDE. Before we continue, you need to understand what a task is and a context in IntelliJ.

A task is an activity that needs to be performed; for example, improving search facilities. It has a name, which is an identifier, and can be correlated in an issue tracker system.

Context is a group of files opened in the editor that are vinculated with a task.

Task and context have a close relationship; the act of creating a task will make the IDE clean the workspace, create a change list and a new context. With the workspace clean, all files you open will be part of the context of the task; this is useful, for example, to filter the files that will participate in the commit process. Even with this strong relationship, it is possible save and clear a context independently of any task. In case you open and edit a file that is bound in another context, IntelliJ will open a dialog asking how you want to resolve the conflict in the change list.

By default, you won't see the task combobox in the main window; to use it, you should create your first task. To create a new task, in the main menu, you can navigate to **Tools** | **Tasks & Contexts** | **Open Task** or simply use the shortcut *Alt + Shift + N*. Type a name for your task and choose the option **Create New Task**. A new window will ask if you want to clear the current context. When you click on **OK**, your task will be created and the context combobox, along with your created task, will be visible at the end of the main toolbar.

[35]

The pop up used to create a new task can also be used to select a task; you will see a list of available tasks below the text field where only the open tasks will be available unless you mark the check box, include closed tasks, or press *Alt + Shift + N* again. In case you need more information about a task, you can use *Ctrl + Q* (notice that this same shortcut will open JavaDocs when the focus is on the code); it will open a box to supply this need. It is possible to merge the contexts of the tasks, selecting them with the *Shift* key pressed, as shown in the following screenshot:
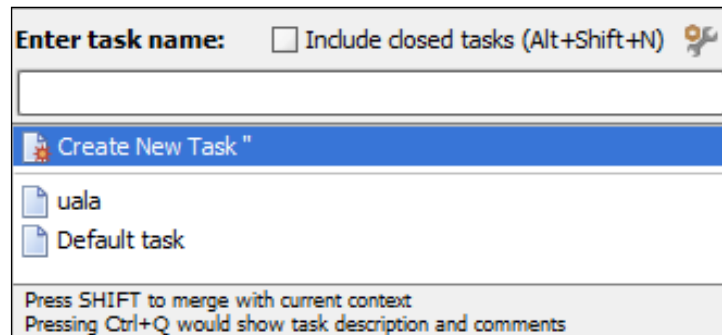

To switch between existing tasks, you can use the combobox or *Alt + Shift + T*; a pop-up will appear with a list of existing tasks and an option to open a task.

Issue tracker system (**ITS**) is a kind of system that is used to manage a list of issues and tasks. It is generally used to report bugs and the new features required by a developer, for example. It is possible to use the tasks available in your favorite issue tracker; you just need configure IntelliJ to use it. You can do this configuration in the **Settings** dialog (**Section Tasks then Server**) or use the icon (gear with a wrench) available in the **Open Task** pop up. No matter which path you choose, in the end you will see a panel with an **Add** button on the right-hand side; when you click on the **Add** button, a list of supported issue trackers will appear to you—select what you want use.

The information asked may vary depending on the issue tracker you are using (e.g. Jira, Youtrack, and so on), but, generally, it is the server URL, user name, and password. If you need, you can configure proxy options using the button **Proxy Settings**. Once you configured the options, you can click on the button **Test** to know if it is configured correctly. To allow access to the specified server for other members of your team, select the **Share URL** check box. To configure how IntelliJ should synchronize with the issue tracker, go to the **Tasks** section in **Settings** dialog.

After the issue tracker is configured in IntelliJ, you will see that when you type in the input text in the open task pop up, the IDE will run a search for the issues synchronized. If it doesn't find anything in the synchronized list, it will synchronize again to search for the task you want. Of course, if the task doesn't exist in the issue tracker, no result will be retrieved, as shown in the following screenshot:
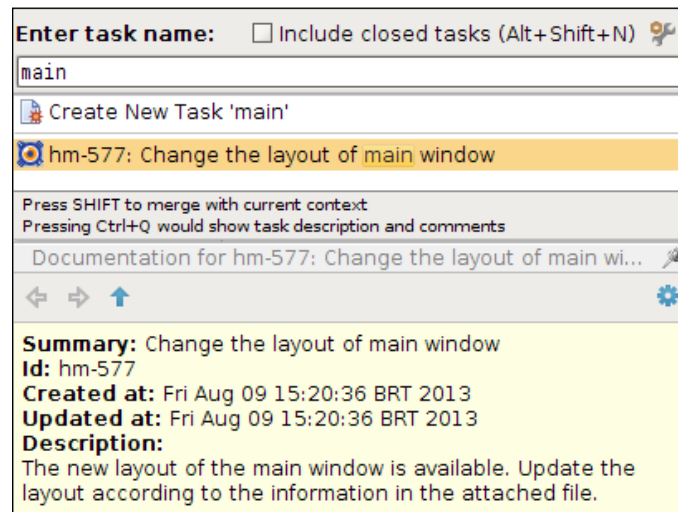
Using TODO marking

Sometimes you will need to postpone some code that isn't so important; however, you know that the code needs to be implemented. To help you remember what you should do but didn't, you can mark your code with the TODO clause.

TODO is a special comment that IntelliJ tracks and processes in a different way. The default pattern is a comment comprising a line with a double slash, and after that, write TODO, followed by anything you want:

```
private void showMessage(Status status) {
  //TODO: something should be done here.
}
```

Differently than with a normal comment, the line with the TODO clause will be shown in a dark blue color, or according to your color and fonts settings. IntelliJ comes with the built-in FIXME clause; its behavior is the same as that of the TODO clause and is useful when you need to filter by type. If you prefer, you can create any patterns you want to be treated as a TODO clause. To do this, open the **Settings** dialog and go to the **TODO** section. There, you will find two divisions; in the first, you can manage the patterns used to identify a sentence as a TODO clause and the second is used to filter the TODO comments when you see them on the TODO tool.

If your TODO window isn't visible in the main dialog, you can open it by going to **View** | **Tool Windows** | **TODO** in the main menu. When you open it, you will see the marks organized in different ways; you just need to choose the tab at the top of the tool to see the marks in different groups.

Another useful feature of TODO is in avoiding a situation where a code marked with this clause is committed. When you try committing a code, a dialog will appear showing some information about changed files and in this dialog exists a check box called **Check TODO**—if any TODO clause is found in the files, it will ask you if you want review the files with TODO. This way, you always be sure no code with TODO marks will be committed without your knowledge, as shown in the following screenshot:
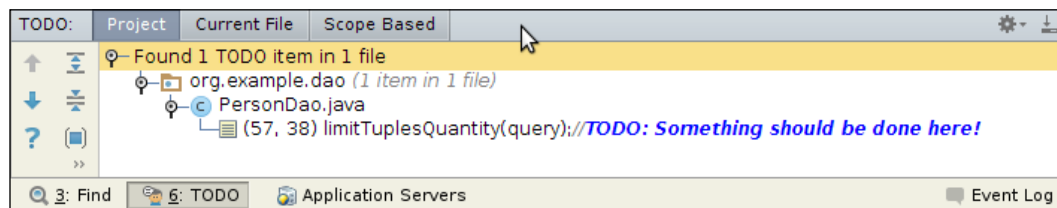
Plugins

Plugins can expand the power of IntelliJ by providing new features. The structure of this IDE is highly directed to plugin usage; plenty of built-in features are implemented this way and shipped with IntelliJ (like Scala support). The use of some plugins can change the way you use the IDE. Before I show you a (really) small list of good plugins, you need know how to install them.

As usual, open the **Settings** dialog and go to the **Plugins** section. In this section, you will see a list with all the plugins that are installed. To enable or disable a plugin, you just need to check (or uncheck) the line, notice that you always enable a plugin that has dependencies, and a dialog will ask if you want activate the referenced plugin. You are free to not activate it, but it is a good idea to do it. As enabling or disabling a plugin is a task that affects the entire IDE, and you will always be asked about restarting IntelliJ.

On the right-hand side of the panel, you will see the description of the selected plugin. There are three buttons at the bottom of the panel; they are used to install a plugin. The first one is Install **JetBrains plugin...**; it will open a list from which you can choose official plugins. The second button is **Browse repositories...**; it opens a dialog from which you can choose third-party plugins. In the dialogs of neither official plugins nor the third-party ones will you see any install button so, to install a plugin in these dialogs, you should right-click on the desired plugin and select the **Download and Install** option. The last button, **Install plugin from disk...**, is used to select plugin files in your computer.

Now, here is a list with some interesting plugins, all of which are developed using third-party plugins. I recommend you navigate through the official plugins list and the third-party list. You will probably find some component that may help you a lot.

JRebel

If you work with web projects using Java, you know that deploys and redeploys are a time-consuming task. Even a little modification in the code would make you lose a lot of time and things may get worse when you notice that your application server has crashed because of the big number of redeploys. I don't know about you, but I don't have time to lose.

JRebel is a commercial plugin that can solve this problem. With JRebel, you won't need to worry about redeploy because it can reload the changed classes on-the-fly, just clicking on the JRebel button added in the main tool bar once the plugin has been downloaded. The magic used by JRebel to reload objects is really powerful, but it can't perform miracles; for example, it can't reload an object that has a class renamed—in these rare cases, you need to redeploy. The following screenshot shows the buttons JRebel will add in the main tool bar:

To use JRebel, you need create a `rebel.xml` file to configure where it will find the classes it needs to reload. There is a free version of JRebel called MyJRebel that can be used in an open source project.

For more information about JRebel, visit `http://zeroturnaround.com/software/jrebel/`.

Hungry Backspace

Sometimes, even with you knowing that better approaches exist, you will insist on using Backspace key to remove excess space between the sentences you created. In cases like this, you will be lucky if you've installed the Hungry Backspace plugin. The following screenshot shows an empty space that can be completely removed with just one press of the Backspace key:

```java
private List<EntityPerson> retrieveByName(String name) {
    Query query = sessionFactory.getCurrentSession().createQuery("select e from Entity
    query.setParameter("name", createSearchParameter(name));
    limitTuplesQuantity(query);



    return query.list();
}
```

As the name suggests, it eats spaces upon hitting Backspace key just once. When Hungry Backspace removes spaces, it makes the content after the cursor indented. With this plugin, you will further reduce the time you spend formatting code.

Key promoter

Key promoter will analyze the usage of IntelliJ and, when you activate an action that has a shortcut using the menu bar, it will show you a message in the upper-right corner of the IDE. It is useful for all users, from the new users that don't know the basics shortcuts to the advanced users that forget the shortcuts all the time. The following image shows the message you receive when you use the menu to rename a variable, for example:

'Shift+F6'
(Rename…)
5 time(s)
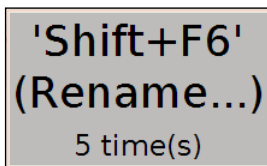
GenerateTestCase

This plugin is a port of an Eclipse plugin with the same name. Its goal is to provide a simple way to create test cases based on comments and annotation that you can put in the method you want to test. For example, the following interface method declaration has a comment stating that it needs to check if the operands aren't null:

```
public interface Test {

    /**
    *
    * @param a
    * @param b
    * @return
    * @should check if operands are not null and add them
    */
    public Integer add(Integer a, Integer b);
}
```

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/supportand register to have the files e-mailed directly to you.

The @should annotation will show what the method should do so that when you create the test class, the plugin will generate one test case for each @should annotation; in this case it will generate the following code:

```
public class TestTest {
    /**
    * @verifies check if operands are not null and add them
    * @see Test#add(Integer, Integer)
    */
    @org.junit.Test
    public void add_shouldCheckIfOperandsAreNotNullAndAddThem()
      throws Exception {
      //TODO auto-generated
Assert.fail("Not yet implemented");
    }
}
```

Now you just need to click on the **GenerateTestMethods** button that is available in main toolbar. You can see more details about this plugin in this YouTube video http://www.youtube.com/watch?v=WYET6PECxuc.

Summary

As you have seen in this chapter, IntelliJ provides a wide range of functionalities that will improve your development speed. More important than knowing all the shortcuts IntelliJ offers is to understand what is possible do with them and when to use a feature. It took me some time until I discovered that IntelliJ provides simpler ways to do some common tasks and I hope you won't need to. Despite having shown lots of IntelliJ's features in this chapter, there are many other features you can use that will ease the way you work, so don't be afraid to explore the IDE to the maximum.

I hope that you can now see how IntelliJ can help you. The features presented in this chapter are powerful and will help you in your life with this IDE. However, don't forget that an IDE is simply a tool and the way you use it is what will help you achieve your goals.

In the next chapter, we show how to extract more of IntelliJ when working with database. We will create a simple project that will cover almost all features related to the database.

3
Working with Databases

Databases are used all the time, in practically all software, directly or indirectly. In this chapter we will learn and understand how to use the tools that IntelliJ provides to ease our work with databases, such as generating tables, managing the data, and even creating ORM entities for JPA or Hibernate.

As the tools used to manage the databases have some specific details, this chapter presents a more practical approach; I hope this way you can improve your abilities in a faster way.

Database tool

One of the more important activities in a project is the modeling and creation of database structures. Depending on the philosophy of the enterprise you work with, each project can adopt a different database; this can make the job of the **Database Administrator** (**DBA**) difficult because he needs to use different tools for each database.

IntelliJ provides a powerful database tool you can use to model and manage any database you want. The only thing you need to configure to use all the power of the tool is the dialect of your database. Generally, it is really simple once IntelliJ suggests a dialect based on the characteristics of the database you are using.

In this chapter, we will work in a more practical way, giving us the knowledge to use the tools correctly and to better establish the concepts.

Creating the database

In the course of this chapter, all activities will be performed using MySQL Version 5.5, so I recommend you to install this version of MySQL to correctly follow the instructions in this chapter. Of course, you can use any database you want; however, it will be your responsibility to understand and adapt the information present here to your favorite database. This is not a big deal as you just need to use a slightly different connection string and syntax.

I'm assuming you already have an instance of MySQL installed and running. As you would know, information about installing and configuring the database server is outside of the scope of this book, especially because this is a simple task that you can do upon running a search for instructions over the internet. Once you have a MySQL installed, let's create a database and see how you can connect with that database using IntelliJ.

To begin, we need to create the database using the MySQL console. To create a database in MySQL, just open Command Prompt on Windows or Terminal on GNU/Linux and connect with the server using the following code:

```
mysql -u root -p
```

When asked, type the superuser password used in MySQL installation. The MySQL console will open. We are going create the `contacts` database, so type the following code in the MySQL console:

```
create database contacts;
```

Connecting to the database

Once the database is created, let's come back to IntelliJ. Assuming you still haven't created any project in IntelliJ, we will see the welcome window when IntelliJ is opened. To do anything in IntelliJ, we need to use a project; so, create one by clicking on the option **Create New Project** available in the **Quick Start** panel. In the **New Project** window that will appear, you will see a list of the possible types of projects. This list may be different depending on the plugins that are enabled or disabled; for example, if you disable the Android plugin, no option of the Android project will be available. As we will only work with database in this part, we use the **Empty Project** option; so, select this option and type `database` in the project name input field. In the project location, use any path you prefer and then click on the **Finish** button.

