

# intelli**IDEA** 3.0

*Develop with pleasure!*



© 2000 - 2003 JetBrains, Inc. All rights reserved.

JetBrains, IntelliJ, IDEA, and IntelliJ Labs are either registered trademarks or trademarks of JetBrains s.r.o. in the Czech Republic and in other countries. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Information in this document is subject to change without notice. JetBrains, Inc. makes no warranties, neither expressed nor implied, in this document. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), or for any purpose, without the express written permission of JetBrains, Inc.

# **JetBrains, Inc.**

Klánova 9/506  
147 00 Prague  
Czech Republic  
Phone: +420 2 4172 2501  
Fax: +420 2 6171 1724

WEBSITE:

[www.intellij.com](http://www.intellij.com)

DEVELOPER'S NETWORK:

[www.intellij.org](http://www.intellij.org)

INTELLIJ TECHNOLOGY NETWORK:

[www.intellij.net](http://www.intellij.net)

Acknowledgements .....	3
About IntelliJ IDEA.....	3
Why Read This Overview?.....	3
IntelliJ IDEA: The Intelligent, Usable Java Editor.....	4
Code Completions .....	4
Basic (Ctrl + Space).....	5
Smart-Type (Ctrl + Shift + Space) .....	5
Class Name (Ctrl + Alt + Space).....	6
Import Assistant.....	7
Live Templates .....	8
Searching for Usages .....	10
Code Layout Manager .....	12
Optimize Imports.....	13
Intention Actions .....	13
Opening Class by its Short Name.....	16
Debugging .....	16
Keymapping.....	17
Navigation .....	18
Code Inspection .....	20
Refactoring .....	22
What is Refactoring? .....	22
Renaming.....	22
Move.....	23
Introduce Variable .....	27
Extract Interface / Superclass .....	28
Extract Method .....	30
Inline Method .....	30
Encapsulate Field.....	31
Change Method Signature .....	32
J2EE Introduction .....	33
JSP Development Support.....	33
XML Development Features .....	35
EJB Integration.....	36
Collaboration Tools .....	37
CVS Integration.....	37
Jakarta Ant.....	39
JUnit.....	40
Jikes .....	41
Visual SourceSafe.....	42
Resources.....	43
Open API .....	43
Default Keystroke Index.....	44
Editing .....	44
Compile and Run.....	45
Debugging .....	45
Navigation .....	45
Refactoring .....	46
Live Templates .....	46

## Acknowledgements

A **Big thanks** to the IDEA development team for making a product that is easy to use and understand.

*Special thanks* to all of the community members who gave us suggestions and constructive feedback for this Overview, including but not limited to: Jacques Morel, Michal Szklanowski, Måns af Klercker, Frantisek Kolar, James Stennett, Sascha Weinreuter, Chris Miller, Dmitry Boulichkov, Patrik Andersson, Bob Frazier, Bruce Ritchie, Christopher Cobb, David Jansen, Daniel Rabe, William Randolph Royere III, Ted Hill, Tom Schreiber, Jon Steelman, Kendall Collett, and Saumendra Poddar.

## About IntelliJ IDEA

*IntelliJ IDEA* is an industry leading Java IDE power packed with leading-edge development features which includes: industry setting refactoring support, intelligent code editing assistance, a wide range of J2EE development features for rapid web-application development, a powerful Code-Inspection tool, integrated CVS support, an Open API for third-party plug-in support, and a mountain of other productivity features that make Java development a pleasure.

This overview was written to better inform you about the powerful yet easy-to-use features and functions found in IDEA.

## Why Read This Overview?

*IntelliJ IDEA 3.0 Overview* was written for developers, project managers, architects, or even sales staff – anyone who wishes to accelerate their learning curve in regards to the powerful features and functions under IDEA’s hood without getting into the type of detail that may normally cause sudden drowsiness and put you to sleep.

This overview assumes you have some understanding of Java fundamentals; it is not necessary for you to have an expert, up-to-date, edge of your seat understanding of all Java based or centered technologies. IDEA is very user friendly, and will be an excellent companion for you whether you are an advanced Java developer or a new student of Java. This overview hopes to be as friendly.

In conclusion, we believe that IDEA is the world’s preeminent Java IDE. We hope that this overview will help us further that claim in your eyes. However, we also know that despite all of the talk at the end of the day, it will be IDEA’s deeds and not this overview’s words – *Acta non verba* – that matter when it comes to influencing you to use IDEA. Therefore, you are both highly encouraged – and triple-dog dared – to give IDEA a few minutes of your time for it to impress upon you its numerous productivity features that will turn you into an efficient coding machine. After all, what do you have to lose besides your old IDE?

## IntelliJ IDEA: The Intelligent, Usable Java Editor

Let us think for a moment about the different philosophies behind building Ferraris and Ford Escorts. Ferraris are put together by hand, take days to build, incorporate the newest technology available for their specific function (racing), and use automotive parts of the highest quality. On the other hand, Ford Escorts are made on an assembly line where computers and automated machines do most of the building with limited human intervention, and hundreds can be made each day, and the automotive parts they use are cheap and not very vibrant.

Now, imagine if it were possible to make a Ferrari as quickly and as easily as an Escort while surpassing the quality of a hand made model. This might be a terrible analogy, but the premise is quite clear: In the Java industry, IDEA helps you make “Ferrari” like applications in the time one would make an “Escort” like application. In other words, IDEA dramatically increases your productivity level (thus cutting over-all development cost) while reducing tedious and repetitive tasks.

IDEA’s mantra is summed up in two words: **Intelligence** and **Usability**. Every facet of IDEA’s construction, its features, functions, and even UI layout, were all created in a manner to ride the line of this philosophy: its simple key stroke navigation, editing assistant features, code layout manager, live templates, and more.

The following sections will cover many of IDEA’s editor features, giving you a better understanding of why IDEA has become the *de facto* industry leader in IDE innovation, and why we just say: “Develop with Pleasure!”

### Code Completions

No matter what the size or scope of the project you are working on, basic “grunt” work has to be completed on each project before you can enjoy the fruits of your labor. IDEA was constructed to help speed up development by lending a virtual third hand to help you complete many of the tedious chores of Java programming. It provides assistance in completing such annoying yet fundamental tasks such as typing code or constructing syntactically correct Java statements. IDEA cannot read your mind, but it can come pretty close: It is armed with today’s most powerful code completion features which aid you in hammering out code without breaking a sweat.

Seasoned Java programmers will notice a remarkable improvement in productivity resulting in swifter project completion; you will spend less time typing and more time thinking and designing. Junior and entry-level programmers will benefit too, because IDEA will help them select the correct Java code fragments required. No matter how much development experience you have, IDEA will give you more confidence and improve your proficiency with Java.

IDEA provides with three different types of code completion features to help you speed up your coding tasks: *Basic*, *Smart-Type*, *Class Name*.

## Basic (Ctrl + Space)

IDEA's Basic code completion feature helps you complete a variety of simple but time consuming coding tasks. As shown in figures *Code Completions 1.1* and *1.2*, you can invoke this completion feature to complete basic Java syntax. Simply type one or more characters, and then press Ctrl + Space.

```
10 public class BasicCodeCompletion e| {  
11
```

*Code Completions 1.1*

```
10 public class BasicCodeCompletion extends| {  
11  
12  
13 }
```

*Code Completions 1.2*

As shown in figures *Code Completions 1.3*, the Basic code completion feature will also allow you to quickly access and insert any Java class, method, and variable from any package that has been imported or is being utilized else where in the project prior to invoking this function.

```
10 import javax.swing.*;  
11  
12 public class BasicCodeCompletion extends JPanel {  
13  
14     public void createButton() {  
15  
16         JButton sampleButton = new JButton();  
17         sampleButton.addA|  
18  
19     }  
}
```

m	addActionListener(ActionListener l)	void
m	addAncestorListener(AncestorListener listener)	void

*Code Completions 1.3*

*addActionListener* method implementation from *javax.swing.\** package

## Smart-Type (Ctrl + Shift + Space)

IDEA's Smart-Type code completion functionality helps users select the correct data types to implement in relevant locations depending on what has already been written in the code. For example, as shown in figures *Code Completions 1.4*, when the Smart-Type completion feature is invoked, a pop-up with appropriate selections (in this example, only one to choose from) appears ready to implement the selected item.

```

10 import javax.swing.*;
11
12 public class SmartTypeCompletion extends JPanel {
13
14     public void createButton() {
15
16         JButton sampleButton = new JButton();
17         sampleButton.addActionListener(new );
18     }
19 }

```

Code Completions 1.4

Once the desired item from the pop-up has been selected, IDEA will automatically implement it with all of its corresponding components as shown in figure *Code Completions 1.5*.

```

10 import javax.swing.*;
11 import java.awt.event.ActionListener;
12 import java.awt.event.ActionEvent;
13
14 public class SmartTypeCompletion extends JPanel {
15
16     public void createButton() {
17
18         JButton sampleButton = new JButton();
19         sampleButton.addActionListener(new ActionListener() {
20             public void actionPerformed(ActionEvent e) {
21             }
22         });
23     }
24 }

```

Code Completions 1.5

## Class Name (Ctrl + Alt + Space)

IDEA's Class Name completion feature as shown in figures *Code Completions 1.6 and 1.7* makes it easy to automatically suggest and implement the name of any class (and its corresponding import, if needed) anywhere in any project or library. (Basic code completion, on the other hand, utilizes only imported packages or those being resolvable in the current scope).

```

11 public class ClassNameCompletion extends JP| {

```

Code Completions 1.6

List of selectable classes in pop-up after Class Name code completion function has been invoked

```
10 import javax.swing.*;
11
12 public class ClassNameCompletion extends JPanel {
```

Code Completion 1.7

*JPanel* is implemented along with its required import package `javax.swing.*`

## Import Assistant

In the beginning, there was nothing! Those of us who started programming Java in *notepad* (Java pre-history for you newbies) or some other primitive editor will really appreciate IDEA's import assistant features that practically do everything for you. Start typing code, and once you input a Java class short name (for example: *JFrame*), the import assistant will automatically launch a pop-up suggesting for you to import the relevant corresponding Java class:

```
8
9
10
11
12 public class LiveTemplates extends JFrame {
13
```

Import Assistant 1.1

However, IDEA will not only make a suggestion, but it will enable you to actually import the suggested Java class with one stroke of the keyboard. In figure *Import Assistant 1.1*, you will notice that IDEA has identified the Java class short name lacking an import statement and has offered to import the corresponding class by pressing **Alt + Enter** on the keyboard.

```
8
9
10 import javax.swing.*;
11
12 public class LiveTemplates extends JFrame {
13
```

Import Assistant 1.2

After selecting **Alt + Enter** on the keyboard, the statement is imported, the pop-up disappears, and the red text on *JFrame* (highlighting feature) vanishes. All of this is done without your caret position moving!