After you click on the **Finish** button, a new window will appear to help you configure the environment of the project; at this time, nothing is necessary to be configured, so close this window by clicking on the **Cancel** button. Finally, the main window of IntelliJ is opened and now you can select the database tool available in the right tool tab pane. If the database tool isn't available in the tool tab pane, go to **View | Tool Windows | Database** in the main menu and open it. The following screenshot shows the **Database** tool:
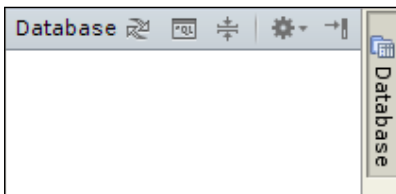
The first impression I had when I saw this tool like it is now was: this tool is totally irrelevant. After some time, I noticed it was a really wrong impression. Looking at the five buttons in the title bar, we can't see anything that might help us use this tool; there is just the **Synchronize** button, the **Console** button, the **Collapse** button, and the other two buttons—the Hide button and the Gear button—and they are common in all tool windows. So, you are now probably asking yourself, "how do I use this tool?"—context menus are the answer.

To create a data source connection, right-click anywhere in the content panel of the database tool and go to **New | Data Source;** a new window will appear asking you for information about the data source connection. In the **Data source name** text field, insert the text `contacts`. At the right-hand side of this text field, you will see a combobox named scope; by default, the selected option is **Project** which means that this data source is available only in this project as opposed to using this data source in other projects and changing it to the **IDE** option. This is a useful option when we need to share the same database with multiple projects.

Below the **Data source name** field, you will see some tabs; in the first tab, we will configure the connection; so in the **JDBC driver files** combobox, click on the **MySQL Connector/J-5.1.22 (download)** option. A message asking for download will appear; click on the **Click here** link to download the files needed, select a directory to save the files, and click on **OK**. After the download of the files, the **JDBC driver class:** combobox will be selected with the correct driver.

If you are in doubt about how to write the jdbc URL, click on the arrow in the combobox and you will see some templates; you can select one of them and substitute the values but, in our case, just type in the database URL text field the value `jdbc:mysql://localhost/contacts`. Enter the username `(commonly: root)` and the password in the corresponding text fields and click on the **Test Connection** button to see if the connection is correctly configured.

If everything looks okay, open the **Schemas & Tables** tab. On this tab, you can see all the schemas available in the data source and select the ones you want to work with. We want to use only one of them to search for tables, so mark the check boxes **Scan for Tables** and **Make Default** for the schema `contacts`.

Now, the tool needs to understand the database dialect it is working with; this information will enable autocomplete and syntax correction when you are typing SQL commands. Open the **Console** tab and select **MySQL** as the default SQL dialect. Now you can click on the **OK** button and see that the new data source is available in the Database tool. The following screenshot shows the **Data Source Properties** window:
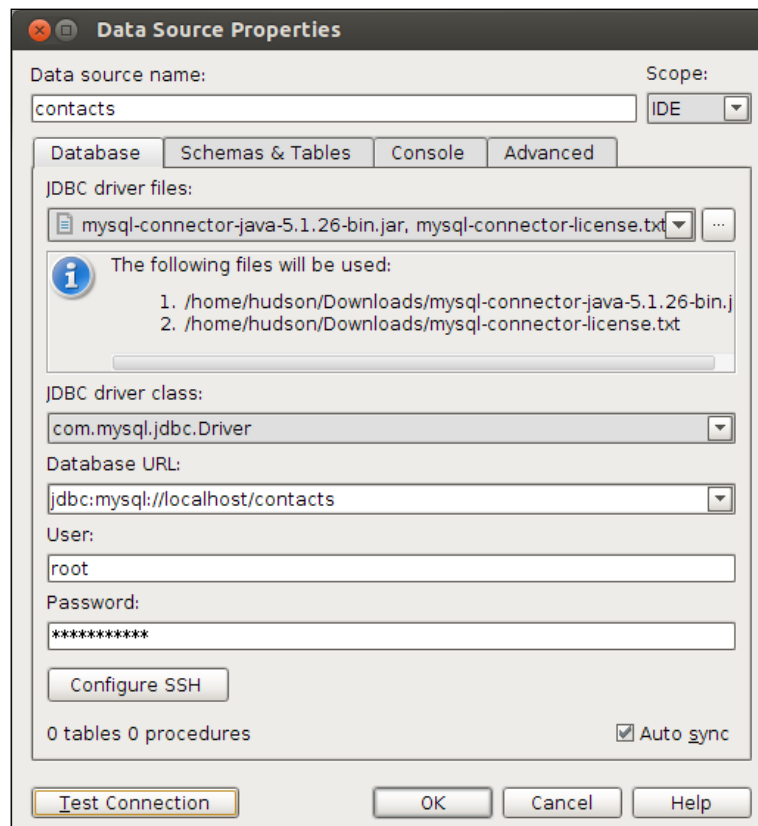
Creating tables

As you know, our database doesn't have any tables; this is the reason you won't see anything beyond the schema and the status of the connection when you expand the `contacts` root node. To create a table, right-click on the `contacts` schema node and select **New** and, after this, click on the **Table** option. The **Create New Table** window will open. The name of our table will be `person`, so type it in the table's text field.

This table will have only four columns: `id`, `complete_name`, `birth_date`, and `place_of_birth`, and we need to create them now. To create a column, just click on the **+** icon and, once the column is added, a pane will be visible to set the properties for the column. The following are the properties you should enter in each column:

- `id`: Its data type is `INT`—don't forget to ensure that the **primary key** option is present
- `complete_name`: Its data type is `varchar (250)`
- `birth_date`: Its data type is `date`
- `place_of_birth`: Its data type is `varchar (250)`

Once you've created all the columns, click on the **OK** button to create the table and you will see the table in the Database tool. The following screenshot shows the dialog used to create `person` table:
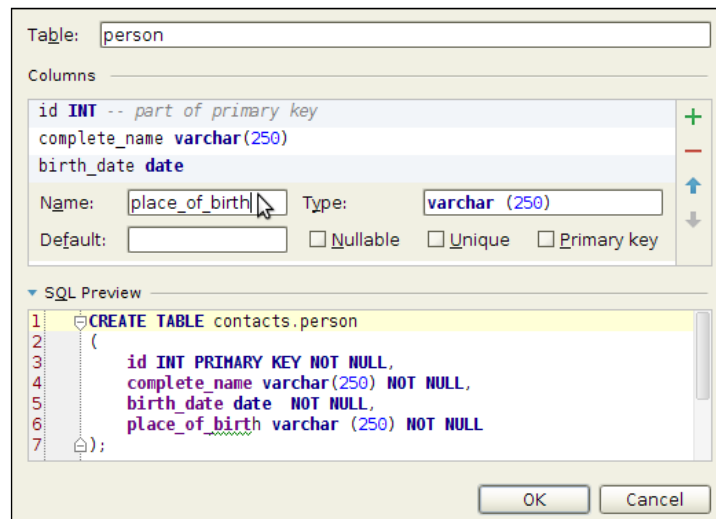
We are going create two other tables but, this time we will use the query console. To open the console, just click on the **Console** button available in the title bar of the tool. When the query console is opened, insert the following SQL query:

```
CREATE TABLE email (
  id int(11) NOT NULL,
  email varchar(256) NOT NULL,
  PRIMARY KEY (id)
);
```

To execute the sentence, click on the **Execute** button located at the top-left corner of the query editor or simply use the *Ctrl + Enter* shortcut (*Command* key + *Enter* in Mac). If everything is okay, you will see the table e-mail in the database tool. Now let's create the phone table:

```
CREATE TABLE phone (
  id int(11) NOT NULL,
  number varchar(13) NOT NULL,

  PRIMARY KEY (id)
);
```

Type this query in the query console and execute it. Now, we have three tables in our little project, but they don't have any relationship between them; it's time to create these relationships. In this example, we assume that a person can have more than one e-mail account and more than one phone. We also assume that an e-mail or phone will correspond to only one person; this way, we have two 1:n relationships: one between person and e-mail, and another between person and phone. So, it's time to create these relationships.

To create these relationships, we need to create a foreign key in phone and in e-mail. Right-click on the table **e-mail**, click on **New,** and then click on **Column**. In the **Add new column** dialog, enter person_id as the name of the column and int as the type, then click on the **OK** button. Now do the same for the phone table.

After doing this, you will see the person_id column in the e-mail and phone tables; however, we haven't created a foreign key yet. To do this, right-click on the **person_id** column in the phone table, click on **New**, and then click on **Foreign key**. In the **Add foreign key** window, you need to specify the referenced table; in our case, the table is person. Note that, while you are typing, IntelliJ will suggest some options based on the information in our database. Click on **OK** when you are finished and do the same thing for the e-mail table. If everything is executed well, you will see a light blue key on the icon of the person_id column. Finally, we have a database with the referenced column.

If you are attentive, you will notice that none of the `id` columns use the `auto_increment` approach to generate the id automatically. Looking at the visual tools, I didn't find a way to add `auto_increment` in a column, so, we will do it using the query console:

```
ALTER TABLE email MODIFY COLUMN id INT NOT NULL AUTO_INCREMENT;
ALTER TABLE person MODIFY COLUMN id INT NOT NULL AUTO_INCREMENT;
ALTER TABLE phone MODIFY COLUMN id INT NOT NULL AUTO_INCREMENT;
```

If you just position the cursor anywhere in the console, it will execute only the query that the cursor is hovering over. A better approach is to select all the sentences and click on the **Run** button; this way, all the selected sentences will be executed in the order they are selected.
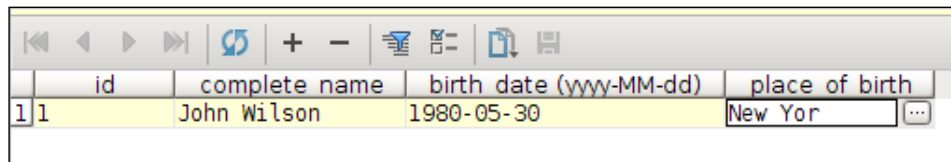
Now that everything looks good, let see the diagram of the database so we can identify the relationships and characteristics of each table better. When creating a diagram, IntelliJ only uses the table you've selected and any other table related to it so, if you select only the phone table, the diagram will show just the `phone` and the `person` tables. If you want to see all the tables, go to the Database tool and select all the three tables; you can do this as you do in your OS, for example, holding *Shift* and selecting the first and last tables. Once you've selected the tables, right-click on the selection and choose **Diagrams** and select one of the two subsequent options: **Show visualization** or **Show visualization popup**. The **Show visualization popup** will only show the diagram in a pop up, without any other option. The **Show visualization** option will show the diagram and some options available on the toolbar, such as the button to show or hide the columns and refactor options. The following diagram shows the database:
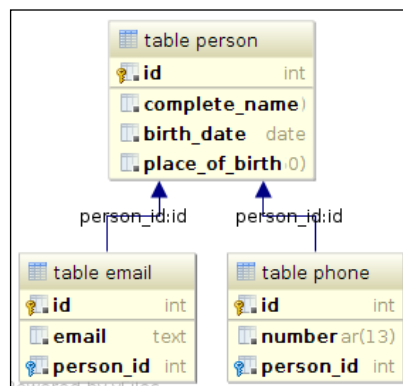
Manipulating data

Now that we have a database and the tables correctly configured, what we need at this time is to populate our tables. If you are working with a low quantity of data, a visual approach could probably be better than using SQL sentences, so let's insert some data in the tables using **Table editor**. Just right-click on the **person** table and select the **Table editor** option. When **Table editor** appears, you will see a new tab with options to manage the data in the table and, at the bottom of it, you will see two other tabs: **Data** and **Text**. The **Data** tab is used to manage the table using visual controls such as buttons and tables. The **Text** tab is used like a query console, however, with fewer options.

We will add a contact using the visual approach, so make sure the **Data** tab is active and click on the **Add new row** button (*Alt + Insert*). When you insert a row, the first column of the new row is focused; in our case it is the id column in the person table. You could try to force this column to lose focus using the mouse; however, you won't have success with it. To lose focus in this cell, you will need to insert a value (for example, null) or simply press the *Esc* key. As we want to use auto_number of this column, make this cell assume a null value. Focus on the complete_name cell, insert the text John Wilson, and press *Enter* (or *Tab*) to go to the birth_date field. If you haven't noticed, in each cell you observe a button with **...** available. Depending on the type of the column, different things will appear when you click on this button; in the case of the date type a calendar will show. Click on that button and select the date **1980- 05-30** then press *Tab*, insert New York, NY in the last column, and press *Enter*. In the following screenshot, you can see the table editor in action:

As we have lot of things to do, you can use the files available in this book to load the data in our database. Click on the File menu and choose Open...; in the dialog that appears, select the file email.sql and click on the OK button. The file will be opened in a new tab. You will notice that at the top of the tab is a yellow bar asking about the SQL dialect; click on the link at the right called Change dialect to.... In the **SQL Dialects** window, select the line project and click on the cell in the column **SQL Dialect**. A list of available dialects will appear, so, select **MySQL** and press the **OK** button.

You've probably noticed that this file wasn't opened in the query console and that there isn't any button available to execute the script. Differently from the query console, when you open a SQL script directly with IntelliJ, you don't need to put the cursor over a line you want to execute; the whole script will be executed, and that is what we want. To execute it, right-click anywhere in the script and select **Run "email.sql"**; now, if you use the **Refresh** button in the table editor in the person table, you will see that the data was inserted into the table. Now you can do the same for `phone.sql` and `person.sql` files.

Once the data is loaded in all the tables, we can work with them; so, if it isn't open, open the person table in the table editor. As you can see, some data is loaded by default, but this is limited to 52 records—you can see this when you click on the **Query** button. Despite being a tool that will sometimes help you, the table editor doesn't permit changes in the query used to retrieve data and doesn't have some visual facilities that are available in tools like Microsoft Access; so, if you want to see two or more tables together, you need to write the query in a query console.

We will now create a simple database view that permits us to see the data from the person table joined with the data from the e-mail table; so, open the query console and type the following query:

```
create view person_and_email as
   select person_id,email.id as
      email_id,
      complete_name,
      birth_date,
      place_of_birth,
      email
   from person
left join email on person.id=email.person_id
```

After inserting the query, click on the **Execute** button (or use *Ctrl + Enter*); you will see a new entry in the database tool with an icon with a magnifier named `person_and_ email`—right-click on it and select the **View editor** option. As you probably know, view editor won't let you change the data as it is only a view, but you can use filters to limit the results. This time, we want to see only the rows in which the name of the person starts with John ordered by email address and we don't want to see any ID, so click on the **Filter** button to open the **Filter** window. In the **Filter** window, uncheck the `person_id` and `email_id` check boxes, then in the **Order** column in the **email** row, select **1 asc**. Finally, at the filter column of **complete_name** row, insert the sentence **like 'John%',** as shown in the following screenshot:
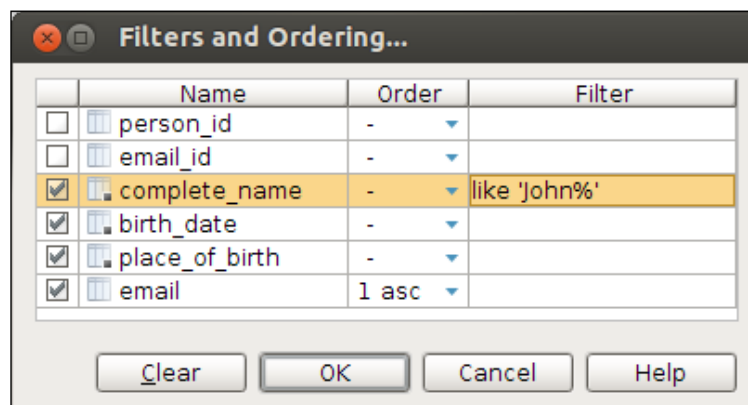
The following image shows data presented in view editor:
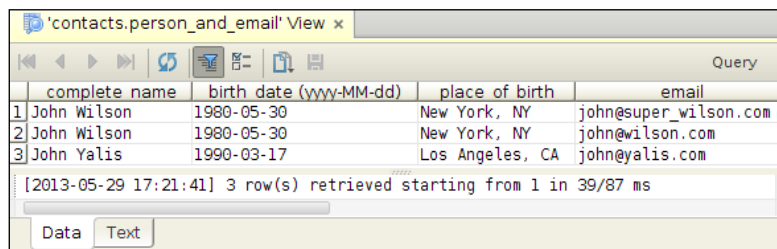
If we want to send this data to someone using another format, we need to export it. At the toolbar in table view exists a button called **Copy data as**; when you click on it, a menu will be shown in this menu. Select the option **Save all to file** and, finally, select **HTML Table**. In the **Save** dialog, select any location you want and a name for the file; you can now open it in your favorite browser and see that the data is correctly inserted in the HTML table.

[52]

As you've seen, you can export the data to various formats, from comma-separated values to the JSON format, allowing you to decide which one will best meet your needs. You can open these kinds of tools in IntelliJ too, for example, if you open a CSV file in the IDE, you will see a text and a **Data** tab in the bottom and clicking on the **Data** tab will permit you to see the data in the file in a tabular format. I should remind you that only the SQL formats can be used to load data using IntelliJ; so, if you want to export the data to load it using the IDE, only export to SQL formats.

ORM Support

ORM (**Object/Relational Mapping**) is a technique used to provide an object approach in a relational structure; this way, the developer's work is facilitated when he/she is working with the database in his project. In Java, a specification called **JPA** (**Java Persistence API**) defines the way that ORM frameworks should work. To facilitate even more the way the developer uses ORM frameworks, IntelliJ provides a lot of support to ORM frameworks like Hibernate and EclipseLink.

To use Hibernate or JPA, we need to activate the persistence tool tab. You may imagine that you just need to go to **View** | **Tool windows** | **Persistence** to activate the persistence tool tab, but you won't find the persistence tool available there. Some tool tabs are only available when you add a corresponding facet to the project; you need to add a facet to enable this tool. As this project was created only to create the tables in the contacts database, it isn't a good approach to add the Java code directly into it. So, instead of creating another project, let's create a new module that will be a child in this project; this way, we continue in the same project without being concerned about the organization of the project become confusing.

Go to the **File** menu and select **New module**; the new module window will appear to permit us to select the kind of module we want. Select the option **Java module**, insert `entities` as the name of the module, and press the **Next** button. The next frame of the wizard will ask which facets you want include in your project and, if necessary, it will download the libraries for you. We will use Hibernate instead of JPA in this project, so check the **Hibernate facet** option (in case you need to use JPA, select the **JavaEE Persistence** facet); when you do that, the panel at the right will show you some options you can use to configure the facet in the project.

We won't configure anything here, so just click on the **Finish** button. After the download of the libraries and some processing, you will see the module entities in the **Project** window. As you can see, the module just has the `lib` folder with the dependencies and an empty `src` folder. However, now you can see the persistence tool tab just because this module has the **Hibernate facet** option. The following screenshot shows the configuring of Hibernate facet:
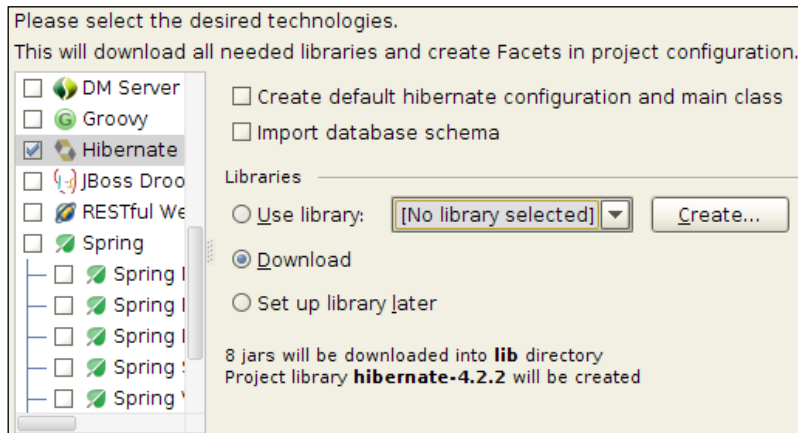
Creating database entities

Click and open the persistence tool; there you will see Hibernate's logo followed by the entities text, and you should right-click on it. In the context menu, select the last option **Generate persistence mapping**, and then click on **By database schema**. A new window will appear to us to generate the database entities. In the **Choose data source** combobox, select `contacts`. As we haven't created any package yet, we can't select anything in the **Package** combobox; however, we can enter the package name we want here and the tool will create it for us, so type `org.example.entities` in this field. There are two other fields available in the window: **Entity prefix** and **Entity suffix**. They are used in case you want to insert something at the start or end of the entity name. In our case, we will insert in the **Entity prefix** field the value `Entity`.

In the **Database Schema Mapping** section, you will see a table with the entities, the tool retrieved from the data source, and some buttons that you can use to edit the relationship between these entities. At the leftmost side of the table exist some colored marks. They are used to inform you that the respective entity isn't already referenced, as you can see in the legend below the table. You can expand the nodes to see if everything is okay and to change type of the properties. In the following screenshot, you can see the **Database Schema Mapping** section:
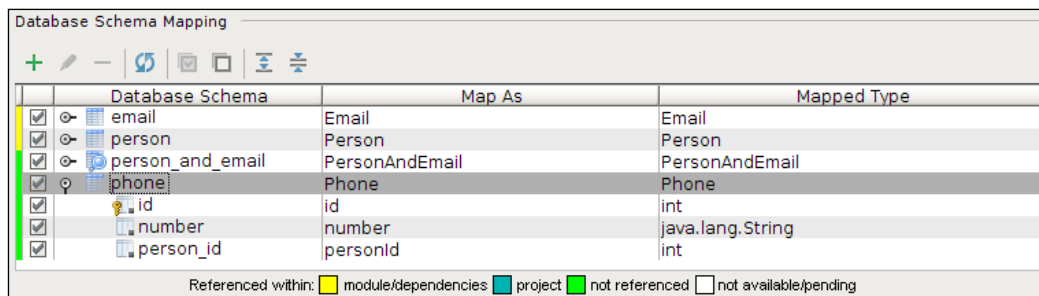

Finally, in the **Generation settings** section, you will see options that directly affect how the entities will be generated; as you can see, you can define if the entities will use JPA specifications, be generated in XML files, and more. As we won't change anything right now, just click on the **OK** button to generate the entities and we get an error. We didn't select any persistence unit and still can't see where we can do this. This is common in this window; it shows an error message and is sometimes confusing. To solve this problem, make sure that the **Add to session factory** option is checked and click on the button with a **+** icon at the right of the related combobox. A new window will appear asking you to choose or create a Hibernate configuration file—let's create it. Click on the button **New file** (the first active button at the top), enter the name of the configuration file `hibernate.cfg`, and click on **OK**. Now select the created file and click on **OK** again. Back at the **Database Schema Mapping** window, the error message immediately is removed and we can click on the **OK** button to generate the entities. Click on **Yes** in the confirm dialog and, after some processing, the entities will be ready for use.

Open the **Project** window and the entity `EntityPerson.java`. You will see that, in this file, some names retrieved from the tables are underlined in red; it means IntelliJ can't identify the fields. To solve this, position the cursor in one of these problematic fields and use the *Alt + Enter* shortcut; the first option available is **Assign data source**—select this and press the *Enter* key. In the **Assign data sources** window, you will see the configuration file we created in the **Session Factory** column and **<none>** in the **Data Source** column; click on the **<none>** entry, select **contacts** and then click on the **OK** button. As you can see, all the fields underlined in red vanish.

Problems that can occur

During the time I have been using IntelliJ, in some rare situations, a simple feature didn't work as I had expected and only one feature forced me to do the job using another IDE: generating the entities from a database. You are probably asking yourself now: "why can't you create the entities if you have already done so in the example and it is so simple?" You're correct, it is really simple; however, IntelliJ isn't free from bugs and one of the bugs it has drastically affects the mapping of entities when we are working with Microsoft SQL Server or Oracle database.

When you create a column in Microsoft SQL Server or in Oracle database with the type `NVARCHAR`, IntelliJ can't identify these types of data as `java.lang.String`, and instead it will identify as `java.lang.Object`. If you click on the **OK** button to generate the entities, it will not work because the properties shouldn't be of type `java.lang.Object`. You can solve this problem in three ways: manually modify the type of the property, change all of your columns to `VARCHAR`, or change your database. These solutions are really simple for small databases with less than 20 tables; however, the database of the project I was developing had more than 100 tables, so it was easier for me to use Netbeans just to generate the entities. You can see more about this issue at `http://youtrack.jetbrains.com/issue/IDEA-79437`. The following image shows IntelliJ can't identify `NVARCHAR` as `java.lang.String`:
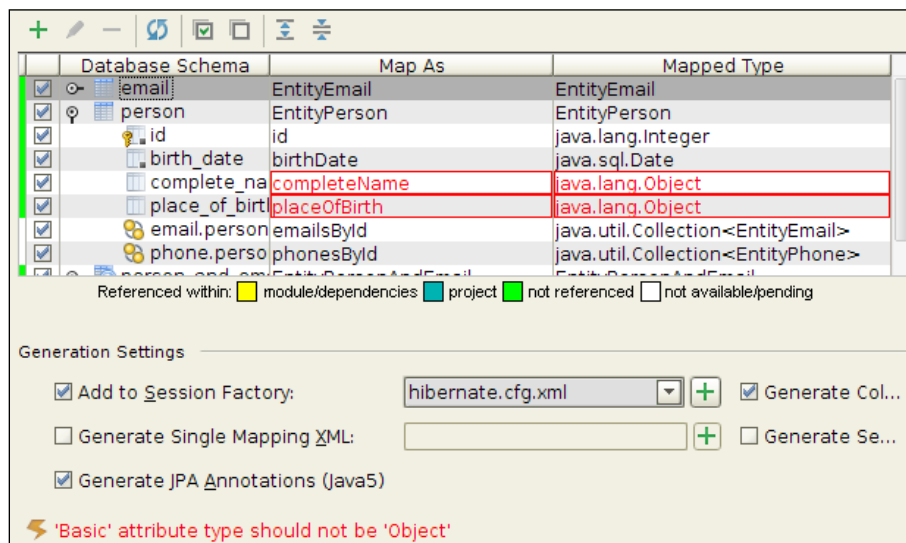
Summary

As you can see, the database tools available in IntelliJ are easy to use and, in most cases, will answer your needs. Of course, there are tools on the market that can do a similar and perhaps better job when working with different kinds of databases. However, the focus of the database tools in this IDE are to help the developer integrate his/her code with the database. The other tools probably won't provide a tight integration between the database and the Java code as you've already seen, and will see again in the next chapters.

4
Web Development

In this chapter we will develop together a simple web application. This will give us the chance to understand and observe some tricks that could be useful for us when we're developing applications using IntelliJ. The application simply tries to pass through some situations the developer goes through when he/she is developing an application.

Creating a web module

Web applications are the most used kind of application; they could be a small and simple website or a complete and powerful system. When we are creating a web project, we generally create a new project and add the other necessary modules to that project as needed. In this chapter we won't do this; instead, we will re-use the project we created in the last chapter; this way, you can see how flexible IntelliJ is compared to the project structure. As we are going to re-use the last project, open IntelliJ if it isn't already open.

Once you open the IDE, you will see that the last project is loaded in IntelliJ. To create a new module, we can proceed like we did before: go to **File** | **New Module...**; in the new module window, select the option **JavaEE Web Module** and change the module name to `contacts-web`. When you click on the **Finish** button, the new module will be positioned on top of the modules of the project.

If you expand the nodes of the new module, you will see the default structure of the web module with an empty `src` folder, a `web` folder with a blank `index.jsp` page, and `web.xml` file as leaf nodes. Let's see the properties of the new module using the **Project Structure** window (C*trl* + *Alt* + *Shift* + *S*). Everything looks okay until we get to the **Facets** option and select the web module we created. At the bottom of the window, you will see a message saying **'Web' Facet resources are not included in an artifact**. Fortunately, this little problem can be solved by just clicking on the button **Create Artifact** to the right of the message. Now you will see that a new artifact is available for our web project.

An artifact can be understood as the specification of the input of the project and as the output too; it can be a simple JAR file or other kinds of outputs (WAR or EAR) depending on the facets that were set in the module. In the **Artifacts** option, you will see how the module is configured to generate the outputs. As we won't change anything here, you can click on the **OK** button to save the changes.

Now that our project is correctly configured, we can build it by just going to **Build | Make Project** (*Ctrl + F9*); and if you go to the out directory in the project root, you won't find any WAR file. This happens because the module is configured to generate an exploded artifact, so IntelliJ will only compile the existing Java files in the project and won't package it.

If you want to generate the WAR file, open the **Project Structure** window and, in the **Type:** combobox, select the desired format (**Web Application: Archive**). Notice that in the **Output Layout** tab, a new option will appear asking about the `MANIFEST.MF` file; you can click on the **Create Manifest...** button and select the `web` folder as the location for the file. When you click on the **OK** button, the option in the **Output Layout** tab will direct you to another panel you can use to configure the `MANIFEST.MF` file; when you are done, click on **OK** to save your changes.

Finally, to build you artifact, you can go to the **Build | Build Artifacts** menu and you will see a folder called `artifacts` in the `out` directory along with the WAR file.

Configuring the application server

The great majority of the time, you will only generate a WAR file when the project is ready for production. Other times, we will use the exploded version running in a local application server so we can test the application in a faster way. It's time to configure and run a web application using IntelliJ.

If you haven't already noticed, in the main toolbar there is an empty combobox beside the **Make Project** button (as you can see in the following figure). This combobox permits the selection of the configuration you want run. The configuration can be of different kinds, from a simple Java application to Maven and application servers. To run our web application, we need to create a new configuration that enables the execution of an application server, as shown in the following screenshot:
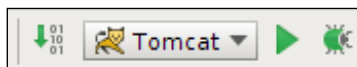
In this example, we will use a Java servlet container called Apache Tomcat 7 (`http://tomcat.apache.org/download-70.cgi`); so, if you haven't already installed it, do it now. As you can imagine, the installation and configuration of Apache Tomcat is outside of the scope of this book. The installation is really simple and you can find good directions on how to do it at the Apache website `http://tomcat.apache.org/tomcat-7.0-doc/setup.html` and over the web.