The import assistant also works when importing large blocks of code. For example, if you “copy” a block of code from one project file, and you paste this block of code into another project file, the import assistant will also prompt you for permission to import the relevant Java classes that are lacking in the target class/interface. In figures *Import Assistant 1.3*, *1.4*, *1.5*, you can see the copy and paste process in action.

```

10 import javax.swing.*;
11
12 public class OneProject extends JFrame {
13
14     JButton butLine = null;
15     JButton butRectangle = null;
16     JButton butCircle = null;
17     JToolBar myToolBar = null;

```

*Import Assistant 1.3*

*Copying from one project*

```

11
12 public class NewProject {
13
14     JButton butLine = null;
15     JButton butRectangle = null;
16     JButton butCircle = null;
17
18 }
19

```

**Select Classes to Import**

The code fragment which you have pasted uses classes that are not accessible by imports in the new context. Select classes that you want to import to the new file.

javax.swing.JButton

*Import Assistant 1.4*

*Pasting into new project*

```

10 import javax.swing.*;
11
12 public class NewProject {
13
14     JButton butLine = null;
15     JButton butRectangle = null;
16     JButton butCircle = null;
17

```

*Import Assistant 1.5*

*Missing class is imported*

No doubt this was a simple example; however, just imagine how useful this feature is going to be when your project manager starts pushing to speed things up.

## Live Templates

IDEA is the ideal IDE for rapid development. It incorporates an advanced *Live Templates* technology that enables developers to input lines of code constructs by short name that inputs evaluated expressions and type casts all with one key-stroke. Coding has never been faster or easier! As shown in figure *Live Templates 1.1*, you only need to type the short name to invoke the code template.

```
38 public void actionPerformed(ActionEvent e) {
39     sout
40 }
```

Live Templates 1.1

Short name template "sout" being implemented

After the short name has been typed, select the **Tab\*** key, and the entire statement will be imported into the source code as shown in figure *Live Templates 1.2*.

```
38 public void actionPerformed(ActionEvent e) {
39     System.out.println(" ");
40 }
```

Live Templates 1.2

Tab is selected, and the live template is imported

The entire live template index can be accessed by selecting **Code | Insert Live Template** on the main tool bar or by pressing **Ctrl + J** on the keyboard as shown in figure *Live Templates 1.3*. You will notice that if you begin typing, the menu will adjust according to the first known characters you input, allowing you to narrow down your choices quickly.

```
25 button = new JButton("");
26 button.addActionListener(this);
27 pane.add(button, BorderLayout.WEST);
28 setSize(100, 100);
29
30 it
31 itar Iterate elements of array
32 itco Iterate elements of java.util.Collection
33 iten Iterate java.util.Enumeration
34 itit Iterate java.util.Iterator
35 itli Iterate elements of java.util.List
36 ittok Iterate tokens from String
37 itve Iterate elements of java.util.Vector
38
```

Live Templates 1.3

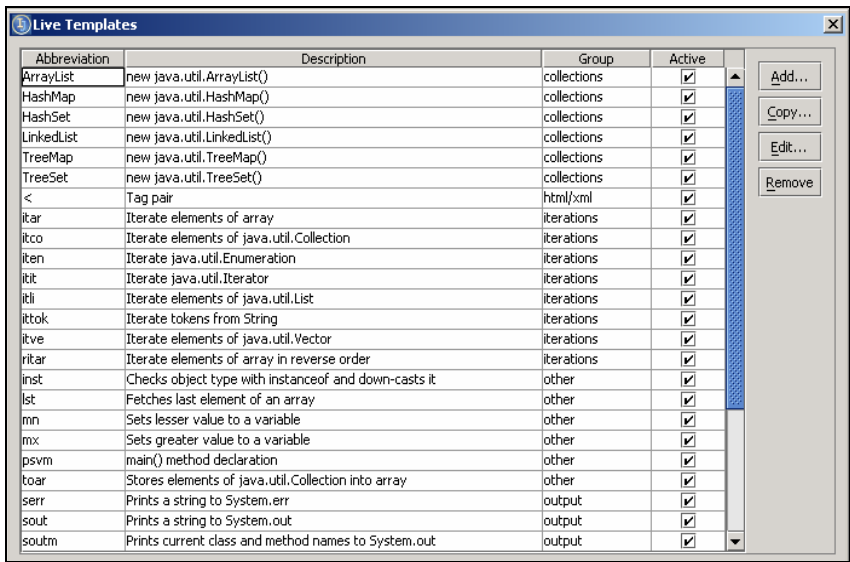
Ctrl + J brings up the live template index

When an item in the index is selected, the corresponding template is implemented as shown in *Live Templates 1.4*

```
25 button = new JButton("");
26 button.addActionListener(this);
27 pane.add(button, BorderLayout.WEST);
28 setSize(100, 100);
29
30 for (int i = 0; i < vector.size(); i++) {
31     Object o = (Object) vector.elementAt(i);
32 }
```

Live Templates 1.4

Depending on your project's requirements, you can edit the existing live templates or add more templates to the index by creating your own. To call up the Live Templates editor, select **Options | Live Templates** on the main menu toolbar. Once you have brought up the Live Template list, you can select the Live Template you wish to edit or add a new one. The process is pretty straightforward. See figures *Live Templates 1.5*.



Live Templates 1.5

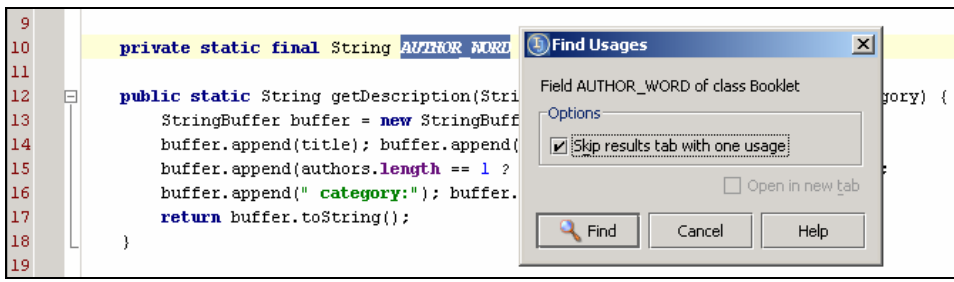
Live Templates editing / adding panel

\* **Tab** is the default key for initiating this function. However, you can use any key you wish to use if you change the default key layout setting. See the Key Mapper section for more information.

## Searching for Usages

Anyone who has worked on a large development project knows that when you want to find a specific class, method, field, or variable in a project, finding it in a sea of code is half the battle; determining the functionality of those corresponding classes, methods, fields, or variables in the code is another battle all in itself. IDEA's *Search for Usages* function was designed to strengthen the ability of developers to quickly hunt down sought out item usages. When the usage search function is invoked, usage results are displayed in an easy to read navigational tree for easy item access.

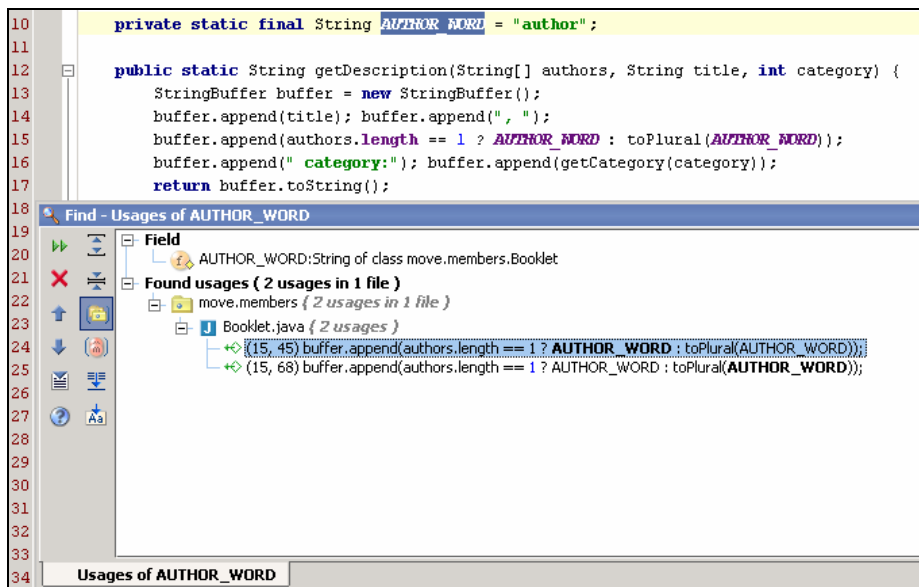
As shown in figure *Searching for Usages 1.1*, any item in a project can be searched in order to find out where that item is being used.



Searching for Usages 1.1

The project item "AUTHOR\_WORD" is being searched.

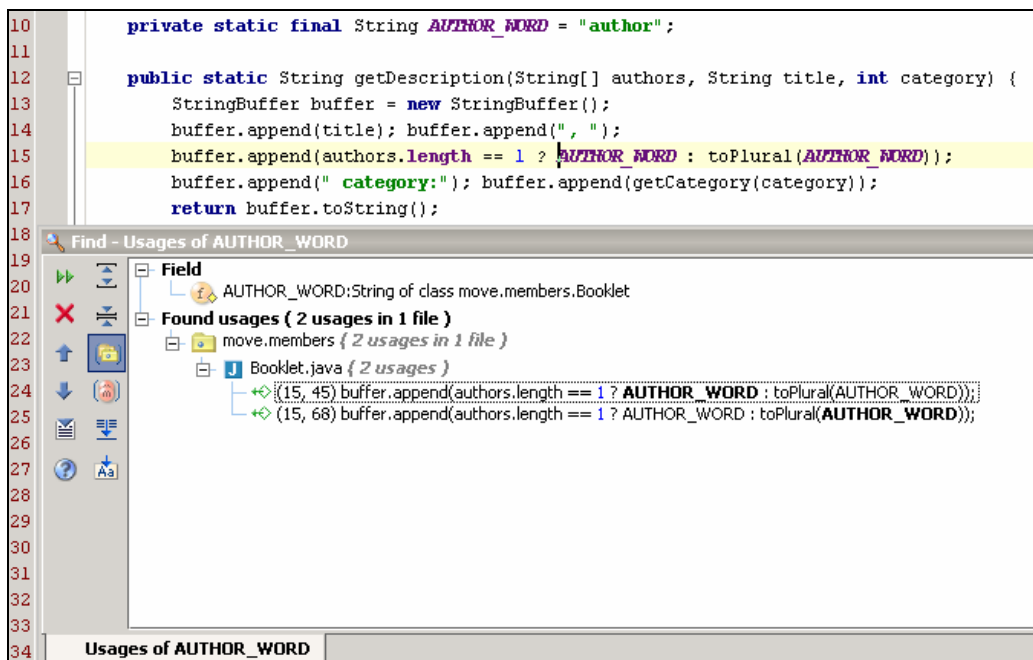
As noted above, all results are viewable in an easy to navigate tree panel as shown in figure *Searching for Usages 1.2*.



Searching for Usages 1.2

Navigational tree with results

When a searched result is selected in the navigational tree, the caret is transported to the actual item location in the source code with a simple double-click, as shown in figure *Searching for Usages 1.3*.



Searching for Usages 1.3

Navigational tree item location returned to source code

In addition to this generic usage search, IDEA also enables users to search for specific element types by incorporating a diverse array of search options and filters. The *Search for Usage* feature will search not only code within the immediate editor window, but you can also enable it to search an entire project.