After you've installed the Apache Tomcat server, we can create a configuration to use it. Click on the **Select Run/Debug Configuration** combobox and select the option **Edit Configurations...**. After you do that, a new window called **Run/Debug Configuration** will be shown. In this window, click on the **Add** button and select the **Tomcat** option. At least in my case, I couldn't see any option called **Tomcat** or **Apache Tomcat**. If, like me, you can't find this option, close this window right now.

This kind of problem generally happens when something in IntelliJ is disabled, in this case, the Tomcat plugin. Open the configuration **Settings** window (*Ctrl + Alt + S*) and, in the search textfield, type `tomcat`. You will see that only two options will be available; click on the **Plugins** option and you will see that the plugins are filtered by the search content. Check the plugin **Tomcat and TomEE Integration**. Click on the **OK** button and IntelliJ will ask if you want to restart the application—do it.

Now that the Tomcat plugin is enabled, we don't have excuses for not yet having found the Tomcat option. Open the **Run/Debug Configuration** window and expand the **Default** node. We don't need to configure anything at the default node to run Tomcat or any other configuration; however, when we configure the elements in the default node, we reduce the time that would be spent in other projects that need similar configurations. For example, let's suppose you're using the same server in four different projects and, on a beautiful day, you decided to change the port that Tomcat was running from `8080` to `80`. If you configured the default node, you need to change it just once. The same occurs in the other configurations available here, such as TestNG.

To configure the default configuration of Tomcat, expand the **Tomcat Server** node and click on the **Local** option. To the right-hand side of the window, the panel will show some tabs, and in the **Server** tab we can, configure the Tomcat Sever. In the **Application Server** combobox, we need to select which installation of Tomcat we want to use; however, at this time, nothing is available, so click on the button **Configure...** which is positioned to the right-hand side of this combobox. A new window will appear asking about the path where Tomcat is installed so, in the first input field, enter the name of the path where you've installed Tomcat. If IntelliJ recognizes Tomcat in the selected directory, it will fill in the other input field and detect the Tomcat version. If everything is fine, click on the **OK** button. The following screenshot shows how the path of the Tomcat server is defined; note that this configuration may vary depending on the place where you've installed Tomcat.
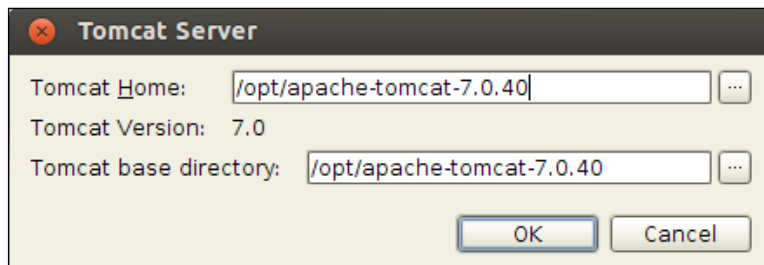
At least for now, there isn't anything more we want to set in the default configuration of Tomcat, so click on the **Apply** button to save the changes. Even after modifying the default Tomcat configuration, we still need to add a specific configuration for this project; so, click on the **Add New Configuration** (*Alt + Insert*) button, select **Tomcat Server**, and then click on **Local**. A new node called **Unnamed** will be created inside the **Tomcat Server** node and the panel to the right will show similar options as does the default configuration. At the top, there is a text field called **Name:** with **Unnamed** as its value (as you can see it in the screenshot below)—change this to `Tomcat`.
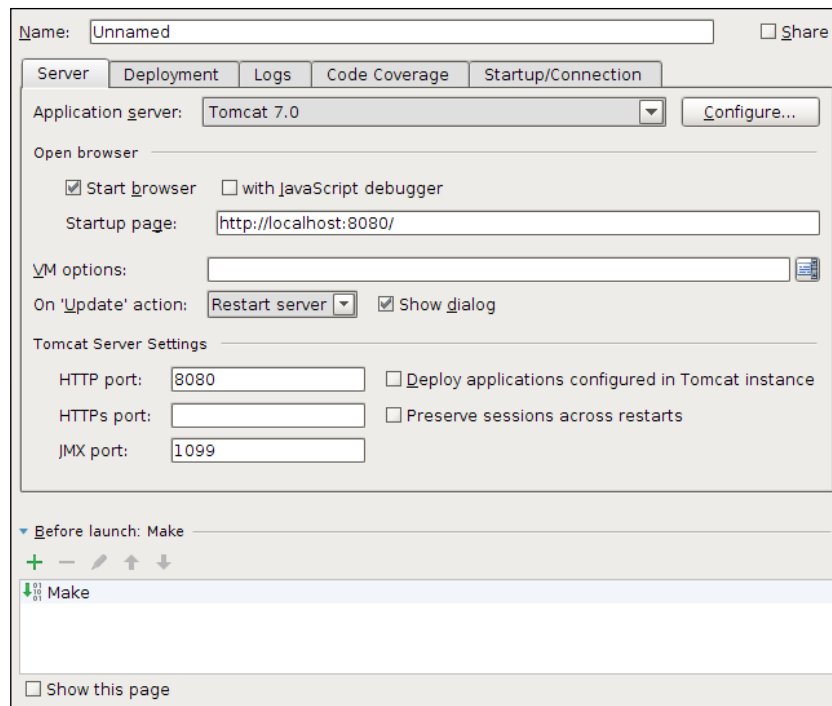
The great majority of the time, IntelliJ is capable of identifying problems and proposing solutions. One important thing you should never forget to define when adding the configuration of an application server is the artifact that you want to deploy. As we haven't declared this, IntelliJ shows a message at the bottom of the window indicating the problem and providing a button to solve the problem. Click on the **Fix** button and you will see that IntelliJ will configure the war artifact we created to be deployed in Tomcat. As we don't want to change anything more here, click on the **OK** button to save and close the changes. The following screenshot shows that IntelliJ knows something is missing. Now, the Tomcat configuration is selected in the **Select Run/Debug Configuration** combobox and a
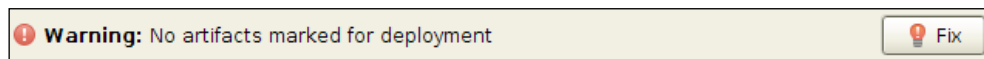
Now that we have the application server configured, let's run our application to see if everything is alright. If it is not selected, select **Tomcat** in the **Select Run/ Debug Configuration** combobox, then press the **Run** button (*Shift + F10*) to start the application server and deploy our application. After making the project, IntelliJ will open a new tool window called **Run**; this tool is used to monitor the execution of the module we selected to run, which in our case is the contacts-web module running on Tomcat. At this moment, as our project doesn't have any content on the JSP page, a blank page will be shown in your default web browser.

Developing our application

To help you understand how to use the facilities in IntelliJ and identify its tricks, we will develop a simple **Java Server Faces** (**JSF**)-based application in the course of this chapter. JSF is an MVC framework used to construct web interfaces using components; this technology evolved from the experiences the community had with JSP/servlet and Struts. The application will simply expose the data we stored in the contacts database and provide a web service that permits the creation of a new contact.

Configuring the JSF environment

First of all, we need to make our `contacts-web` module compatible with JSF; the smart reader you are, you probably know that we will add the JSF facet to our module now.

Open the **Project Structure** window and select the option **Modules**. Then, expand the **contacts-web** node, select **Web** and click on the **Add** button (*Alt + Insert*). When you select JSF, you will see that the panel at the right changes and permits you to select if facelets will be supported; select **enabled**. (Facelets is the default view declaration since JSF version 2.0.). At the bottom of this panel, a warning message about the missing libraries in JSF will be shown; click on the **Fix** button to select which JSF library you want to use. In the new window that will appear, select the option **Download** and click on the **Configure** button. As you can see in the new window, we can select which version of JSF we want to download and use. We will indicate that we want to use JSF 2.0 and then just close this window. Click on the **OK** button to start the download of the libraries.

After the download, a red message will appear at the bottom telling us that the library is missing in the artifact; just click on the red lamp and select the option **Add 'jsf-2.0.1-FCS' to the artifact** to solve the problem. As we don't want to create all the functionalities using default JSF tags or implement all the behavior, we will use Primefaces to ease our job. (Primefaces is a component suite for JSF, `http://primefaces.org/`). Fortunately, IntelliJ provides some useful facets to use with the JSF facet. Select the **JSF** node, click on the **Add** button, and then select **Primefaces**; the new window that will appear will ask us which version of Primefaces we will use—just check the **Download** radio and click on **OK**.

Again, a message will appear telling us that the library is missing in the artifact. Click on the icon with the red lamp and select the first choice to fix the problem. Finally, click on the **OK** button to save our changes and close the project structure window. Now our project is configured for JSF and a new tool window called JSF is available to the right of the IDE.

As we won't use JSP pages, delete the file `index.jsp` that is inside the `web` folder of `contacts-web` module. After this is done, copy the file `web.xml` that is available in the `chapter 4` folder in this book and override the existing `web.xml` that is in the `web/WEB-INF` folder of the module. This new `web.xml` file configures the application to use the Faces servlet, XHTML as the default suffix, and `home.xhtml` as the default page.

As you may have noticed, we still don't have the `home.xhtml` file; to create it, right-click on the **web** folder and go to **New** | **JavaEE Web Page**. A new window will appear asking for the name of the file and the type. Name the file `home.xhtml` and in the **Kind** combobox, select the option **Facelets Page**. IntelliJ will create and open the `home.xhtml` file with the basic structure of the page. Just to see if the application is working correctly, change the text in the title tag to `The contacts` and add `h:outputText` to the `body` tag with `Hello contacts` as the value, as follows:

```
<h:head>
<title>The contacts</title>
</h:head>
<h:body>
<h:outputText value="Hello contacts"/>
</h:body>
```

Click on the **Run** button to execute the application and IntelliJ will automatically open your default web browser where you will see the page of the application.

Resolving the dependencies

Our application will just show the user the contacts that are stored in the database. To add new contacts, we will provide a web service that will be covered in the next chapter. Despite being a very simple application, we need to do some work to get things working. As it could be confusing to perform a step-by-step process from the actual state of the application, I will provide you some code to jump-start this part.

In the `Chapter 4` folder of the book, you will see two folders. The first is called `contacts-web` and comprises two other folders. Despite it being possible to do all the following steps in IntelliJ, it is easier and less error-prone if you do it on your own operation system. Copy the folder called `java` to the `src` folder in your `contacts-web` module folder. After doing this, replace the contents of the folder `web` that exists in the root folder of this module. When your OS asks, override all files. Open the `chapter 4` folder again; there is a folder called `entities`; copy the `src` folder inside it to the `entities` module folder and override the content when asked. As we won't use the view `EntityPersonAndEmail`, navigate to the `src/org/example/entities` folder in the entities module and delete it.

Back at IntelliJ, you will see that a balloon will appear telling you that IntelliJ has detected the Spring framework; click on the **Configure** link and, at the new window that will appear, just click on **OK**. Every time IntelliJ detects a framework or something it knows how to work it will show you a message, sometimes ask you to perform an action, and finally add the corresponding facet in the project.

Theoretically, when you click on the **Run** button, mark the option **Redeploy** and click on **OK**—everything should work; however, some errors will appear in the messages tool window. The messages the IDE shows us say that it can't find the classes of the module entities. This happens because we didn't add entities to the `contatcts-web` folder which is what we will do now. The following screenshot shows error messages:
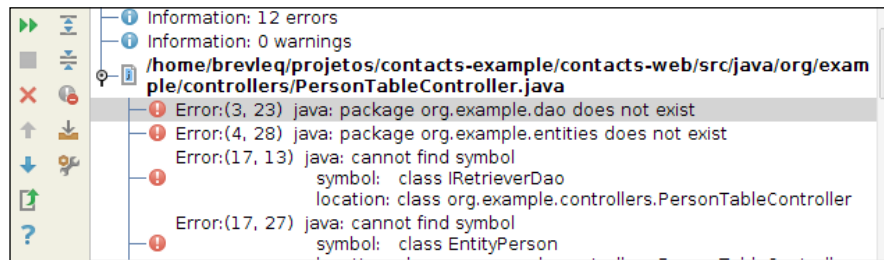
Position the cursor over any word in red and press *Ctrl + Enter*; this will show the intentions menu. Select the first option **Add dependency on module 'entities'** and, after doing this, you will see that all the red words become black again. This indicates that IntelliJ didn't find any problems in your code. Even if you try to run the module again, you will still receive error messages. This time it will be because some external libraries are missing.

Open the **Project Structure** window again and you will see a message telling you that the module `entities` is missing in the `contacts-web` module; click on the red light bulb and select the first option. After this, you will see a message asking about the Hibernate library; do that, click on the red lamp icon, and select the first option. Now select the facets option and then the end node **Spring (contacts-web)**. You will see that IntelliJ can't find the Spring library in the module; click on the **Fix** button, then click on **Download**, and then click on **OK** to solve it. When the new red message appears, click on the red lamp icon and select **Add 'Spring-3.2.2' to the artifact**.
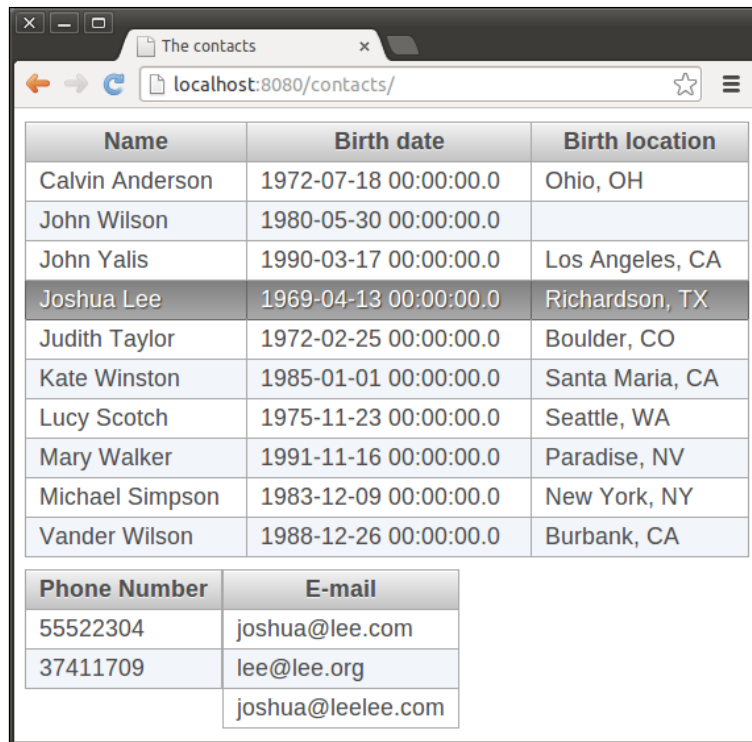
Unfortunately, IntelliJ won't detect that some libraries are missing so, in these cases, we need to identify which library is missing and provide it to the module. This time what is missing is the Spring Web library; an easy way to find it is by adding the Spring MVC facet to the module `contacts-web`. Select the **Modules** option in the **Project Structure** window, then select the contacts-web module, click on the **add** button (*Alt + Insert*) and, finally, select the **Spring MVC** option. You will probably need to download the library, so do it. Again, a red message will be shown telling you that the library is missing from the artifact; click on the red lamp and select the first option.