## Code Layout Manager

Have you ever looked at someone's code, and thought to yourself, "What the heck is going on in this code?" Well, you are not alone. It is a well known fact that if source code is organized in a chaotic way, future development or additions to that same source code could become a daunting and time consuming task. IDEA is the perfect tool for creating, optimizing, controlling, and directing a uniform approach to code development layout.

```
5 public class BookletEditor extends JDialog {
6     JTextField myNameField;
7     JButton myOKButton;
8     private String getEnteredName() {
9         return myNameField.getText();
10    } public String getBookletName() {
11        return getEnteredName();
12    } private void validateOKButton() {
13        myOKButton.setEnabled(getEnteredName().length() > 0);
14    } public void usage() {
15        validateOKButton();
16    } }
17
```

Code Layout Manager 1.1

Select the block of code you want to format

Utilizing this powerful feature is initiated at the touch of a key. As shown in figure *Code Layout Manager 1.1*, you only have to highlight the block of code you wish to format and then select **Ctrl + Alt + L** (or from the menu *Tools | Reformat Code*). Depending on your layout preference, the code is automatically arranged as shown in figure *Code Layout Manager 1.2*.

```
6 public class BookletEditor extends JDialog {
7     JTextField myNameField;
8     JButton myOKButton;
9
10    private String getEnteredName() {
11        return myNameField.getText();
12    }
13
14    public String getBookletName() {
15        return getEnteredName();
16    }
17
18    private void validateOKButton() {
19        myOKButton.setEnabled(getEnteredName().length() > 0);
20    }
21
22
23    public void usage() {
24        validateOKButton();
25    }
26 }
```

Code Layout Manager 1.2

IDEA's default code layout reformatting result

In addition to highlighting individual blocks of code, the code layout feature also allows you to format entire classes and even entire projects all at the stroke of a key. If you are a project manager, you can even export your particular style preference to everyone in your team via email!

## Optimize Imports

In addition to the code layout manager function, another great tool for tidying up code in IDEA is the optimize imports feature. The optimize imports function searches for and removes redundant and unused imports that have a tendency to turn readable code into the exact opposite.

As shown in figure *Optimize Imports 1.1*, there are three “grayed-out” imports that are not currently being used by the open project (they may have been being used, but not now). Simply select the Optimize Imports function (menu **Tools | Optimize Imports** or **Ctrl + Alt + O**) and these imports will be removed as shown in *Optimize Imports 1.2*

```
9  import java.awt.*;
10 import java.beans.*;
11 import java.io.*;
12
13 public class OptimizeImports {
14
15     OptimizeImports() {
16
17     }
```

*Optimize Imports 1.1*

*Three grayed out imports to be removed*

```
9
10
11 public class OptimizeImports {
12
13     OptimizeImports() {
14
15     }
```

*Optimize Imports 1.2*

*Three grayed out imports have been removed*

## Intention Actions

Sometimes the greatest of ideas suddenly appear in your head like a ton of bricks, and when we find ourselves in such creative interludes, we do not want to be bothered with little things. When it comes to coding, IDEA’s ability to create classes, methods, fields, and local variables from unknown usages is the work horse you need for taking care of the little things, so you do not have to be bothered when re-structuring, re-designing, or just adding new things to the source code.

Enter the Light bulb: The crafty little icon that magically appears throughout the development process to give you a helping hand.

For example, as shown in figure *Intention Actions 1.1*, IDEA allows you to first implement an unknown usage, and then after the usage has been implemented, the light bulb studiously alerts you to the code’s missing constructs.

```

12 public class Server {
13     void processRequest(Request request, int flags) {
14         allocateResources();
15
16         createAndStartProcessingThread(request, flags);
17     }
18
19     private void allocateResources() {
20     }
21 }

```

Intention Actions 1.1

Unknown usage import correction dialog

As the unknown usage pop up appears, IDEA offers a variety of import options depending on current code construction. After one of the various selections has been chosen (in this example, to create a new method), IDEA intelligently creates and then places the selection into an appropriate position within the source code editor as shown in figure *Intention Actions 1.2*.

```

12 public class Server {
13     void processRequest(Request request, int flags) {
14         allocateResources();
15
16         createAndStartProcessingThread(request, flags);
17     }
18
19     private void createAndStartProcessingThread(Request request, int flags) {
20     }
21 }

```

Intention Actions 1.2

Selected item imported into source code

After the selection has been imported, you can then continue to edit this newly imported selection or you can return back to your previous place in the source code editor by selecting **Ctrl + Shift + Backspace** and continue working.

Not only will IDEA create methods for you, it will also implement methods into your code from packages where the methods already exist. As shown in figure *Intention Actions 1.3*, a known Java keyword has been entered into the editor, and IDEA will suggest that you implement this keyword's corresponding items (in this case, the `MouseListener` methods).

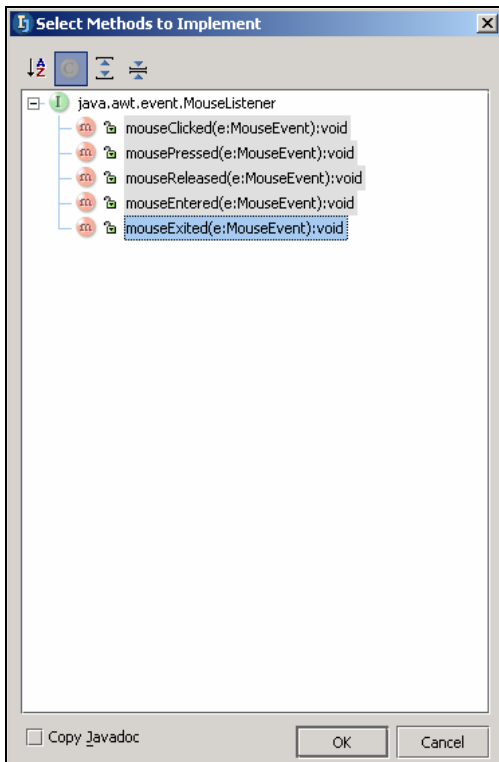
```

10 import java.awt.event.MouseListener;
11
12 public class BasicCodeCompletion extends ClassLoader implements MouseListener {
13
14     Implement Methods
15     Make 'BasicCodeCompletion' abstract
16 }

```

Intention Actions 1.3

After the selection has been chosen, in this case the "Implement Methods" selection for `MouseListener`, IDEA will prompt the user as shown in figure *Intention Actions 1.4* to select which methods from the `MouseListener` package they would like to implement. IDEA allows you to implement all of the methods from the package, or just individual methods by highlighting individual selections and then selecting **OK**.



Intention Actions 1.4

After the methods in the package have been selected, and the OK button selected, IDEA will automatically import and implement these packages into your project as shown in figure *Intention Actions 1.5*.

```

10  import java.awt.event.MouseListener;
11  import java.awt.event.MouseEvent;
12
13  public class BasicCodeCompletion extends ClassLoader implements MouseListener{
14  public void mouseClicked(MouseEvent e) {
15  }
16
17  public void mousePressed(MouseEvent e) {
18  }
19
20  public void mouseReleased(MouseEvent e) {
21  }
22
23  public void mouseEntered(MouseEvent e) {
24  }
25
26  public void mouseExited(MouseEvent e) {
27  }

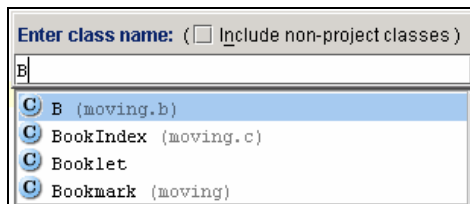
```

Intention Actions 1.5

## Opening Class by its Short Name

In large projects with multiple classes, it has been up until now quite the norm to access individual classes within a project by selecting the “Open Class” option somewhere in the main toolbar menu. However, this process has now been streamlined. IDEA ensures that you no longer have to close the window of one class to open up another, nor take your hands off the keyboard for that matter.

Simply select **Ctrl + N**, and once you start to enter the first letter of the sought out class, IDEA will dynamically begin to shave down your choices. Once your choice has been selected, the desired class will be viewable in the source code editor panel.

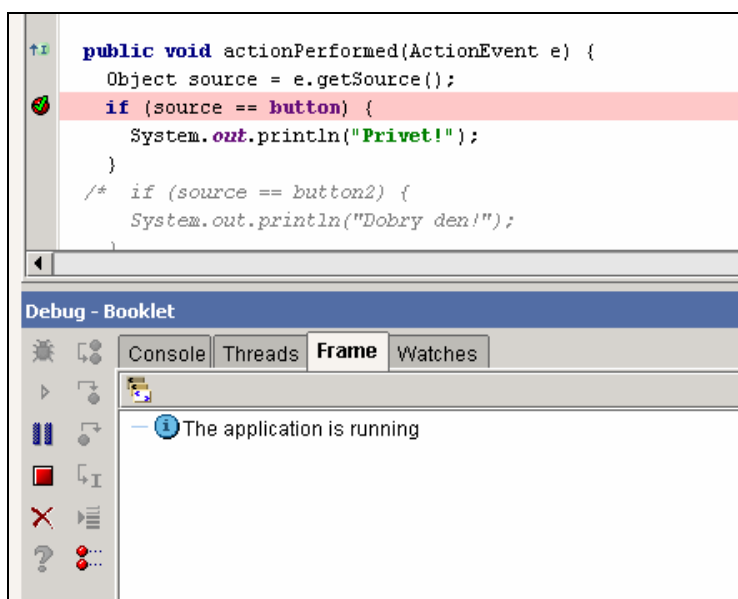


Opening Class by its Short Name 1.1

Class search by short name

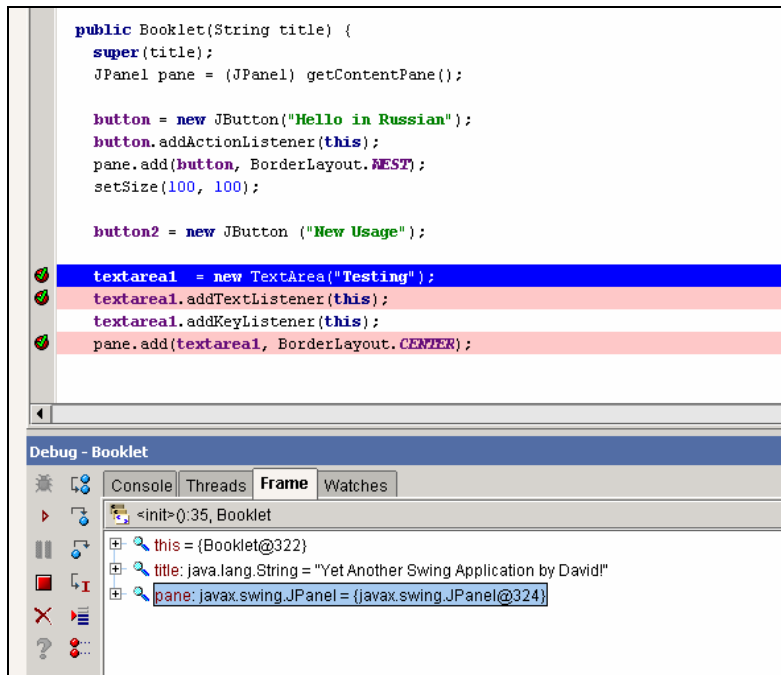
## Debugging

One of the most widely used and important features in any IDE is a fast working and effective debugger. When one is confronted with an unconventional or unplanned code result, it is this fundamental feature that we turn to most. This is why IDEA has integrated a JPDA-based debugger that is both extremely fast and easy to use.



Debugging 1.1

As figure *Debugging 1.1* shows, IDEA's debugger incorporates a very friendly user interface that allows you to quickly hunt down and debug code errors should they arise. You can select an entire class file, block of code, or just single lines of code to observe during the debugging process, and you can access the debugger's output in an easy to read tree-view event window as shown in figure *Debugging 1.2*.



*Debugging 1.2*

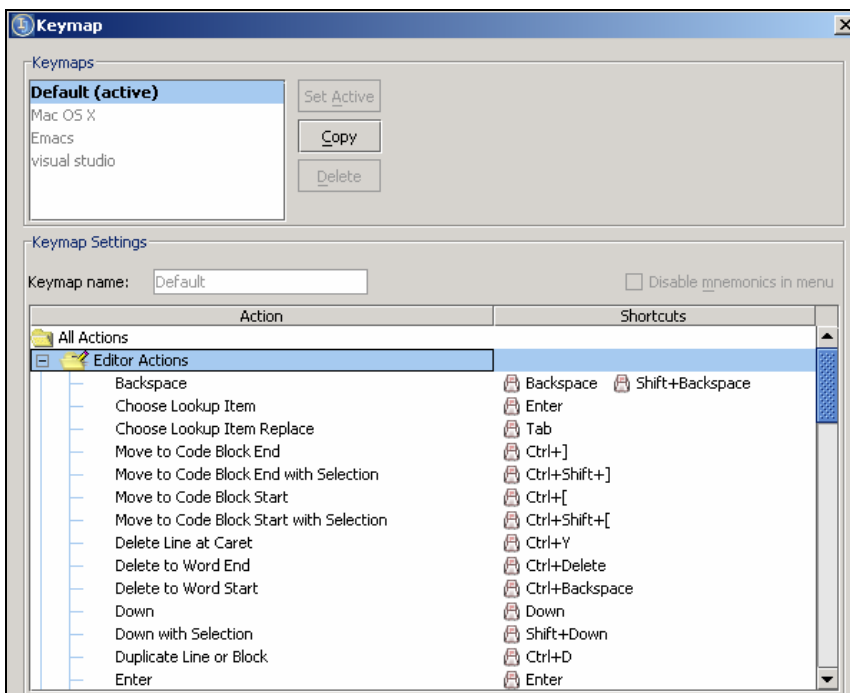
## Keymapping

One feature that is sure to help cut development time is IDEA's Keymapping feature that allows 100% access to IDEA's features with keystrokes, eliminating the need for you to take your hands off the keyboard. As shown in figure *Keymapping 1.1*, keystroke shortcuts are visible on the right side of drop down menus from the main toolbar.

Go to	Code	Refactor	Build	Run	Tools	Options
Class...						Ctrl+N
File...						Ctrl+Shift+N
Line...						Ctrl+G
Declaration						Ctrl+B
Implementation(s)						Ctrl+Alt+B
Type Declaration						Ctrl+Shift+B
Super Method						Ctrl+U
Next Highlighted Error						F2
Previous Highlighted Error						Shift+F2
Next Method						Alt+Down
Previous Method						Alt+Up
Back						Ctrl+Alt+Left
Forward						Ctrl+Alt+Right
Last Edit Location						Ctrl+Shift+Backspace

Keymapping 1.1

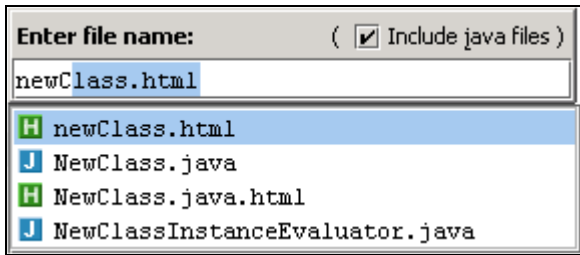
IDEA comes, by default, pre-loaded with shortcuts ready to use immediately after initial installation. However, the great thing about IDEA is that all keystrokes can be configured and customized to fit your own style and needs. As shown in figure *Keymapping 1.2*, you can access the keymap index and change the default keymap selections and replace them with your own customized versions.



Keymapping 1.2

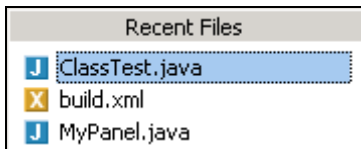
## Navigation

IDEA allows you to quickly open a class' source code by short-name, to navigate to any file in a project, as shown in figure *Navigation 1.1*, by name, to quickly find declarations and type declarations, implementations, and super methods.



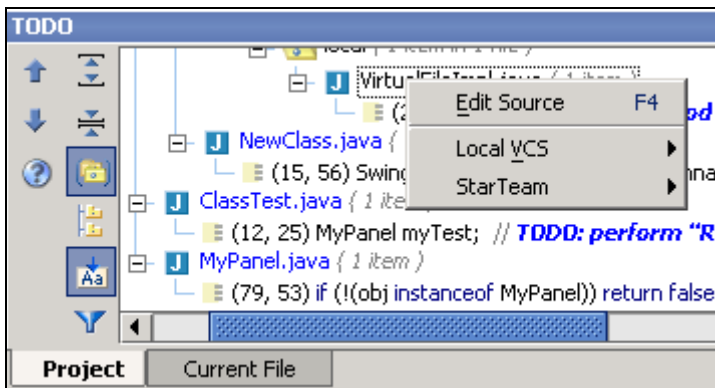
Navigation 1.1

In addition, you can quickly jump to the last change made in a file, or even view a list of the files you previously edited as shown in figure *Navigation 1.2*.



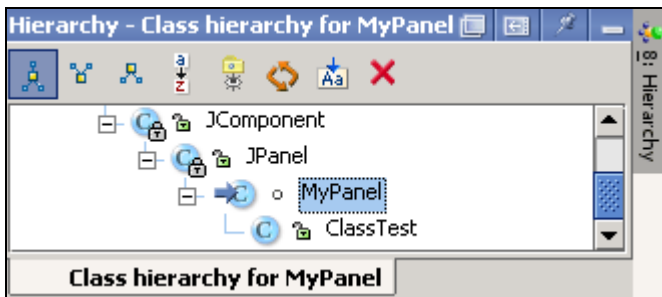
Navigation 1.2

Another feature to help you find and manage your code is the “Bookmark” and “To-Dos” functions. The Bookmark function allows you to mark lines of code in your project, and then quickly navigate to those locations quickly. The To-Dos function allows you to see your To-Dos comments in your source code in an easy-to-read tree-view panel as shown in figure *Navigation 1.3*.



Navigation 1.3

Lastly, IDEA also enables you to quickly browse class, interface, and method hierarchies and then transport you to those locations in the source code as shown in figure *Navigation 1.4*.



Navigation 1.4

## Code Inspection

If the previous section left the impression that IDEA's editor simulates a second set of hands, then it might be said that IDEA's powerful *Code Inspection* feature provides you with a second pair of eyes. This powerful code inspecting tool analyzes your source code for irregularities and informs you when your source code's design logic is "fuzzy." It has the ability to notify you when and where you have unassociated, unused, and redundant classes, interfaces, methods, and fields.

In addition to this design verification function, the Code Inspection feature is equipped with a powerful code implementation validation tool that reports where run-time exceptions might arise based upon certain conditions, varying from whether or not certain expressions have their execution results used or if execution flow never reaches certain statements.

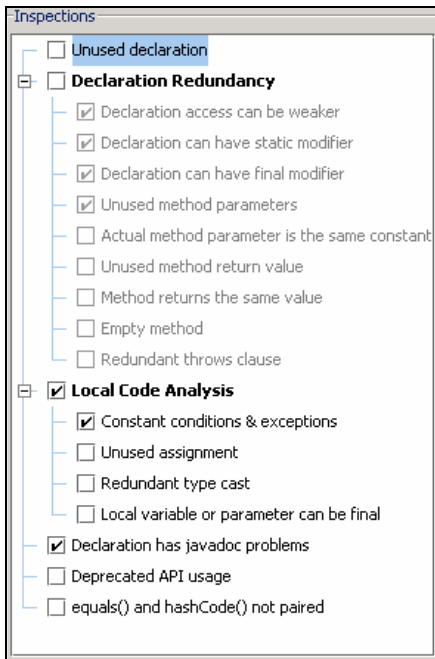
To get a taste of how powerful and useful the code inspection feature is, please take a look at the source code in figure *Code Inspection 1.1*. On line 29 we have commented out and noted a deliberate error we have thrown in the source code.

```
18 private class MyKeyListener extends KeyAdapter {
19
20 public void keyTyped(KeyEvent e) {
21     int keyCode = e.getKeyCode();
22
23     if (keyCode == KeyEvent.VK_F1) {
24         showHelp();
25     }
26     else if (keyCode == KeyEvent.VK_F2) {
27         saveCurrentFile();
28     }
29     else if (keyCode == KeyEvent.VK_F1) { // Error here. Should be KeyEvent.VK_F3
30         openNewFile();
31     }
32     else if (keyCode == KeyEvent.VK_F4) {
33         closeCurrentFile();
34     }
35 }
```

Example source code with conditional error

Code Inspection 1.1

Now, we invoke the Code Inspection control panel and select our desired analyze and search criteria as shown in figure *Code Inspection 1.2*, and then run the Code Inspection function.

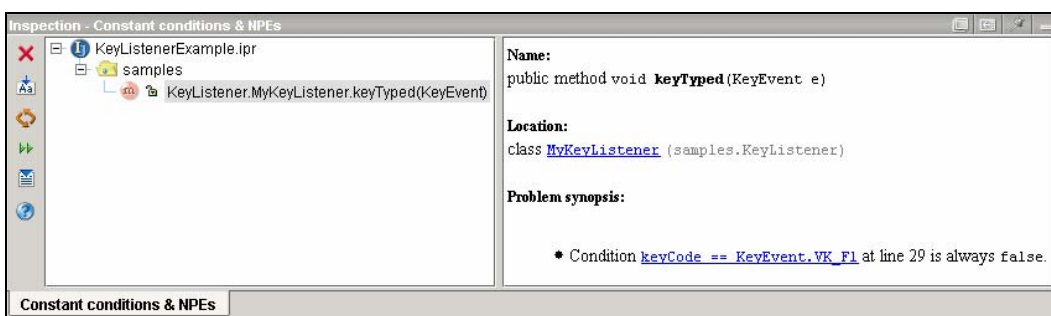


Code Inspection 1.2

Code Inspection Control Option Panel

Now, if a user were to compile the error-riddled source code just previously mentioned, a compiler would not throw an exception because the error it is not a Java error. You could deploy this application at this point and it would work, but not 100% correctly. A Q&A team might not find this error immediately and once they did find it, they would send it back to development and the developer would have to spend more time debugging the application, eventually fixing it after a lot of wasted (and costly) time.

This simple example of source code, of course, could be easily debugged manually without much fanfare; however, in a project with hundreds or even thousands of classes, interfaces, methods, and fields, would you want to look for such errors manually? Of course not! This is why you simply fire up the Code Inspection tool to do this job for you.



Code Inspection 1.3

Code Inspection Output Control Pane for Constant Conditions and NPE analysis

Once the code inspection function has completed its various selected analyses, the code inspection's analyzed results will be viewable in an easy to view tree-like navigation window in an output control panel as shown in figure *Code Inspection 1.3*. As noted previously, the Code Inspection function will not only perform the above mentioned inspection as noted in the example, but a multitude of various analyses that will dramatically reduce your chances of introducing errors into your projects. Not to mention that it will help you streamline your source code by ridding it of left-over development chaff.