Now we need to add the MySQL JDBC driver to our project. As we have already downloaded MySQL Connector in *Chapter 3*, *Working with Databases*, we just need to add it to the project. Select the option **Libraries** in the **Project Structure** window, click on **New Project Library** (*Alt + Insert*), and select **Java**. At the **Select Library Files** window, navigate through the tree and select the file `mysql-connector-java-x.x.xx-bin.jar`, then click on the **OK** button. Another window will ask us which module will receive the library; select `contacts-web` and click on **OK**. By now, I don't think I need to tell you that a red message will appear; click on the red lamp and select the first option.

The last library that is missing is the AspectJ weaver; in this project, AspectJ is used by Spring to manage the database transactions. You will find an AspectJ facet in the **Project Structure** window; however, you will see that it won't download any libraries. Despite this not being a Maven project, we can use the power of this tool in IntelliJ. Click on **New Project Library** (*Alt + Insert*) again and, this time, select the **From Maven…** option. In the new window that will appear, type `org.aspectj.weaver` in the first input field, then click on the **Search** button (*Shift + Enter*). After some searching, you will see that it found one entry; click on the down arrow at the end of the input field and select the option **org.aspectj:com.springsource.org.aspectj.weaver:1.6.2.RELEASE**. Now, just click on **OK** to download the file using Maven. After the download ends, select the `contacts-web` module in the next dialog and fix the problem by clicking on the red lamp. Finally, we can click on **OK** to save our changes.

Before we finally run the application, we need do a simple task. We will use Spring **AOP** (**Aspect Oriented Programming**) to manage the interfaces that will use Spring's transaction manager, as it is a good practice to develop code based in the interfaces we will use them. As no interface exists in `org.example.dao`, we will extract an interface from `RetrieverDao`; so, open this class file and go to **Refactor** | **Extract** | **Interface...**in the main menu. A new window will enable you to configure the new interface; in the **Interface Name** field, enter `IRetrieverDao`, check all members, and click on the **Refactor** button. After refactoring, IntelliJ will ask if you want to substitute all usages of `RetrieverDao` in the source code, so you need to confirm whether or not you do. As the preview option is checked in that window, we need to open the find tool window to see the changes before refactoring them; after looking at the changes, click on the **Do Refactor** button to proceed.

You can now press the **Run** button to see the application running. If you have problems connecting to the database, check these two files: `hibernate.cfg.xml` and `appContext.xml`, and check the connection properties. The following screenshot shows the running application:

Creating the filter code

Now that our application is running well, we will create a filter to search the database for the results that will be shown in the contacts table; it will help you understand some of IntelliJ's features. First of all, add the following piece of code to the `<h:form>` tag:

```
<h:panelGroup layout="block">
<h:outputLabel value="Filter:" for="filterInput" style="margin-
right: 5px;"/>
<p:inputText id="filterInput" value="#{personTableController.
filterValue}"/>
</h:panelGroup>
```

As you may notice, the `filterValue` property doesn't exist in the `PersonTableController` class. Let's create this field with the setter and getter methods using the simplest and fastest way IntelliJ provides—using intentions. As you know, when we position the cursor over some place in the code that presents some possibility of improvement, a small icon representing some intention will appear. So, position the cursor at `#{personTableController.filterValue}` and click on the yellow bulb icon (or simply press *Ctrl+Enter*). In the menu that will appear, select the option **Create getter, setter and field for filterValue** and you will see that a field the with the setter and getter methods will be created in the class.

At this time, the input field can't filter anything because it doesn't have the necessary code to do the filter. To do this, change the code `filterInput` as follows:

```
<p:inputText id="filterInput" value="#{personTableController.
filterValue}">
<p:ajax event="keyup" listener="#{personTableController.filter}"
update=":form:personTable"/>
</p:inputText>
```

Again, we've used a method that doesn't exist in the class `PersonTableController`; so, position the cursor at `#{personTableController.filter}` and press *Ctrl + Enter*. Again, you will see the three options that were shown earlier; however, the listener method is a simple method, so it won't use any field. As a shortcut, we will select the option **Create getter for filter**. Once you've done that, press *Ctrl + B* to go to the method declaration. Once you are in the method declaration, position the cursor over `getFilter()` sentence and press *Shift + F6* to rename the method as `filter()`. As this method won't return anything, change the string type `return` to `void` and remove the `return` statement; notice that if this method is being used anywhere in the code, we can navigate to **Refactor | Change signature** (*Ctrl + F6*) to change the return type and add parameters to all usages. After the changes, your method should look as follows:

```
public void filter() {
}
```

I don't know about you, but I like to organize my code positioning all getters and setters at the bottom of the class file. We can move the filter method to the top of the class using *Ctrl + Shift + Up*, so use this shortcut and position the method as the second one in the class.

The code still doesn't do any filter, so go to the filter method and change it to look as follows:

```
public void filter() {
  setPersons(retriever.retrieveByFilter(filterValue,1));
}
```

If you run the application, you will see that when you try to filter the contacts you receive `java.lang.UnsupportedOperationException`; this happens because the default implementation has just returned this exception. Let's create a class that implements this method. Expand the tree of entities module until you see the package `dao`; right-click on this node and click on **New** and then click on **Java Class**. The dialog that appears is used to create classes, interfaces, enums, and more; you just need to select the type. In our case, select the class and type in the name field `PersonDao`, then click on the **OK** button.

As we want to implement just the method `retrieveByFilter`, we need to extend the class `RetrieverDao`. Make your class declaration look as follows:

```
public class PersonDao extends RetrieverDao<EntityPerson> {
}
```

You will see that the declaration will be underlined in red. Position the cursor over the class declaration, press *Ctrl + Enter,* and select the first option **Create constructor matching super**, as shown in the following figure:
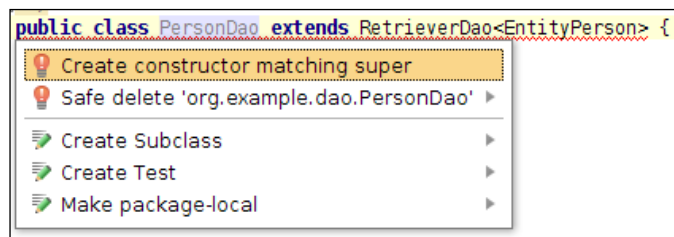
It will create a constructor compatible with the super class but, at this time, we have better knowledge about the entity that will be manipulated, so remove the second constructor parameter and pass `EntityPerson.class` as the second parameter to the super constructor. Your code should now look as follows:

```
public class PersonDao extends RetrieverDao<EntityPerson> {
  protected PersonDao(SessionFactory sessionFactory) {
    super(sessionFactory,EntityPerson.class);
  }
}
```

Finally, we will override the method we wanted; so press *Alt + Insert* (in Mac, use *Ctrl + Enter*) anywhere in the class and select **Override methods...**. In the new window that will be shown, select the method **retrieveByFilter(value:Object, filter:int):List<T>** and click on **OK**. IntelliJ will create a default implementation for this just by calling the super method. As we don't want to use the default implementation, change the implementation of the retrieveByFilter method as follows:

```
 @Override
 public List<EntityPerson> retrieveByFilter(Object value, int filter)
{
   switch (filter){
case 1:return retrieveByName((String)value);
     default:return new ArrayList<EntityPerson>();
   }
}
```

Again, we've used a method that doesn't exist, so position the cursor at the retrieveByName usage, type *Alt + Enter*, and select the unique choice available. It will focus on the return type declaration; as we don't want to change anything, just press the *Esc* key twice. Now change the implementation of retrieveByName as follows:

```
private List<EntityPerson> retrieveByName(String name) {
   Query query = sessionFactory.getCurrentSession()
.createQuery("select e from EntityPerson e where
e.completeName like :name order by e.completeName");
   query.setParameter("name", createSearchParameter(name));
   limitTuplesQuantity(query);
   return query.list();
}
```

You probably noticed that in the switch at retrieveByFilter we are using 1 as the filter value as we know that using a value instead of a constant will impair the readability of the code. Fortunately, IntelliJ provides an easy way to refactor this kind of source. Position the cursor at 1 character present in case:1 sentence, then go to the **Refactor | Extract | Constant** menu (or use the shortcut *Ctrl + Alt + C*) to create a constant. When you do this, the value 1 changes to INT and the focus is on this declaration; type NAME_FILTER and press enter—a new constant is created at the top of the class.

Now, we need to declare a Spring bean that uses this new class. Open the file `appContext.xml` in the `contacts-web` module and locate the bean with the ID `personDao`. Remove the content in the class attribute and type `PersonDao`. Then press *Ctrl* + Space bar and select the unique option. You will see that IntelliJ shows an error telling us that the constructor argument doesn't exist, so remove the constructor argument that has its name in red. Your bean declaration should look like the following code:

```
<bean id="personDao" class="org.example.dao.PersonDao"
scope="request">
<constructor-arg name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Redeploy the application to see if everything is running well.

Final adjustments

Even with our application running as expected, we can make some modifications that would improve it. You've probably noted that the `PersonTableController` is performing the role of the controller and model (`http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`). Fortunately, IntelliJ provides some refactoring tools that will help us extract the model from the `PersonTableController` class.

First of all, open the class file `PersonTableController`. After doing this, go to **Refactor** | **Extract** | **Class...**. At the Extract Class window, enter `PersonTableModel` as the name of the new class and select the properties: `selected`, `persons`, and `filterValue`; notice that when you select a property, its getter and setter methods are selected as well. Finally, click on the **Refactor** button.

The created `PersonTableModel` property was constructed as expected. However, the `PersonTableController` still remains with the getters and setters of the properties that are now at the model class, so remove all getters and setters that are related to the model properties. Notice that refactoring created the `personTableModel` property in the controller class as final. As we are using Spring, we prefer to pass an instance of the model using a setter. Change the declaration of `personTableModel` by removing the instantiation sentence and final clause, then create a setter for `personTableModel` using *Alt* + *Insert* as we did before. After this, change the `updateTable` and `filter` methods in the code as follows:

```
public void updateTable(){
  personTableModel.setPersons(retriever.retrieveAll());
}
```

```
public void filter(){
personTableModel.setPersons(retriever
.retrieveByFilter(personTableModel.getFilterValue(), 1));
}
```

At this time, we've decided that model will be a better name for the
PersonTableModel variable. Position the cursor over the personTableModel
variable anywhere in the code and press *Shift + F6*; change the name to model
and press *Enter*. When asked, answer yes to rename the respective setter.

We've noticed that we have two methods that do almost the same thing. As the
filter method is the most powerful, we will remove the updateTable method; but
before that, we should note that the updateTable method is used as init-method in
Spring. Near the updateTable declaration, there is a green icon with a light red circle
with an **m** inside, as shown in the following screenshot:
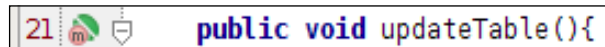
Click on that mark to navigate to the bean declaration. IntelliJ will open the
appContext.xml file and will position the cursor at the init-method parameter;
change this parameter to a filter, then press *Ctrl + B* to come back to the
PersonTableController class. Now you can remove the updateTable method
and your class should look like the following code:

```
public class PersonTableController {
  private PersonTableModel model;
  private IRetrieverDao<EntityPerson> retriever;

  public void filter() {
 model.setPersons(retriever.retrieveByFilter(
model.getFilterValue(), 1));
  }

  public void setModel(PersonTableModel model) {
    this.model = model;
  }

  public void setRetriever(IRetrieverDao<EntityPerson> retriever) {
    this.retriever = retriever;
  }
}
```

---

[73]

We still need to do some work. Open the `appContext.xml` file and create a `PersonTableModel` bean declaration as follows:

```
<bean id="personTableModel"
class="org.example.controllers.PersonTableModel"
scope="request"/>
```

Now add the model bean we just created to the controller bean. Your personTableController should look like the following code:

```
<bean id="personTableController"
class="org.example.controllers.PersonTableController"
scope="request" init-method="filter">
<property name="model" ref="personTableModel"/>
<property name="retriever" ref="personDao"/>
</bean>
```

In the end, we need to change some values in `home.xhtml`. As you know, we moved the properties that were in `PersonTableController` to `PersonTableModel`. Open the `home.xhtml` file and change all the`personTableController` usages (except the ones that are in the`p:ajax` tag) to `personTableModel`. You can do this easily using find and replace (*Ctrl + R*). Use the blue arrows to navigate through the entries IDE found and the **Replace** button to replace the selected entry. You will notice that you can do a more powerful search and replace using regex, preserving case, and so on. The following image shows search and replace in `home.xhtml`:
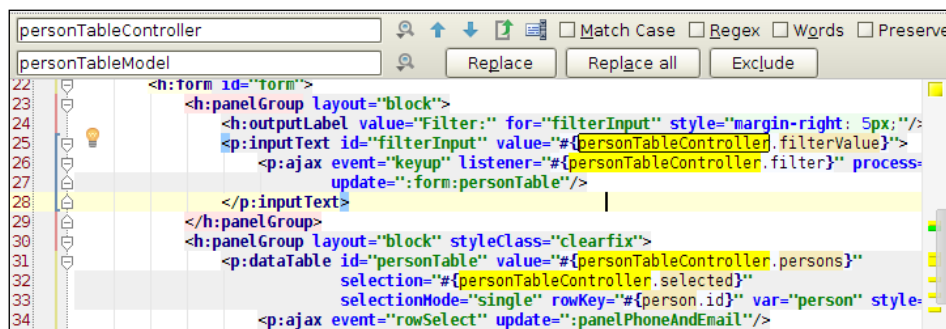
Finally, open the `PersonTableModel` class and instantiate an empty string in the `filterValue` variable. This way, all existing values in `Person` table will be retrieved when you run the application using the following command:

```
String filterValue="";
```

Now you can run the application to see the final result.

Creating SOAP web services

Our web application only retrieves the data from the database. However, we will create a web service that can insert data into database. This web service will be consumed by a desktop application we will create in *Chapter 5*, *Desktop Development*.

To create a web service, the first thing we need is to add a web service facet to our project. Open the **Project Structure** window and select the **Facets** option, then click on the **Add** button (*Alt + Insert*) and select the option **WebServices**. In the **Select Parent Facet** window, select **Web** and click on **OK**. After you add the facet, the panel will permit you to change the web service engine used; we will use the already selected engine (as in the following figure), so just click on the **OK** button to save the changes. IntelliJ will ask you if you want to download and install the SoapUI plugin—click on the **No** button. SoapUI (`http://www.soapui.org/`) is a great tool you can use to test your web service; however, if you like to use SoapUI, it is preferable that you use it as an external tool as it is more stable than the plugin version.
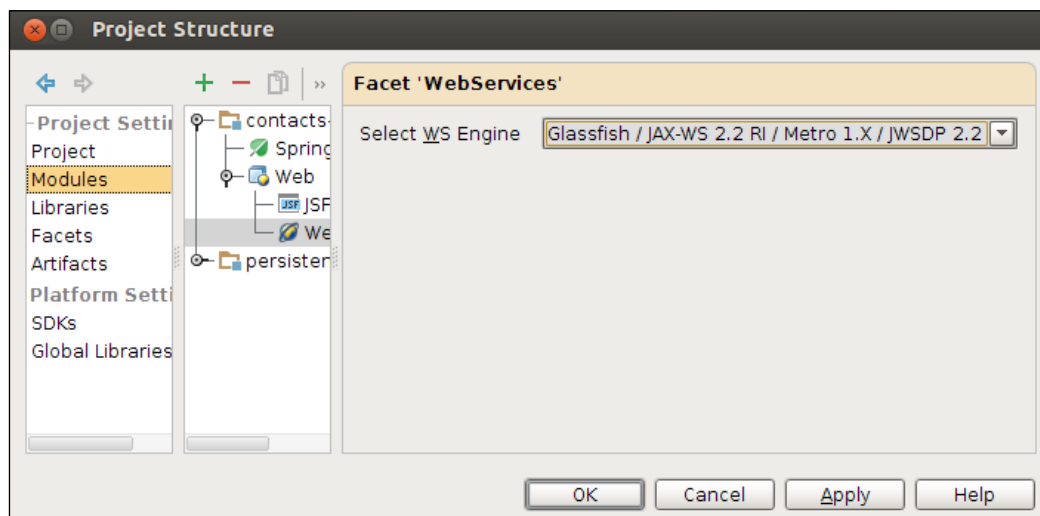
Before we create the web service, we need to create a package to organize the code. Right-click on the**org.example.controllers** package and select **New** and then click on **Package** (*Alt + Insert*). In the **New Package** window, type `ws` as the name of the package and click on **OK**. You will see that IntelliJ creates the `ws` package as a child of the**org.example.controllers** package, but we want to move it to the `org.example` package. Right-click on the `ws` package and select **Refactor** and then click on **Move**, or simply press *F6*. In the **Select Refactoring** window, select the first option: **Move package 'org.example.controllers.ws' to another package** and click on **OK**. In the **Move** window, change the **To package** value to `org.example` and click on the **Refactor** button.

Now that our package is in the correct place, right-click on it and go to **New** | **Web Service**. In the **New web service** window, enter the name of the web service as `ContactsPersister` and click on the **OK** button. You will see that IntelliJ will create an example method and a main method. As we won't use any code that is generated here, delete any method and create the methods specified in the following code:

```
@WebService()
public class ContactsPersister {
  @WebMethod
  public boolean create(EntityPerson person) {
    return false;
  }
}
```

Let's run the application to see if the web service is being deployed correctly. You will probably receive this exception: `java.lang.ClassNotFoundException: com.sun.xml.ws.transport.http.servlet.WSServletContextListener`. This happens because the JAX-WS (`http://jax-ws.java.net/`) dependency library `jaxws-rt.jar` is missing. To solve it, copy all the jar files that are available under `chapter 4|dependencies` folder and paste them in the Tomcat library folder `{$TOMCAT}/lib`.

Now that we've solved this problem, restart the server and access the address `http://localhost:8080/services/ContactsPersister`. Here, you will find more information about the deployed service and a link to his WSDL.

Creating test code

Unit tests are an important tool in agile methodology; this way, any modern IDE integrates unit test tools. Normally, the correct approach (TDD, `http://www.agiledata.org/essays/tdd.html`) to working with unit tests is by creating a test and implementing the method to pass the test, then creating another test and modifying the method to pass the new test and repeating this process until all the needs are implemented. However, here I will only show you how to execute your test codes in IntelliJ.

One thing I noticed when I began using IntelliJ is that it is difficult to find any indicator of unit test creation. I didn't find any menu named something like: `create unit test code`. After some searching, I finally noticed that there exists an intention you can use to create the test code. Basically, you only need to position the cursor anywhere in the class body, press *Ctrl + Shift + T*, and select the **Create Test** option. This is the default IntelliJ way to create a test code; however, we will use the GenerateTestCases (Generate test case requires that IntelliJ is running in JDK 7) plugin we talked about in the *Chapter 2*, *Improving Your Development Speed*. The following screenshot shows the creation of test with IntelliJ intention:
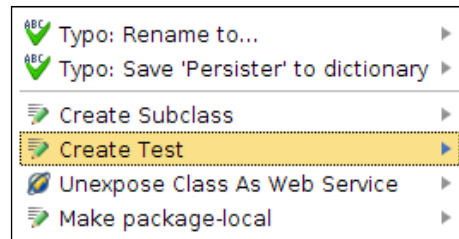
As you know, our web service will persist in the database entities passed to it; this way, we need guarantee that the `EntityPerson` object passed to be persisted is valid. Let's create a method that verifies whether or not the entity is valid in the `ContactsPersister` class:

```
protected boolean checkPersonIsValid(EntityPerson person){
return false;
}
```

Before we create the test code, we will add some comments in this method. You're probably asking, "why do we need to create comments before creating the test code?"—Simple; the GenerateTestCases plugin will look for `@should` clauses in comments to create the test code for us. So, create comments in the method using the following code:

```
/**
  * This method will check if a EntityPerson is valid
  * @param person
  * @return
  * @should return false if person is null
  * @should return false if person's email or phone list are null
  * @should return false if person's email and phone list are
empty
  * @should return false if any person's properties are null or
empty
  * @should return true if all person's properties have some value
and email or phone list has at least one value
  */
```

To generate the test code, click on the button **Generate Test Methods** that is available in the main toolbar near the **Build** button. In the **Generate Test Cases** window, it will ask which test framework we want to use; select **JUnit 4** and click on **OK**. A new window will ask if we want to add the test library to the module. Of course we want to, so click on **OK**. This time, a new window will ask which @should annotations we want to use—select all of them; you can use *Ctrl* + click or *Shift* + click to select the items. When you click on **OK**, a new window will expect you to inform it where the test code will be created; select the node available in the contacts-web module and click on **OK**. The following screenshot shows the **Choose Destination Directory** dialog:
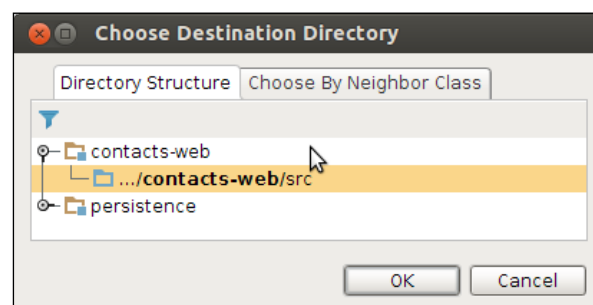
The generated test class file will present five methods, all of which follow a pattern: methodName_shouldCommentsAfterShouldAnnotation(). As our time is really short, I've provided the test code for you. Replace the class ContactsPersisterTest that exists in your project with the one that exists in the Chapter 4 folder.

Even though we still haven't implemented anything in our web service class, we can run the test case. Right-click on the class declaration and select the **Run'ContactsPersisterTes...'** option or simply use the *Ctrl + Shift + F10* shortcut; you will see that ContactsPersisterTest is available in the **Select Run/Debug Configuration** combobox, the tests will run, and, at this time, only one test fails.

At this time, nothing is implemented in the service class and, as we are not following the correct approach to unit tests, you can use the following code to pass all the tests:

```
protected boolean checkPersonIsValid(EntityPerson person){
  return person!=null &&
  person.getBirthDate()!=null &&
  person.getCompleteName()!=null &&
 !person.getCompleteName().isEmpty() &&
  person.getPlaceOfBirth()!=null &&
  !person.getPlaceOfBirth().isEmpty()&&
  person.getEmailsById()!=null &&
  !person.getEmailsById().isEmpty() &&
  person.getPhonesById()!=null &&
  !person.getPhonesById().isEmpty();
}
```

Finalizing the web service code

Our web service still doesn't do anything. We need to make it persist the entities passed by the client. Just to finalize our web service, copy the files `IPersisterDao` and `PersisterDao` that are in the `Chapter 4` folder to the `org.example.dao` package that exists in the entities module. After you've done that, open the `PersonDao` class and make it extend `PersisterDao` instead of `RetrieverDao`; then press *Alt + Insert*, select override, select the method mentioned above, and click on **OK**. The new `create` method should look like the following code:

```
@Override
public void create(EntityPerson entity) throws Exception {
  configurePhonesAndEmails(entity);
  super.create(entity);
}

private void configurePhonesAndEmails(EntityPerson person) {
  for(EntityPhone phone:person.getPhonesById())
  phone.setPersonByPersonId(person);
  for(EntityEmail email: person.getEmailsById())
  email.setPersonByPersonId(person);
}
```

Now create a `IPersisterDao` variable and implement the methods `create`, `edit`, and `delete` as they are presented in the following code:

```
@WebService
public class ContactsPersister {
  private IPersisterDao<EntityPerson> personDao;

  @WebMethod
  public boolean create(EntityPerson person) {
  if(checkPersonIsValid(person)){
    try {
      personDao.create(person);
      return true;
    }
catch (Exception e) {
throw e;
    }
  }
  return false;
  }
…
}
```

We need to create a setter for the `personDao` variable; press *Alt + Insert*, choose setter, select **applicationContext**, and click on **OK**. After IntelliJ creates the setter, annotate it using `@WebMethod(exclude = true)`; this way, you will avoid a situation where JAX-WS interprets the setter as a web service method.

As we are using Spring Framework, we need to integrate JAX-WS with Spring; this can be done using the `jaxws-spring.jar` and `xbean-spring.jar` files that are available in the `Chapter 4` folder. I imagine you already know how to add an external library to the `contacts-web` module—do it. After you've added the library, we need to create the beans used to run the service as follows:

```
<wss:binding url="/services/ContactsPersister">
<wss:service>
<ws:service bean="#{serviceBean}"/>
</wss:service>
</wss:binding>

<bean id="serviceBean" class="org.example.ws.ContactsPersister">
<property name="personDao" ref="personDao"/>
</bean>
```

Add the previous code into the `appContext.xml` file. You will see that the `wss` and `ws` prefixes will be in red; this happens because the XML file hasn't declared the necessary namespaces. Fortunately, intentions can help us again. Instead of searching over the web to find the correct namespaces, you can position the cursor over the red `wss` sentence, press *Alt + Enter*, and select the namespace ending with servlet. On the red `ws` sentence, you only need to press *Alt + Enter* as it only comprises one option.

We need to change the servlet used to expose the web service. Open the `web.xml` file and change the servlet-class content of `WSServlet` to `com.sun.xml.ws.transport.http.servlet.WSSpringServlet`, and then look for the following listener and remove it:

```
<listener>
<listener-
class>com.sun.xml.ws.transport.http.servlet.WSServletContextListen
er</listener-class>
</listener>
```

Just delete the file `/WEB-INF/sun-jaxws.xml` and you can run the application and test it with external tools (like SoapUI).

Summary

By the end of this chapter, you have seen how IntelliJ may ease your work. Even if you followed the steps of the how-to strictly, you probably saw other features that weren't mentioned in the step-by-step approach, especially the code completion feature that IntelliJ provides conforming that you are coding.

Perhaps you became confused because of the way IntelliJ works. Beginners to this IDE may get scared because some tasks are executed in unusual ways; an example of this is the fact that you can't find a menu that creates a test code. Unless you use an intention (or a plugin) you won't create it; for developers who work using TDD, easily locating where they can create test code is really important.

However, this unusual way of doing some tasks is minimized the more you work with the IDE. After some time, you will even like how IntelliJ works and will catch yourself trying to work in the IntelliJ way in other applications.

5
Desktop Development

With AWT and SWT, Swing is one of the most used technologies for developing stable GUI desktop applications in the Java world. Support for Swing components is found in practically all IDEs that work with Java; however, good integration with third-party libraries focused on Swing, such as JGoodies (`http://www.jgoodies.com/`), isn't common in IDEs, which can make the developer's job harder.

In this chapter, you will see how to work with Swing applications in IntelliJ. Specifically, we will use, the JGoodies library to manage the position of the elements, do some work with Maven, and create the client for the web service of *Chapter 4*, *Web Development*.

Creating the desktop application

Like we did in the last chapter, we will develop an application. The goal here is to show you how to work with Maven modules and Swing applications in IntelliJ. If you've never used Maven or Swing before, don't worry; all the steps described in this chapter can be executed by anyone and the hardest tasks and configurations will use files that are available in the `Chapter 5` folder.

First of all, we need to create a new module called **contacts-desktop**. Navigate to **File** | **New Module** and, in the **New Module** window, select **Maven Module**, type `contacts-desktop` in the name text field, and then click on the **Next** button. As we won't change anything in the new window, just click on the **Finish** button to create the module. After creating the module, the first file that is opened is the `pom.xml` file that is created based on the information we provided in the **New Module** window.

If you expand the **contacts-desktop** node, you will see that it contains a structure that is different from that of the modules created without the Maven facet. Notice that Java and the resources folders are using different icons too. Perhaps, for instance you right-click, on the `src/main/java` folder, try to create a new class, and don't see any option available; this happens because IntelliJ interprets this folder as a simple folder instead of a source folder. To correct this, right-click on the `src/main/java` folder and click on **Mark Directory As | Source Root**; now you can create Java files and other related files.

Let's create a package in the `src/main/java` folder. Right-click on this folder, click on **New | Package**, and type `org.examp.le.desktop` in the name text field for the package. Now we can finally create our main window.

Discovering the visual editor

When you are working with Swing applications, you probably will prefer to use a visual tool to create your pane and dialogs instead of hardcoding the visual elements. IntelliJ provides a good WYSIWYG (What you see is what you get) tool that will ease and improve your development performance, so make sure that the UI Designer plugin is enabled. To create a new pane, right-click on the **org.example.desktop** package, then click on **New,** and then **GUI Form**. In the **New GUI Form** dialog, you can make initial configurations to the form that will be created, such as the **Base layout manager**. The following figure shows you how to create a new GUI form:
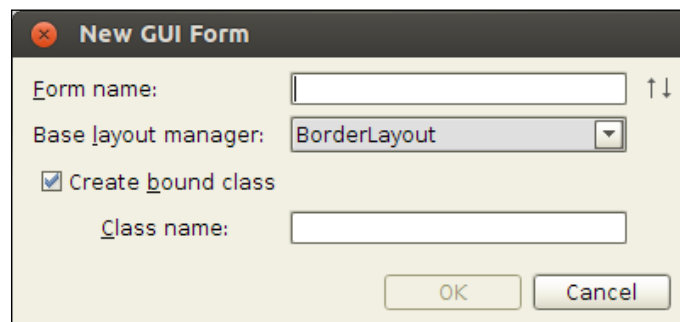

In the **Form name:** text field, enter the name `MainForm`; in the **Base layout manager:** combobox, select **FormLayout (JGoodies)** and then click on the **OK** button to create the form. When you create the form, you will see that IntelliJ opens two tool windows: **UI Designer**, which will show the properties of the element you are working with, and **Palette**, which will enable you to drag-and-drop the components in the pane. To the center of the screen, the file **MainForm.form** will be shown as a pane that you can manipulate.

We are going to develop a pane that permits the user to store a new contact in the database, calling the web service we created in *Chapter 4*, *Web Development*. As we know, a contact is composed of a person's information, phone numbers, and e-mail addresses. In this way, we will provide three fields that the user will enter: the complete name, the birth date, and the place of birth of the person; two editable tables will be available to manage the phone and e-mail values.

As we now have a pane we will use to show the user, let's add three labels and three input fields there. Go to the **Palette** tool window, drag-and-drop the **JLabel** component into the pane, and position it to the top left of the pane. Now add two more **JLabel** components next to the first **JLabel** component we added. Double-click on the first label and insert the text `Complete name:`. Do that for the second and third labels as well and enter `Birth date:` and `Place of birth:` respectively. Select the three labels simultaneously, using *Ctrl* + click; when you're done, look at the **UI Designer** tool window for the property **Horizontal Align** and change the value to **Right**.

Now that the labels are positioned, we need to add the fields that will receive the values to send to the server. In the **Palette** tool window, click on the **JTextField** component and drag-and-drop it to the right of the first label. Drag-and-drop a **JSpinner** component to the right of the second label and finally, drag-and-drop another **JTextField** component to the right of the third label component, as shown in the following screenshot:
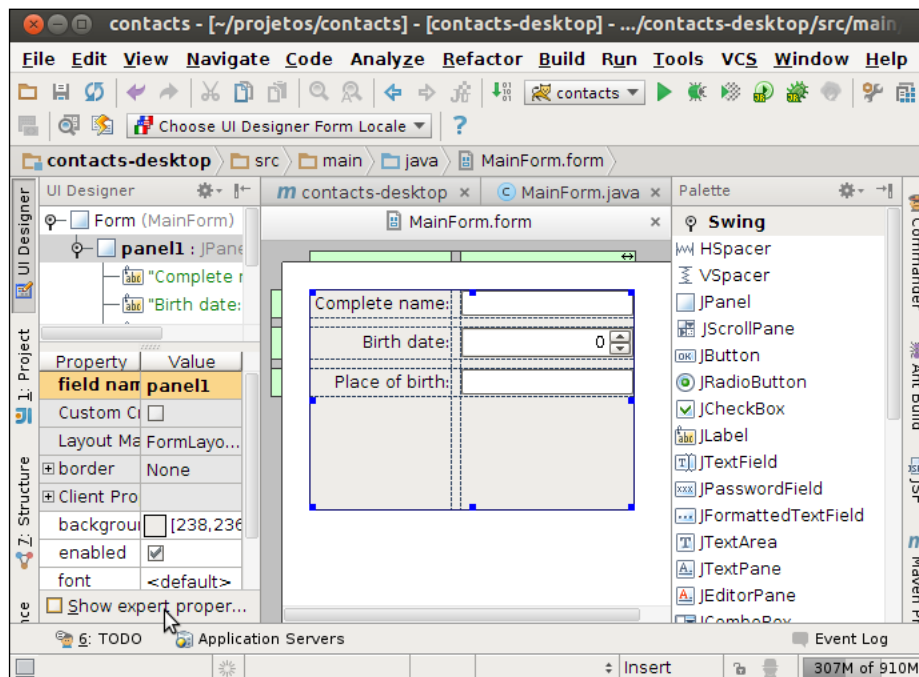
If you select an input and position the mouse over it, you will see a yellow lamp; the intention proposed by IntelliJ this time is to assign the label to the left of the selected input, so click on the lamp and choose the unique option available. Do this for all inputs.

At this time, we have created three inputs for the `EntityPerson` object. Now we need to create two tables: one for phones and another for e-mails. So, drag-and-drop a **JPanel** component to the top of the last input field; you will see that it will occupy just a part of the available width, so position the mouse at any corner and expand it until it covers all the width available. Even if you try, you can't expand the height of the new panel—no problem; it will expand based on the components you put inside it. Now drag-and-drop two **JTable** components to the new panel and position them side-by-side; notice that you can drag a component directly to the **UI Designer** tree.

When you click on each of the added tables, IntelliJ will show a yellow lamp with an option to surround them with a **JScrollPane** component. It is a good idea, to select the two **JTable** component and accept the intention surround with **JScrollPane**. When you've done this, you will notice that now the tables are using all the height available. Your tree should look like that shown in following screenshot:
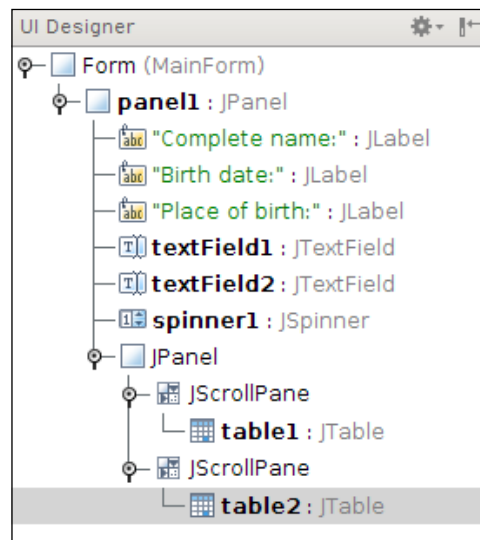
If you look at the existing columns in the tables, you will notice that the columns present static data in the column's' shape and color. However, if you preview the form by clicking on the Preview button, you will see that nothing is shown in the space reserved for the tables. The configuration of the columns can be done only in code.

Before we configure the tables, we need to specify better names for the **JTable** components we instanced. Select the left-hand side **JTable** component and change the field name property value to `phoneTable`; then select the right-hand side **JTable** component and change the field name property to `emailTable`.

We still need to create the action buttons to add and remove the phone and e-mail address values in a person's contact details; one button will save the contact in the server and another will clean the form so that we can create another contact. We want to insert the add and remove buttons for each table below, but we don't want to put the buttons inside JScrollPane; in this way, we need to wrap the scroll panel with a simple **JPanel** component. Right-click on the first **JScrollPane** in the **UI Designer** tool window and select the option **Surround With | JPanel**; then do this for the other **JScrollPanel**. Now, the two **JScrollPanel** components are surrounded with **Jpanel**.

We now need to add two buttons in each **JPanel** we just added. So, drag-and-drop one button into the panel which is on the left-hand side. Notice that the button will fill the entire horizontal space available; as we want to position the two buttons side-by-side, this is unacceptable. Before you add the other button to this panel, select the button we already added and change the property **Horizontal Align** value to **Left**; this time, the button is using just a small part of the available horizontal space.

You may imagine, that now you can drag-and-drop a new button to the right side of the first and it will be positioned exactly where you want it to. Do this now and you will see that there is a big empty space between the two buttons; but we don't want so much space. If you are an attentive developer, you will notice that to the top and to the left of the visual editor exist green bars of the same size as the panel in which we are placing the buttons. Click on the first green bar at the top of the window, and you will see that the panel with the properties will change; this will help us set the correct size for each column in the panel.

Let's change some properties for the first green column. Select the checkbox **Grow**, set its value to 1 and Width to **Minimum**, and uncheck the **Minimum** checkbox. Now select the second green bar at the top and follow the same process we did for the first column. The following screenshot shows the visual editor with the green bars and their properties:
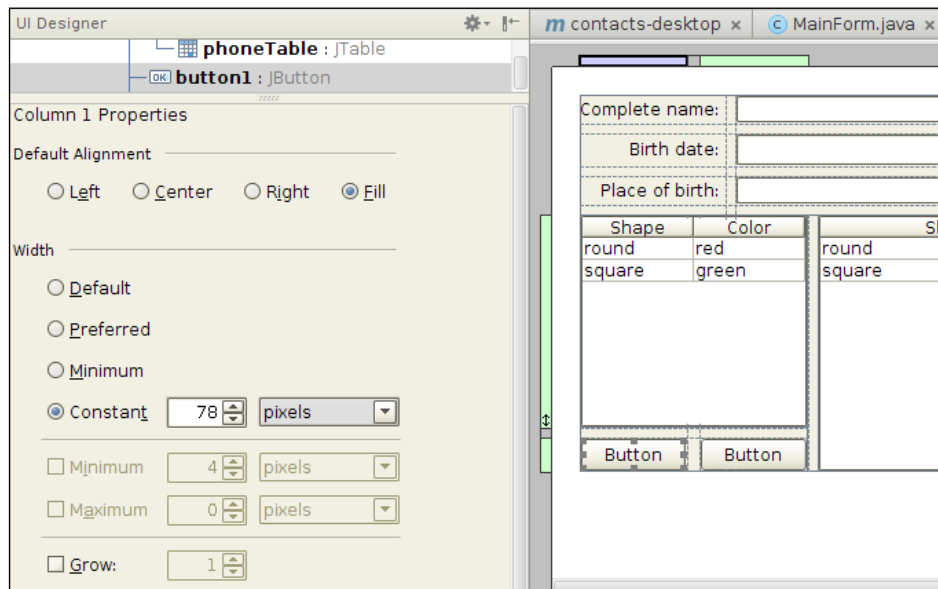
Now the two columns are of the same size; however, the first button isn't filling the entire space available. Select the left-hand side button and change the value of the property **Horizontal Align** to **Fill**. Now that you know how to add the buttons and position them, do this for the second panel.

Our buttons still don't have the correct labels and names, so do this as follows: change the label of the left-hand side button to Add and the label of the right-hand side button to Remove. Then change the name field for each button, in accordance with the label and the table that the button will manipulate, to addPhoneButton, removePhoneButton, addEmailButton, and removeEmailButton.

Finally, we need to create the save button and the clear button. Drag-and-drop a button component to the **pane1** component; notice that it will be positioned to the bottom of the panel. Then place a new button to the right of the recently added button. As you can see, the second button fills almost all the horizontal space in the panel. Again, you will notice two green bars at the top of the panel; select the first one and set the **Width** to **Minimum**, uncheck the **Minimum** checkbox, check the **Grow** checkbox and set its value to 1, and follow the same process for the second green bar.

We want to position these buttons to the right corner of the panel, so we will use an **HSpacer** component to do this. Drag-and-drop an **HSpacer** component to the left of the first button and you will see that a column that covers the entire panel will fill a smaller space and push the components to the right. Now, drag the first button and drop it into the small rectangle that separates it from the second one, and then drag the **HSpacer** component you positioned into the panel to the rectangle where the first button was positioned. Click on the middle green bar at the top of the panel and perform the same process you did for the others.

Rename the label of the left-hand side button to `&Save` and the label of the second button to `&Clear`. Notice that the & symbol at the beginning of the label will create a shortcut for this button which, in this case, will be *Alt + C* for the clear button. Finally, change the field name of the buttons to `saveButton` and `clearButton` in accordance with the label of the button. If you did everything correctly, you can click on the Preview button, which is shown in the following screenshot:

On clicking the Preview button available in the main toolbar or in the context menu, you will see a dialog like that shown in the following screenshot:

Creating the web service client

As we already have our user interface ready, we can now create the web service client that will, in fact, create the contact and send it to the database. To create a web service client, we will use a **Web Service Description Language** (**WSDL**) file—this is nothing more than an XML file that defines the methods, parameters, and other characteristics of a web service. Using this definition, IntelliJ can identify what is needed to consume the web service and will generate the necessary classes to access it.

To create web services, first of all we need our web service running, so if the contacts-web module isn't running, select it in the **Run/Debug Configuration** combobox and click on the **RUN** button. Once Tomcat is running, open the **Project Structure** tool window, expand the **contacts-desktop** node until you get to the package **org.example.desktop**, then right-click on the package and select **WebServices | Generate Java Code From WSDL**. If you followed the steps in *Chapter 4*, *Web Development* without changing anything, you can insert in the field **Web service wsdl URL** of the window that will appear, the value: `http://localhost:8080/services/ ContactsPersister?wsdl`. However, if you did change something, you will need to find the URL of the WSDL in the web service.

The previous step was shown to you just to highlight the fact that you can work with web services with different technologies (Axis 2, JAX-WS, and so on) in IntelliJ with ease; however, we will generate the client using Apache CXF with Maven. As we need to configure some complex things in `pom.xml`, it is easier for those new to Maven to just override the `pom.xml` file of the module; so, open the `Chapter 5` folder and override the `pom.xml` file in the **contacts-desktop** module to the one in the folder. In the following code, you can see the `cxf-codegen-plugin` configuration:

```
<plugin>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-codegen-plugin</artifactId>
    <version>${cxf.version}</version>
    <executions>
        <execution>
            <id>generate-sources</id>
            <phase>generate-sources</phase>
            <configuration>
                <sourceRoot>src/main/java</sourceRoot>
                <wsdlOptions>
                    <wsdlOption>
                            <wsdl>http://localhost:8080/
services/ContactsPersister?wsdl</wsdl>
                    </wsdlOption>
                </wsdlOptions>
            </configuration>
            <goals>
                <goal>wsdl2java</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

After overriding the `pom.xml` file, you will see that IntelliJ will start background tasks to meet the modifications made, such as downloading the cxf-codegen-plugin. Notice in the previous code that I'm considering that the location of the WSDL is `http://localhost:8080/services/ContactsPersister?wsdl`; if your WSDL is located in another place (for example, if your server is running in another port), change it based on your configurations. You can download the WSDL file and specify the path in the `<wsdl>` tag. If you need to change something in `pom.xml` to adapt to the changes you've made, don't forget that you can use *Ctrl* + Space bar for code completion and *Alt* + *Enter* to use intentions—perhaps you will need to use them. As you probably know, the code will only be generated when we build the project using Maven; but we still haven't created any configuration for Maven in this module.

Navigate to **Run | Run/Debug**, and click on the **Edit Configurations**... option. Now, click on the Add New Configuration button (*Alt* + *Insert*) and select Maven. In the **Name** text field, change **Unnamed** to `contacts-desktop`. In the **Parameters** tab, you will see the **Command line** text field; type in this text field the value `clean install`. Notice that IntelliJ will propose the values as you are typing. At this time, we won't change any thing else, so click on the **OK** button.

The configuration we've already created is selected; click on the **Run** button to build the module using Maven. Once Maven is running, you will notice that new classes and interfaces are present in the **org.example.ws** package. Now we finally have the classes necessary to bind the GUI with the data structure.

Data binding

Data binding is a set of techniques that permits two data sources to maintain synchronicity. At this time, we need to bind our visual elements to the data structure of the web service. Before we start binding the data, we need to make some adjustments. As you have seen, we've used a **JSpinner** component to provide the birth date of the person. However, by default, **JSpinner** doesn't know how to work with `java.util.Date` objects, so we need a way to configure the spinner without compromising the code generated by the UI Designer.

Click on the **birthDateSpinner** element and check the option **Custom create**; when IntelliJ focuses on the method `createUIComponents`, enter the following content in the method:

```
private void createUIComponents() {
        Date date = new Date();
        SpinnerDateModel dateModel = new SpinnerDateModel();
        dateModel.setValue(date);
        birthDateSpinner = new JSpinner(dateModel);
}
```

What we did is make the **JSpinner** component use `SpinnerDateModel`—now this spinner can work with dates. Notice that the `createUIComponents` method is generated by IntelliJ as a way for you to modify the instantiation of the components without compromising the auto-generated code. So, if you've checked the **Custom create** option of a component, IntelliJ won't instantiate this element; it is your responsibility to do it now.