# Refactoring

## What is Refactoring?

One of the most powerful features heaped upon developers and architects by divine providence is the almighty power of refactoring! Yes, refactoring is great, but why is it great? Even more so, what is refactoring? One of the industry's best, Martin Fowler, described refactoring as:

*“The process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. It's a disciplined way to clean up code that minimizes the chances of introducing bugs.”<sup>1</sup>*

This pleasant sounding paragraph pretty much sums up refactoring theory; however, in practice if done incorrectly, you can really wreck havoc on your code. This is why IDEA comes fully equipped with the most powerful refactoring tools available to date. Refactoring methods such as *Renaming*, *Extract Method*, *Change Method Signature*, *Extract Interface*, *Move*, and more are bundled with IDEA for more than 25 different refactoring methods *in toto*.

This section of the overview will therefore introduce some of the 25 plus refactoring methods provided by IDEA, with the intention of giving you a better understanding of when and why they are used and to see how IDEA makes invoking them as easy as pressing a few keys.

## Renaming

One of the most simple yet most used *and useful* refactoring methods integrated into IDEA is the *Renaming* refactoring function. Renaming allows you to change the name of any package, class, method, field or variable in a specific file or desired project. What is the reason for doing this? Simple: to clean up your code. When naming methods, for example, a good programmer will reveal the purpose of that method by its name as shown in figure *Renaming 1.1*.

Example 1.1:

```
class CarDatabase...
```

```
public String getCrTpMkCl();
```

Would be renamed to:

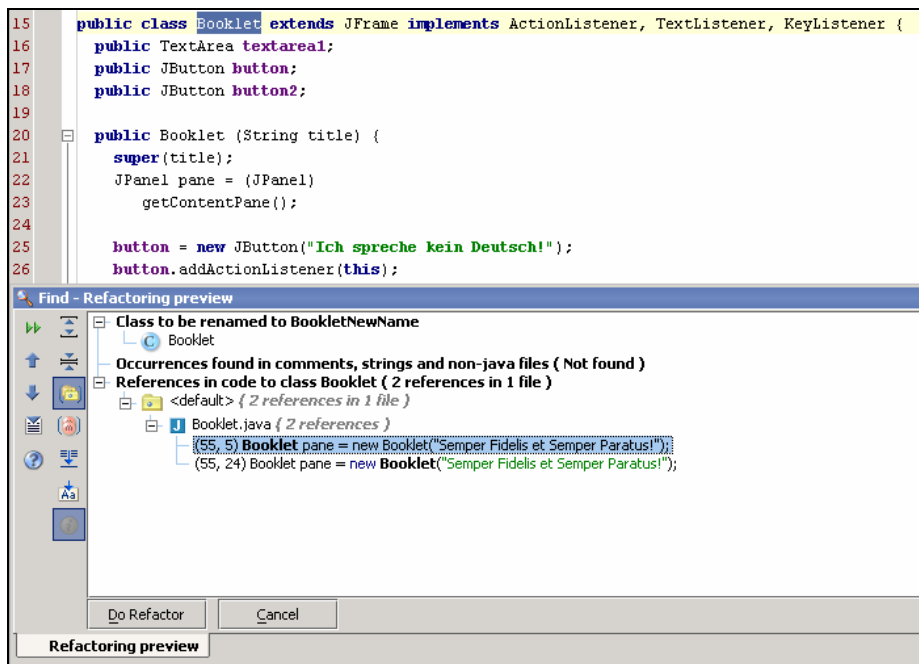
```
→ public String getCarTypeMakeColor
```

*Renaming 1.1*

Thus the renaming function allows you to change the name of that method and then automatically finds and corrects all references to this element (in both the working class and the rest of the entire project). As figure *Renaming 1.2* shows, an easy to read prompt will ask you to verify your changes – either by each individual instance or entire project.

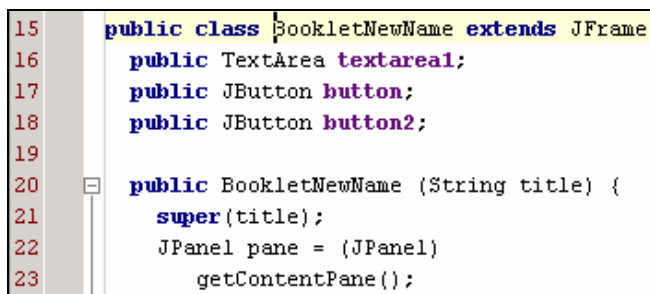
---

<sup>1</sup> Martin Fowler, *Refactoring: Improving the Design of Existing Code*, ISBN # 0201485672 (Addison-Wesley).



Renaming 1.2

Once you have determined the appropriate items to refactor, and you have refactored them by selecting the *Do Refactor* button, the concluding refactoring results are shown back into the editor as in figure *Renaming 1.3*.

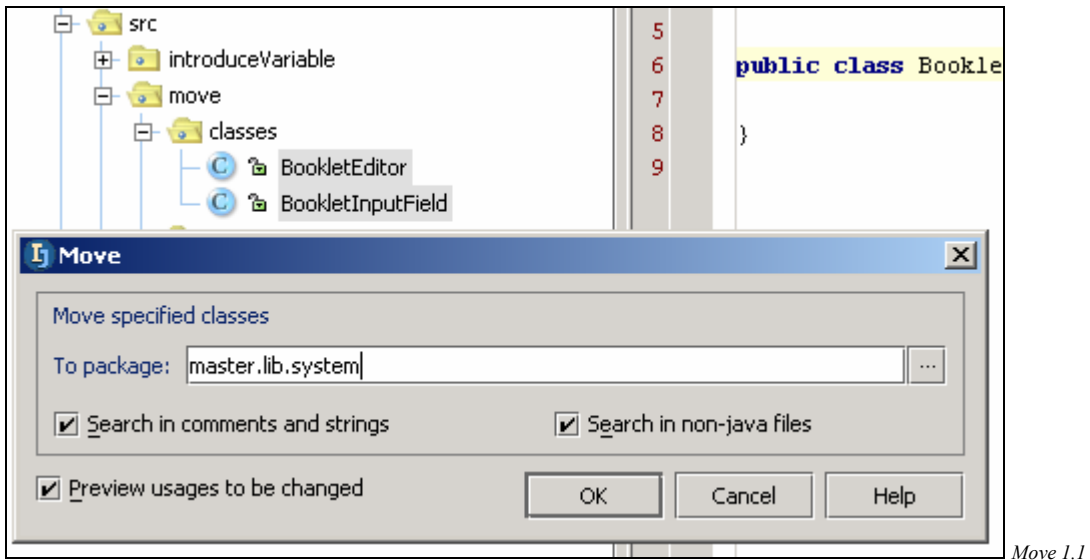


Renaming 1.3

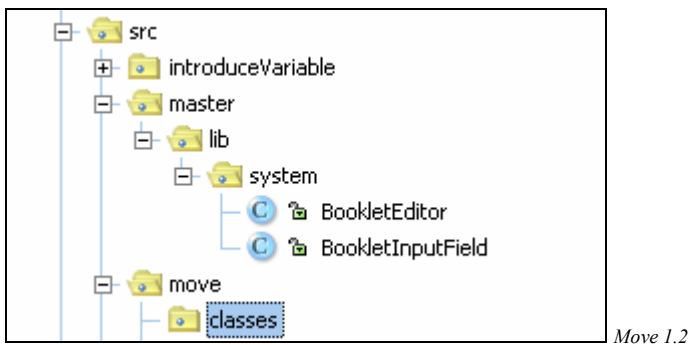
## Move

Along with the *Rename* function, IDEA's *Move* refactoring feature is another straight forward yet highly powerful and widely used refactoring method that allows you to correct, improve, or transfer misplaced responsibilities in source code without a lot of hassle. It also enables you to quickly move methods or static fields from one class into another, and in addition, you can also move entire classes or even entire packages into other packages all by invoking IDEA's *Move* function. This is done in an autonomic fashion that virtually eliminates your chances of introducing bugs into your code.

For example in figure *Move 1.1*, two Java class files (*BookletEditor* & *BookletInputField*) shown in the project view are easily moved into a new location (or a previously existing one) as shown in the "To package:" field. All references to these classes within the entire project will be changed to accommodate such changes.



Once the move process has been completed, you will see that the two previously mentioned Java class files have been moved from the “move.classes” package into a new package called “master.lib.system” as shown in figure *Move 1.2*.



As noted previously, in addition to moving classes between packages, you can move members of a class into a new class. As shown in figure *Move 1.3*, simply point the caret to the member you wish to move from the class `Booklet`, in this case `getDescription()` on line 14, and invoke the Move refactoring function (you can right-click your mouse and select **Refactor** | **Move** or press **F6** on the keyboard).

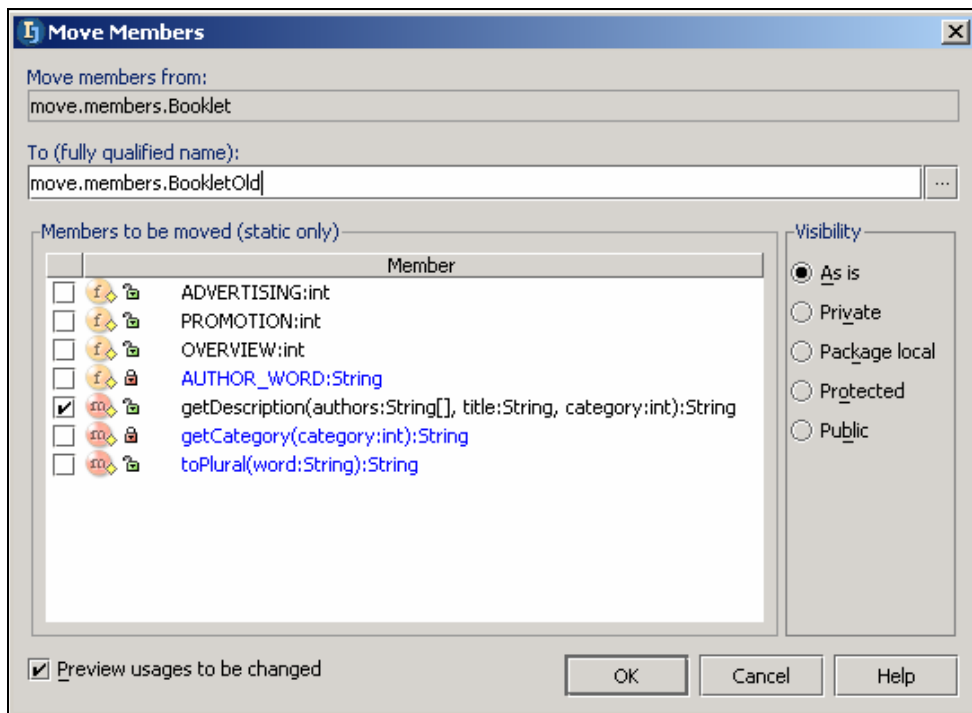
```

6 public class Booklet {
7     // Booklet categories
8     public static final int ADVERTISING = 0;
9     public static final int PROMOTION = 1;
10    public static final int OVERVIEW = 3;
11
12    private static final String AUTHOR_WORD = "author";
13
14    public static String getDescription(String[] authors, String title, int category) {
15        StringBuffer buffer = new StringBuffer();
16        buffer.append(title); buffer.append(", ");
17        buffer.append(authors.length == 1 ? AUTHOR_WORD : toPlural(AUTHOR_WORD));
18        buffer.append(" category:"); buffer.append(getCategory(category));
19        return buffer.toString();
20    }
21
22    private static String getCategory(int category) {
23        switch(category) {
24            case ADVERTISING: return "Advertising";
25            case PROMOTION: return "Promotion";
26            case OVERVIEW: return "Overview";
27            default: return "Unknown";
28        }
29    }

```

Move 1.3

Once the refactoring has been invoked, you will be shown a control dialog informing you of your selection, and more importantly, a list of other members highlighted in blue which will be required to be moved along with your initial selection as shown in figure *Move 1.4*.