To see if it worked, we need to run our application. If you open the **MainForm. java** class file, you will notice that the generated class doesn't extend any Swing component; this way, we probably need to configure the `MainForm` class based in the main panel. Fortunately, IntelliJ can create all the code we need to run the application. Press *Alt + Insert* anywhere inside the `MainForm` class declaration and select the option **Form main**. It will create a new main method and create and configure a `MainForm` object. To run the application, we can simply use *Ctrl + Shift + F10*. However, we will receive an exception that JGoodies libraries are missing. We can simply add the following dependency in the `pom.xml` file and run the application again:

```
<dependencies>
        <dependency>
            <groupId>com.jgoodies</groupId>
            <artifactId>jgoodies-forms</artifactId>
            <version>1.6.0</version>
        </dependency>
</dependencies>
```

Now, we can finally do the data bind. If it isn't open, open the file **MainForm.form**, then click on the button Data Binding Wizard. The icon for the Data Binding Wizard is shown in the following screenshot:

The wizard window will appear with the first radio button selected; as we won't create a new bean, select the second radio button: **Bind to existing bean**. Once you've done it, click on the button with suspension points and a new window will appear to help you find the class we will use. On the text field in the tab **Search by Name**, type `EntityPerson`. Two results will appear; select the one that is located in the package **org.example.ws** and click on **OK**. You can now click on **Next** to go to the next window.

A table with two rows will appear to select the **Bean Property** option and bind it to the Form Field. You can select the two properties according to the name of the field; in other words, select the property `placeOfBirth` for `placeOfBirthTextField` and `completeName` for `completeNameTextField`, then uncheck the checkbox **isModified()**, and then can click on the **Finish** button. The code created is, really simply, just a getter and a setter; however, it can save us some time.

Unfortunately, the bind wizard doesn't help us so much; there are two tables and a spinner that do not get binded. You can manually bind the **birthDateSpinner** element by simply adding a line in the setter and getter of the class. However, with the **JTable** components, we need to do some more work as we still haven't configured their models.

As we need to implement so many things, I will provide you with some ready code, so, this way we can focus on specific parts of implementation. In the `Chapter 5` folder, there are three files: `MainForm.java`, `EmailTableModel.java`, and `PhoneTableModel.java`; copy these files to the **org.example.desktop** package and override if necessary. Theoretically, the application can run correctly just by adding these files; however, we need to make some little changes that will guarantee that it will run well.

It's not the best idea to use a `WS` object directly in the data binding process; however, as this code is just an example, we can ignore that. Open the class `EntityPerson` that is located in the package **org.example.ws**; notice that the variables `emailsById` and `phonesById` have getters, but don't have setters. We could implement a setter for each variable or instantiate the variables and use the getters; however, this code is generated automatically, so it will be overridden every time you build the module with Maven. The simplest way to solve this problem is by creating a class that extends `EntityPerson` just to initialize these variables.

Right-click on the **org.example.desktop** package and create a new class named `WrappedEntityPerson`. Then, make it extend `EntityPerson`, press *Alt* + *Insert*, and select **Constructor**. In the following constructor type, position the cursor anywhere and press *Ctrl* + *D* to duplicate the line:

```
this.phonesById=new ArrayList<EntityPhone>();
```

Now that we have a copy of the `phonesById` instantiation sentence, change `phonesById` in the second sentence to `emailsById` (don't forget that you can use *Tab* to select the propositions IntelliJ will make). You will see that the second sentence will be underlined in red; press *Alt* + *Enter* and select the intention **Change "new ArrayList<EntityPhone>()" to "new ArrayList<EntityEmail>()"**, and then click on **Enter**. Unfortunately, it will create **ArrayList** with nothing inside <>; but, if you position the cursor inside <>, you will see that a yellow lamp appears and a new intention helps us. Click on the lamp icon (or use *Alt* + *Enter*) and select **Replace "<>" with explicit type argument**.

Now that we've created our extended class, we need to make sure that only this class will be instantiated by our code instead of the original class `EntityPerson`. I didn't find any refactoring option that could do it in an easy way, so we will use the **Replace in Path** dialog just to show you this approach. This isn't a safe way of doing this, mainly because it searches and replaces simple text and no code analysis is done when working this way—the probability that things will go wrong here is high. A safer approach would be to use the **Rename** refactoring and then change the class name back without refactoring; anyway, this project is really simple and it is a good thing to know that this tool exists.

While on any screen, press *Ctrl + Shift + R* to open the **Replace in Path** dialog. In the **Text to find** input field, enter `new EntityPerson` and in the **Replace with** input field, type `new WrappedEntityPerson`. To make sure we won't change the instantiations in other modules, on **Scope** session, select the **Module** option and then select the **contacts-desktop** module in the combobox that will be enabled. Your **Replace in Path** dialog should look as shown in the following figure:
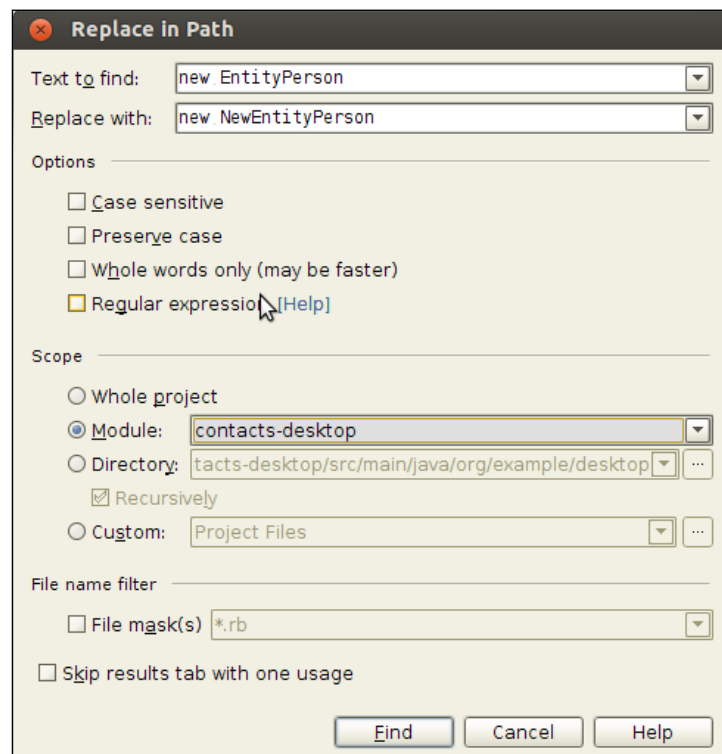
Click on **Find** to start the search. After the search is done, it will find two entries as we still don't know if we can change everything; it is more secure to use the **Replace** button instead of the other options. The first entry found was in the `MainForm` class, so we can replace it securely. The second one was found in the `ObjectFactory` class; click on the **Skip** button as we don't want to replace this.

Our code, at this time, has a problem that will make the user suffer while using the program—the data edited in the tables are only submitted to the object when the user presses *Enter* after editing a cell. To solve this, we need to simply check a checkbox. Open the **MainForm.form** file and select the two **JTable** component that are in the panel, then expand the **Client Properties** node, and check the checkbox **terminateEditOnFocusLost**.

At this time, we can run the application and add new contacts calling the web service. To finalize our program, we need to implement a clear action. Click on **clearButton**, press *Ctrl + O*, and select the first option: **ActionListener**. In the new window that will appear, select the method **actionPerformed** and click on **OK**. IntelliJ will create the listener code in the `MainForm` constructor.

The code needed to clear the panel is really simple; basically, we just need to set a new instance to the person variable. So, enter the following code in the listener:

```
clearButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        setData(new WrappedEntityPerson());
    }
});
```

Migrating GUI

Sometimes, we may come across purely coded visual components and depending on our proficiency in the Swing and AWT code, a change in this kind of user interface may become a challenge. IntelliJ provides a tool that can convert hand-coded GUI and forms created in other designers to its form format. This way it is possible to normalize the visual elements of different projects and make it easy to maintain the code even for those who don't have much proficiency in Swing/AWT code.

To make it possible to extract a form format from any GUI, we need to use the Snapshot feature. IntelliJ will analyze the user interface that is running and will take some snapshots of it, then it will traverse the tree components of the snapshot and create the form in its own format.

In the Chapter 5 folder, there is a file named Find.java. I didn't create this file, I just found it over the internet, so I don't have any idea about its code; I just want to convert it to the IntelliJ format. Copy the file Find.java to the package **org. example.desktop**, right-click anywhere inside it, and select **Run "Find.main()"** or simply press *Ctrl + Shift + F10*.

We need to run the application to capture the snapshot; however, we need to configure it before we run it, so close the **Find** dialog. Notice that a configuration was created for this file; Navigate to **Run | Run/Debug**, and click on **Edit Configurations...**.

Expand the node **Applications** and click on **Find**; in the right panel, check the checkbox **Enable capturing form snapshots**, then click on **OK**. Now, right-click on the package **org.example.desktop** and select **New | Form Snapshot**; when asked if you want to run the application, click on **Yes**. In our case, we just have one dialog; but, if you are working in an application with more than one dialog, open the dialog that you want to take the snapshot of, and click on **OK** in the dialog that IntelliJ will show you. The following screenshot shows the **Create Form Snapshot** window:
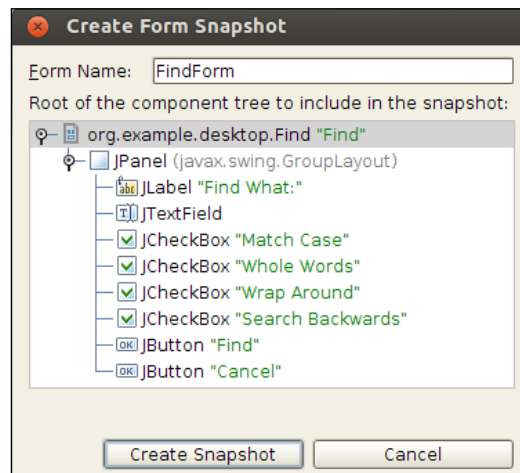
In the **Create Form Snapshot** window, select the first node; this will grant that all components of the **Find** dialog will be extracted. Name the new form as FindForm and click on **Create Snapshot**. IntelliJ will tell us that it can't understand the **GroupLayout** element and say that its components will be ignored; click on the **Yes** button. After some time, you will see that a message with a timeout error will return to us.

Despite being a good idea, IntelliJ's snapshot tool can't extract the elements of a GUI all the time; it will sometimes fail.

---

[96]

Summary

In this chapter, you saw that the integration between the IDE and Swing components can improve the development work, no matter which kind of layout manager you use. We also learned a little about Maven integration with IntelliJ and noticed that even without using all the features that this integration provides, working with Maven in IntelliJ is something that's really easy.

At the end of the chapter, we also saw the Snapshot tool and, unfortunately, couldn't extract the form from the Java code provided. As a warning, never forget that we should not completely trust a tool, even if the tool does most of the things it promises, especially when it promises some kind of a miracle.

Index

L

live templates
  about 30-32
  parameterized template 30
  simple template, 11
  surround template 30
Local History option 34

M

Make Project button 60

N

navigation
  features, highlighting 9, 10

O

Object/Relational Mapping. *See* **ORM**
On frame deactivation option 18
ORM
  about 53
  database entities, creating 54, 55
  problems 56

P

Palette tool window 85
parameterized template 30
PersonTableModel class 74
PersonTableModel property 72
plugins
  about 38, 39
  GenerateTestCase 41
  Hungry Backspace 40
  JRebel 39
  key promoter 40
primary key option 47
productivity guide 26
Project Structure button
  Artifacts 17
  Facets 17
  Libraries 17
  Modules 16

  Project 16
Project tool tab 28
Project tool window 23

Q

Query button 51

R

refactoring techniques
  using 32-33
Refresh button 51
Replace in Path dialog 94
retrieveByFilter method 71
Run tool 63

S

Schemas & Tables tab 46
Shift + Delete key 29
Show visualization option 49
Signature dialog 33
simple template 30
Skip button 95
SoapUI 75
SOAP web services
  code, finalizing 79, 80
  creating 75
  test code, creating 76-78
source code
  navigating through 27, 28
surround template 30
Spring Frameworks 12
Synchronize button 45

T

Table editor option 50
Table option 47
tables
  creating 47-49
  data, manipulating 50-53
tasks
  managing 35, 36
TODO
  about 37

[PACKT PUBLISHING] Thank you for buying
Getting started with Intellij IDEA

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
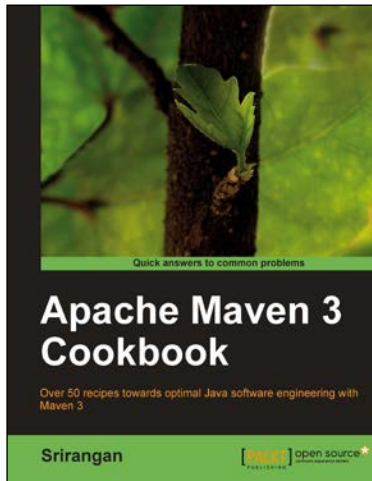
Apache Maven 3 Cookbook

ISBN: 978-1-84951-244-2   Paperback: 224 pages

Over 50 recipes towards optimal Java software engineering with Maven 3

1. Grasp the fundamentals and extend Apache Maven 3 to meet your needs

2. Implement engineering practices in your application development process with Apache Maven

3. Collaboration techniques for Agile teams with Apache Maven

Gradle Effective Implementation Guide

ISBN: 978-1-84951-810-9   Paperback: 382 pages

Empower yourself to automate your build

1. Learn the best of Gradle

2. Work easily with multi-projects

3. Apply Gradle to your Java, Scala and Groovy projects

Please check **www.PacktPub.com** for information on our titles

[PACKT]
PUBLISHING

Instant Apache Maven Starter

ISBN: 978-1-78216-760-0   Paperback: 62 pages

Get started with the fundamentals of developing Java projects with Apache Maven

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.

2. Create Java projects and project templates with Maven archetypes

3. Manage project dependencies, project coordinates, and multi-modules
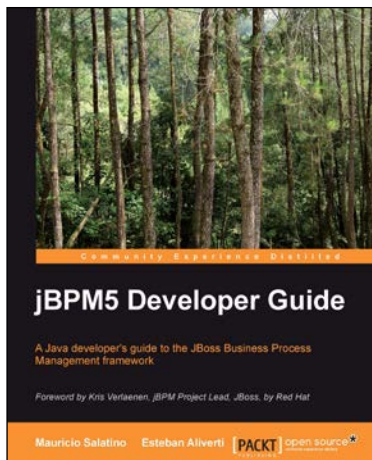
jBPM5 Developer Guide

ISBN: 978-1-84951-644-0   Paperback: 364 pages

A Java developer's guide to the JBoss Business Process Management framework

1. Learn to model and implement your business processes using the BPMN2 standard notation

2. Model complex business scenarios in order to automate and improve your processes with the JBoss Business Process Management framework

3. Understand how and when to use the different tools provided by the JBoss Business Process Management platform

Please check **www.PacktPub.com** for information on our titles