Move 1.4

After the appropriate desired selections have been made, and the Move function has been completed (including your verification of the members to be moved), a new class will then be made in the newly mentioned location with your previously selected members to be moved as shown in figure *Move 1.5*.

```

10 public class BookletOld {
11     private static final String AUTHOR_WORD = "author";
12
13     public static String getDescription(String[] authors, String title, int category) {
14         StringBuffer buffer = new StringBuffer();
15         buffer.append(title); buffer.append(", ");
16         buffer.append(authors.length == 1 ? AUTHOR_WORD : toPlural(AUTHOR_WORD));
17         buffer.append(" category:"); buffer.append(getCategory(category));
18         return buffer.toString();
19     }
20
21     private static String getCategory(int category) {
22         switch(category) {
23             case Booklet.ADVERTISING: return "Advertising";
24             case Booklet.PROMOTION: return "Promotion";
25             case Booklet.OVERVIEW: return "Overview";
26             default: return "Unknown";
27         }
28     }
29
30     public static String toPlural(String word) {...}

```

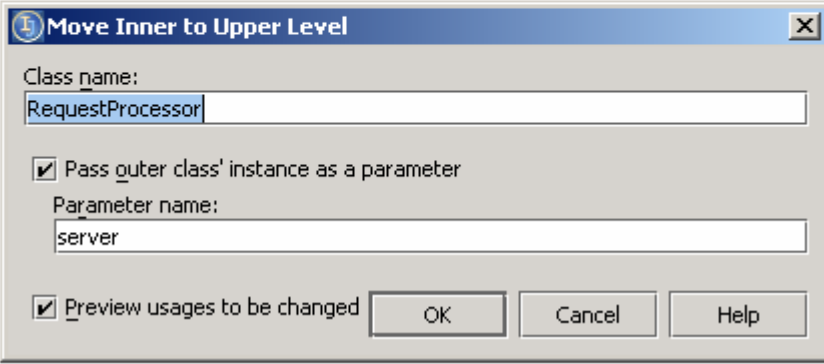
Move 1.5

You can also move “inner” classes and make them “outer” classes with the Move refactoring function. As shown in *Move 1.6*, the Move dialogue appears after the caret is placed on the desired inner class to move (in this case class RequestProcessor on line 8) and the Move refactoring function has been invoked.

```

4 public class Server {
5     /**
6      * Thread processing responses to request
7      */
8     class RequestProcessor implements Runnable {
9         private Request myRequest;

```



Move 1.6

After the Move refactoring procedure has been completed, a new class is born as shown in figure *Move 1.7*.

```

13 class RequestProcessor implements Runnable {
14     private Request myRequest;
15     private Server server;
16
17     public RequestProcessor(Server server, Request request) {
18         this.server = server;
19         myRequest = request;
20     }

```

Move 1.7

## Introduce Variable

Most of us eventually find ourselves in a situation where our code begins to grow into an untamed beast, and as it becomes more and more robust, it become difficult to understand. When this occurs, IDEA allows you to initiate another cool refactoring function called *Introduce Variable* (also called *Introduce Explaining Variable*). This function will simplify complicated expressions (or any part of one) by transforming them into a temporary variable with a name that expresses its function.

For example, figure *Variable 1.1* is your typical run of the mill expression.

```
3 public class StringUtil {
4     public static String toPlural(String word) {
5         if(word.length() == 0) return word;
6
7         if(word.charAt(word.length() - 1) == 'x' || word.charAt(word.length() - 1) == 's') {
8             return word + "es";
9         } else if (word.charAt(word.length() - 1) == 'y') {
10            return word.substring(0, word.length() - 1) + "ies";
11        } else {
12            return word + "s";
13        }
14    }
```

Variable 1.1

You can see that this expression is a little messy; however, if you do not think so then watch how IDEA makes it even clearer. As shown in figure *Variable 1.2*, the refactoring function *Introduce Variable* is invoked on the expression `word.charAt(word.length() - 1)`.

```
3 public class StringUtil {
4     public static String toPlural(String word) {
5         if(word.length() == 0) return word;
6
7         if(word.charAt(word.length() - 1) == 'x' || word.charAt(word.length() - 1) == 's') {
8             return word + "es";
9         } else if (word.charAt(word.length() - 1) == 'y') {
10            return word.substring(0, word.length() - 1) + "ies";
11        } else {
12            return word + "s";
13        }
14    }
```

Variable 1.2

As shown in figure *Variable 1.3*, the above mentioned complicated expression (and all of its occurrences) was changed into the expression `lastChar`.

```

3 public class StringUtil {
4     public static String toPlural(String word) {
5         if(word.length() == 0) return word;
6
7         char lastChar = word.charAt(word.length() - 1);
8         if(lastChar == 'x' || lastChar == 's') {
9             return word + "es";
10        } else if (lastChar == 'y') {
11            return word.substring(0, word.length() - 1) + "ies";
12        } else {
13            return word + "s";
14        }
15    }

```

Variable 1.3

Then, as a closer, we invoke introduce variable once again, this time on the expression `word.length() - 1` as shown in figure *Variable 1.4*.

```

3 public class StringUtil {
4     public static String toPlural(String word) {
5         if(word.length() == 0) return word;
6
7         int lastCharIndex = word.length() - 1;
8         char lastChar = word.charAt(lastCharIndex);
9         if(lastChar == 'x' || lastChar == 's') {
10            return word + "es";
11        } else if (lastChar == 'y') {
12            return word.substring(0, lastCharIndex) + "ies";
13        } else {
14            return word + "s";
15        }
16    }

```

Variable 1.4

Now, go back and look at figure *Variable 1.1* and compare it to our refactored expression in figure *Variable 1.4*. The former is a good hard numbered mathematical expression; the latter, a nice and easy to read word story problem. If you were working on a much larger project, and needed to find out what this expression did quickly, no doubt it would be the story problem and not the numbers which informed you the quickest. Not to mention, you code simply looks better.

## Extract Interface / Superclass

When the time comes to radically optimize both the code's readability and its design, *Extract Interface / Superclass* are the perfect refactoring functions to invoke. IDEA allows you to extract from classes or public interfaces public methods or static final fields into a new, single public interface or superclass that can be easily shared between multiple classes. This procedure removes the need to type repetitive code or use multiple implementations of the same object. As shown in figure *Extract I/S 1.1*, simply point the caret to a class or interface you wish to bundle into a new interface or superclass, and then select **Refactor | Extract Interface** from the main menu.

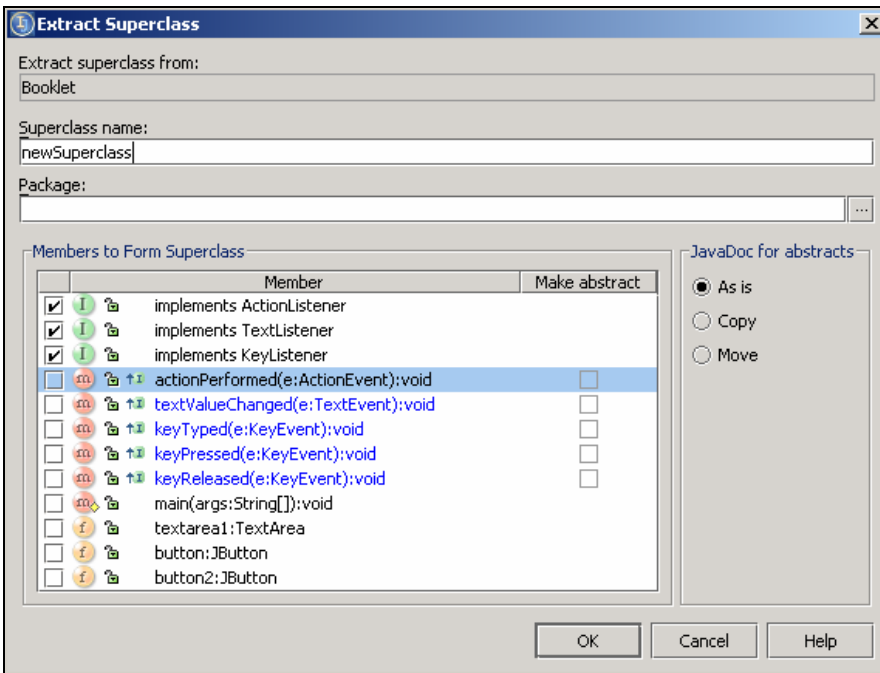
```

15 public class Booklet extends JFrame implements ActionListener,
16     public TextArea textareal;
17     public JButton button;
18     public JButton button2;

```

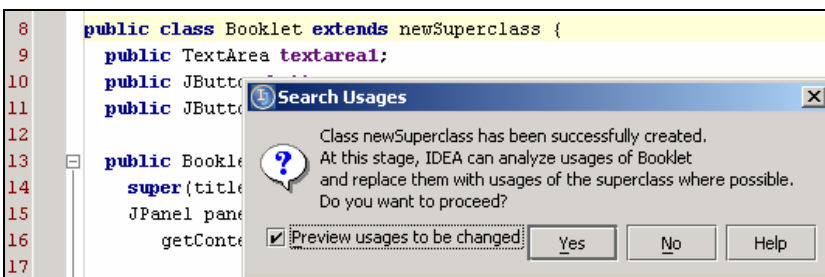
Extract I / S 1.1

Figure *Extract I / S 1.2* shows that once the refactoring procedure has been called, IDEA launches a popup console with various options allowing you to package the chosen interface or class, including their relevant methods and other objects, into the new interface.



Extract I / S 1.2

Once the refactoring procedure has been completed, IDEA will then prompt you for your permission to search the usages of the parent class to replace old usages with the new and improved ones as shown in figure *Extract Interface 1.3*. Like other refactoring functions in IDEA, a tree-view will be shown allowing you to approve your individual selections before making the changes final.



Extract Interface 1.3

An alternative to using *Extract Interface* is, depending on your situation of course, to invoke the refactoring function *Extract Superclass*. This function works in a similar fashion: You notice that you have two classes that basically contain the same code, and you are tired of updating the same bugs twice or improving the code in more than two places (and sometimes in 100s of places), and you want to eliminate this nuisance. IDEA will help you by automating the process of removing the common features used by varying classes, and package the contents into one shareable superclass.

## Extract Method

When one is faced with a block of characters that reads more like encryption than actual code, those using IDEA know they are fortunate to have the power to bring their coding universe back into order! The *Extract Method* refactoring function is one such enforcer of order that lets you extract code from one of these chaotic conglomerates of code and creates for you a new, unscathed and pristine method that is easily identifiable. In laymen terms, this means you can take a large method, and divide it up into multiple methods that are well defined and clearly marked – and – they are easily usable by other methods, because they are well defined.

For example, as shown in figure *Extract Method 1.1*, the `bookletToRename` method and its contents are a large cluttered mess. To fix this, just highlight the code that you wish to extract as a new, cleaner method, and invoke the Extract Method refactoring function.

```
6 public class BookletLibrary{
7     ArrayList myBooklets;
8
9     public void rename(String oldName, String newName) {
10
11         Booklet bookletToRename = null;
12         for (int i = 0; i < myBooklets.size(); i++) {
13             Booklet booklet = (Booklet) myBooklets.get(i);
14             if(booklet.getName().equals(oldName)) {
15                 bookletToRename = booklet;
16                 break;
17             }
18         }
```

Extract Method 1.1

*Extract cleaner and well defined methods from cluttered methods*

As shown in figure *Extract Method 1.2*, a new method has been created with the bulk of the “messy” contents being referenced somewhere else. Now the new method is easily identifiable and easily referenced by other methods and classes.

```
6 public class BookletLibrary{
7     ArrayList myBooklets;
8
9     public void rename(String oldName, String newName) {
10
11         Booklet bookletToRename = bookletNewMethod(oldName);
12     }
```

Extract Method 1.2

## Inline Method

The refactoring function *Inline Method* is the opposite of *Extract Method*. Then why would you want to use it, especially after the fact that we just told you how great the extract method function was? Simple. Sometimes you run into too many delegation indirections that clutter code and are simply confusing, so using inline method removes needless delegation and creates a responsible method! As shown in figure *Inline Method 1.1*, you see that there is some un-needed delegation in

the `getEnteredName` method.

```
7 public class BookletEditor extends JDialog {
8     JTextField myNameField;
9     JButton myOKButton;
10
11     private String getEnteredName() {
12         return myNameField.getText();
13     }
14
15     public String getBookletName() {
16         return getEnteredName();
17     }
18
19     private void validateOKButton() {
20         myOKButton.setEnabled(getEnteredName().length() > 0);
21     }
```

*Inline Method 1.1*

Just move the caret to the method you want to inline, in this case the `getEnteredName` method, and invoke the inline method function to remove the indirection chaff.

```
7 public class BookletEditor extends JDialog {
8     JTextField myNameField;
9     JButton myOKButton;
10
11     public String getBookletName() {
12         return myNameField.getText();
13     }
14
15     private void validateOKButton() {
16         myOKButton.setEnabled(myNameField.getText().length() > 0);
17     }
```

*Inline Method 1.2*

As shown in figure *Inline Method 1.2*, after the inline refactoring process has been completed, the needless indirection has been removed, the code has been streamlined, and no bugs have been introduced.

Just to note, a good idea to keep in mind is that you can use inline method as a precursor to utilizing the extract method function. What?!? Simply put, sometimes there are methods that are simply factored in a sloppy manner, and the quickest way to fix them is first to inline the sloppy code into one tidy method, and then to initiate extract method on this new and improved block of code to create finely tuned smaller methods that are much more friendly to share and easily identified.

## Encapsulate Field

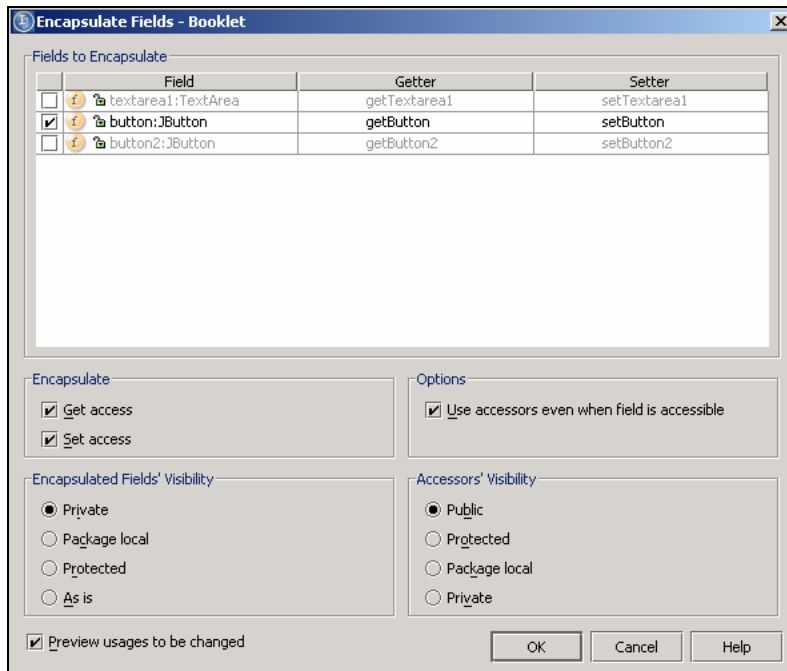
If you enjoyed playing hide-and-go-seek when you were a kid, then you are going to love the refactoring function *Encapsulate Field*. This function is utilized best when you want to make data in one object private and inaccessible from other public objects. In other words, you hide the contents of one object from other objects that may attempt to alter the former's behavior. As shown in figure

*Encapsulate Field 1.1*, you see that you simply point the caret at a targeted public field, select encapsulate field, and you are prompted with a relevant control console. In figure *Encapsulate Field 1.2*, once the Encapsulate Field has been invoked, it helps you create the appropriate **getter** and **setter** methods which hide the initial content of any selected field.

```
15 public class Booklet extends JFrame
16     public TextArea textArea1;
17     public JButton button;
18     public JButton button2;
```

*Encapsulate Field 1.1*

Select the public field you wish to encapsulate



*Encapsulate Field 1.*

IDEA has prompts you with an advanced multi-functional control panel to personalize your refactoring selection

## Change Method Signature

*Change Method Signature* is a refactoring method that encompasses a multitude of options for making a number of cosmetic and design changes to any desired method signature. IDEA enables you to initiate the following changes:

- Change method name
- Add parameter
- Remove parameter
- Reorder parameters
- Change return type
- Change parameter type

It is not our intention to cover these specific functions in greater detail in this overview, because by their names alone their functions are pretty obvious. Some of these above mentioned changes can be read in more detail in Martin Fowler's book on Refactoring previously mentioned in the Refactoring introduction page.

## J2EE Introduction

Creating component based J2EE modules has become the *de facto* standard in today's highly competitive, quickly changing and complex market of B2B, B2C, and B2E (Business-to-Everything else)! Picking the right tools for development can, literally, make the difference between making a multi-million dollar deadline and sinking a company into oblivion.

In any case, whether you are a lone developer or part of a large corporate development team, the success of the project is defined, to a greater or lesser degree, by its relations to its completion schedule and budget. Working with J2EE is no different. EJB, JSP, and Servlets are the bedrock of J2EE, with XML and HTML acting as mortar. IDEA gives you the power to utilize, organize, development, and launch this compendium of technologies in an intelligent, fast, efficient, and timely fashion.

IDEA doesn't pull any punches when it comes to J2EE development:

- Code Completion for JSP and XML
- Syntax and Error Highlighting in JSP/XML and EJB code
- JSP tag library support
- XML DTD / Schema completion / validation support
- EJB Setup / Create Integration Support, Code Assistance
- EJB Refactoring support

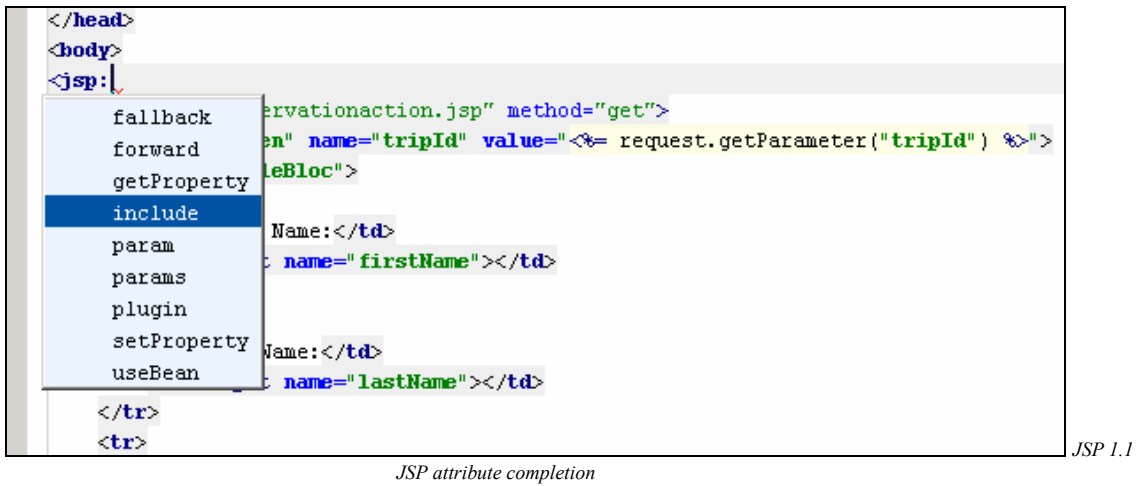
### JSP Development Support

JSPs (JavaServer Pages) are yet another integral part of J2EE development. If you have never implemented or coded JSPs before, here is a quick run down: JSPs allow web developers and designers to quickly deploy and easily maintain, information-rich, dynamic web content that leverages an existing business infrastructure.

JSPs can be used to build interfaces to e-commerce back-ends, intranet based project management and development tracking tools, and pretty much anything else that calls upon you to utilize Java packages, a HTML (or variants) based browser, and database connections. Of course, this is a quick and simplistic description of the immense and diverse functional capabilities that JSPs possess; however the premise should be quite clear: JSPs are invaluable in an enterprise development environment.

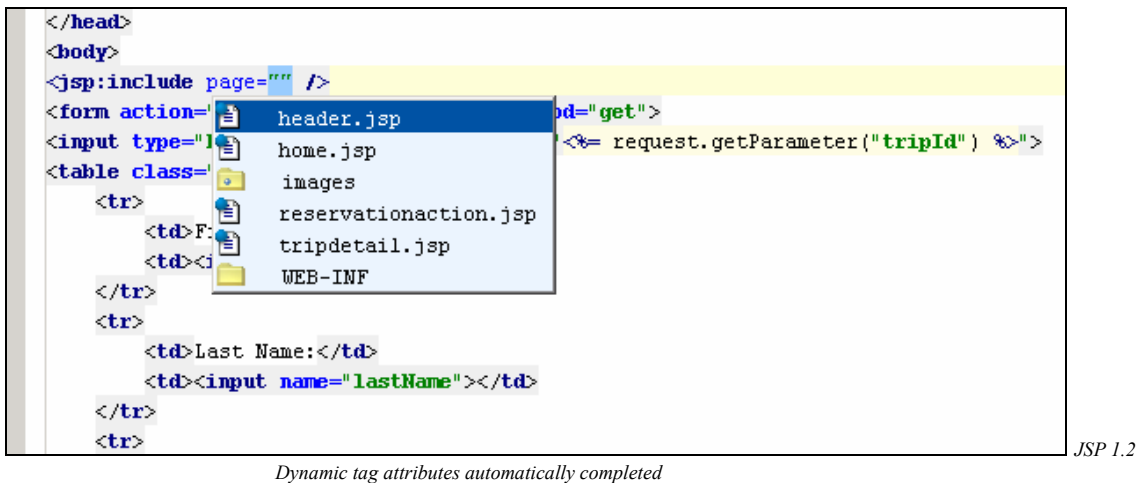
Having said this, if you are looking to utilize your limited time and resources to maximum efficiency, not to mention code for future scalability, then IDEA is the ideal development tool for JSP development. IDEA packs JSP development tool features more advanced than Batman's crime-fighting utility belt: JSP tag library and attribute code completion, code refactoring, error highlighting, debugging, and even JSP deployment capabilities.

IDEA's JSP code completion feature works in a similar fashion as the standard Java code completion feature. IDEA will automatically complete code when invoked to do so. For example, as shown in figure *JSP Development 1.1*, once you begin to code JSP tags, you simply invoke the code completion function – by selecting **CTRL + Space** – and a library of selections will appear.



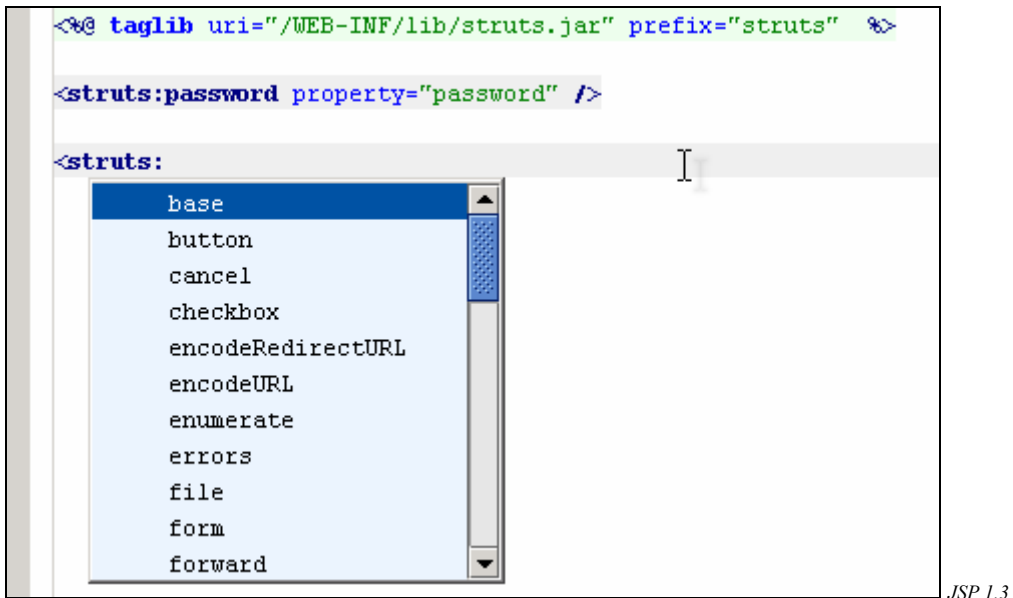
JSP attribute completion

Once the selected attribute has been chosen from the automated attribute list, IDEA will automatically complete this JSP tag by filling in all static data. As shown in figure *JSP Development 1.2*, any part of the tag that allows for multiple selections of data input, IDEA will intelligently offer more attributes based upon project content to automatically complete this dynamic data.



Dynamic tag attributes automatically completed

In addition to basic attribute completion, IDEA also enables developers to quickly add tag library selections, including TEI tags, at the stroke of a key as shown in figure *JSP Development 1.3*.

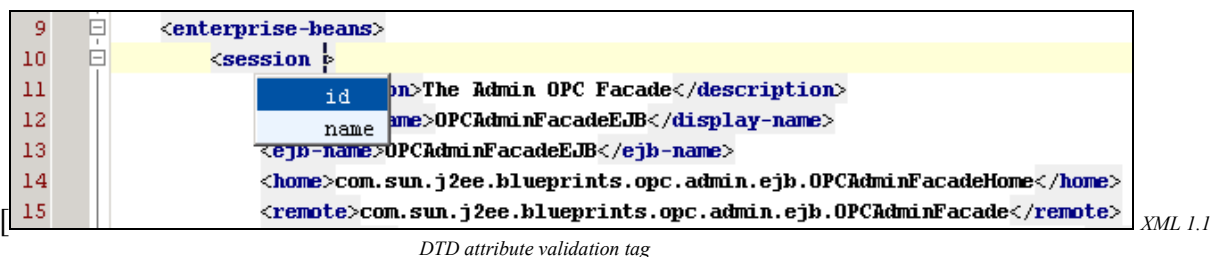


## XML Development Features

XML needs no introduction, or it shouldn't anyway. If you have ever done any extensive programming in Java, you have probably run into and used XML, if for nothing else, to create Ant `build.xml` files for rapid application deployment. For more extensive J2EE development, XML is utilized for multiple purposes: B2B (EDI, SOAP), Web service descriptors (WSDL), and even automated discovery and transaction services (UDDI, UNSPSC, NIC).

Whatever the case, if you are going to be deploying any Java applications coupled with XML, IDEA is going to dramatically enhance your ability to create applications at a faster and more efficient rate. How is that you might ask? Simple: Not only does IDEA's editor know Java, it also enables you to meet the demands of XML coding with smart editing features.

For example, IDEA allows you to quickly edit XML documents that support both DTD and Schema validation. As shown in figure XML Features 1.1, IDEA can digest any given DTD's specification and automatically include these special attributes into the editor's intelligent XML attribute completion function.



In figure XML 1.2, schema specifications, like DTDs, can be appropriated by IDEA's intelligent editor for faster and more accurate automated attribute-tag completion.

```

1 <?xml version='1.0' encoding='UTF-8' ?>
2 <xs:schema targetNamespace="http://www.w3.org/2001/XMLSchema" blockDefault="#all"
3 <xs:a
4 xs:annotation
5 xs:attribute
6 xs:attributeGroup
7 part 2 version: Id: XMLSchema.xsd,v 1.49 2001/10/25 10:25:41 ht Exp
8 </xs:documentation>
9 </xs:annotation>

```

Schema tag-attribute validation

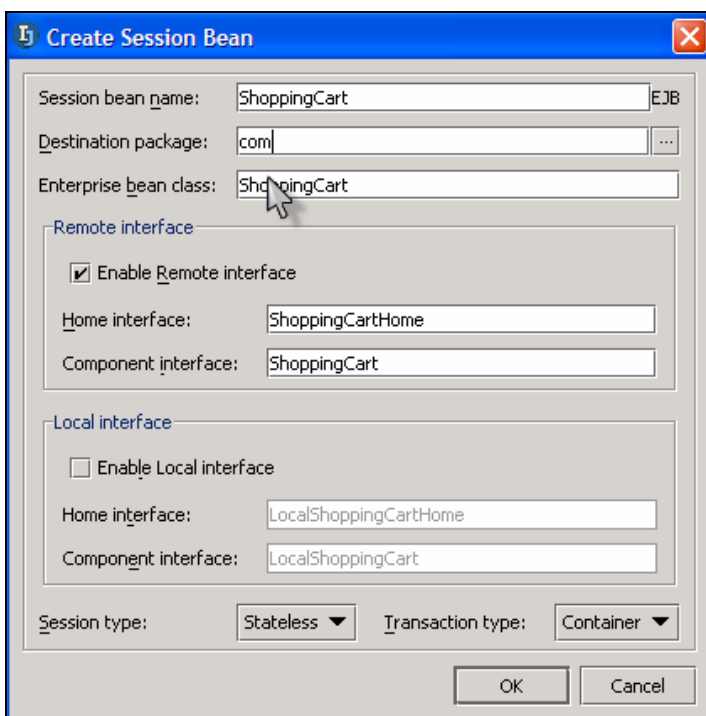
XML 1.2

In addition to the aforementioned features, IDEA also incorporates a XML error high-lighting function. As shown in figure *XML 1.3*, if an error is made in the XML code, IDEA will color-code the errors making them easy to find and fix.

### EJB Integration

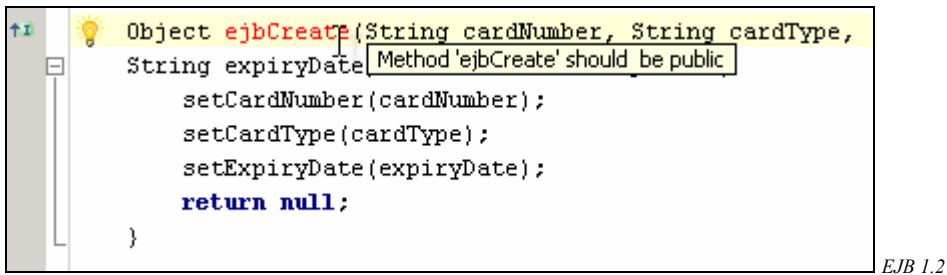
For those developers who are looking for a set of tools to aid you in much more complicated, robust, and over all time consuming enterprise centered development projects – or – in other words, you need to crank out a plethora of EJBs under a dead-line or simply want to create EJBs that are flexible, scalable, and that work quickly, then IDEA’s EJB support is what the doctor ordered.

For starters, with IDEA you can create new beans to get you up and going quickly as shown in figure *EJB 1.1*.

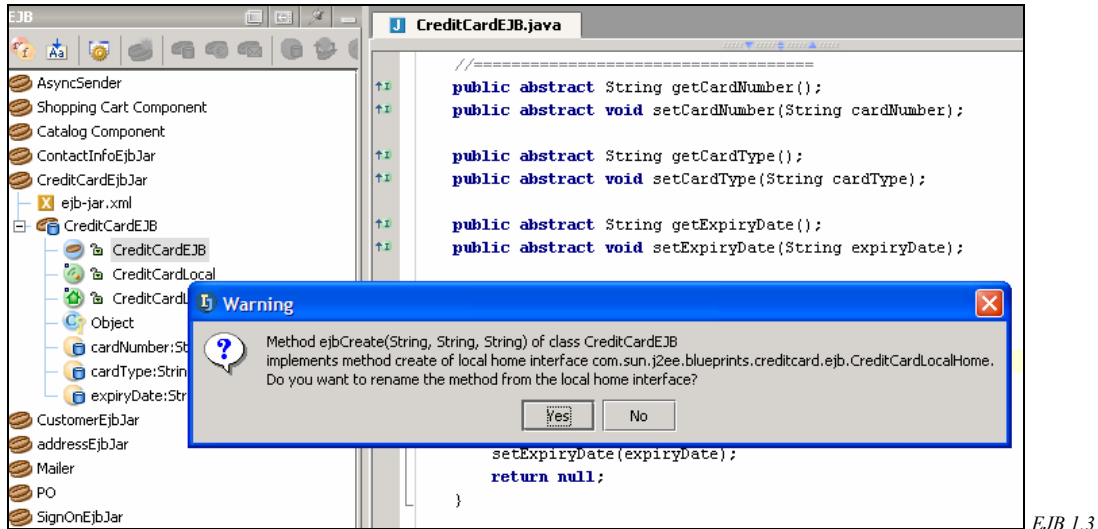


EJB 1.1

In addition, IDEA has an integrated error high-lighting function for EJBs. Red is the magic color: major errors that make deploying the EJB impossible will be shown in red, including compatibility errors and errors in any of the deployment descriptors.



IDEA's refactoring support also supports EJB development as shown in figure *EJB 1.3*.



## Collaboration Tools

If you have read through the overview up to this point, it is probably safe for us to assume that you are now pretty familiar with IDEA and have a grasp of the firepower it packs in regards to the multitude of powerful features and functions that in short, among a gazillion other things, hastens development, cleans up your code, and increases productivity. However, one should never expect IDEA to rest on its laurels, because being content is about the last thing the makers of IDEA have on their minds.

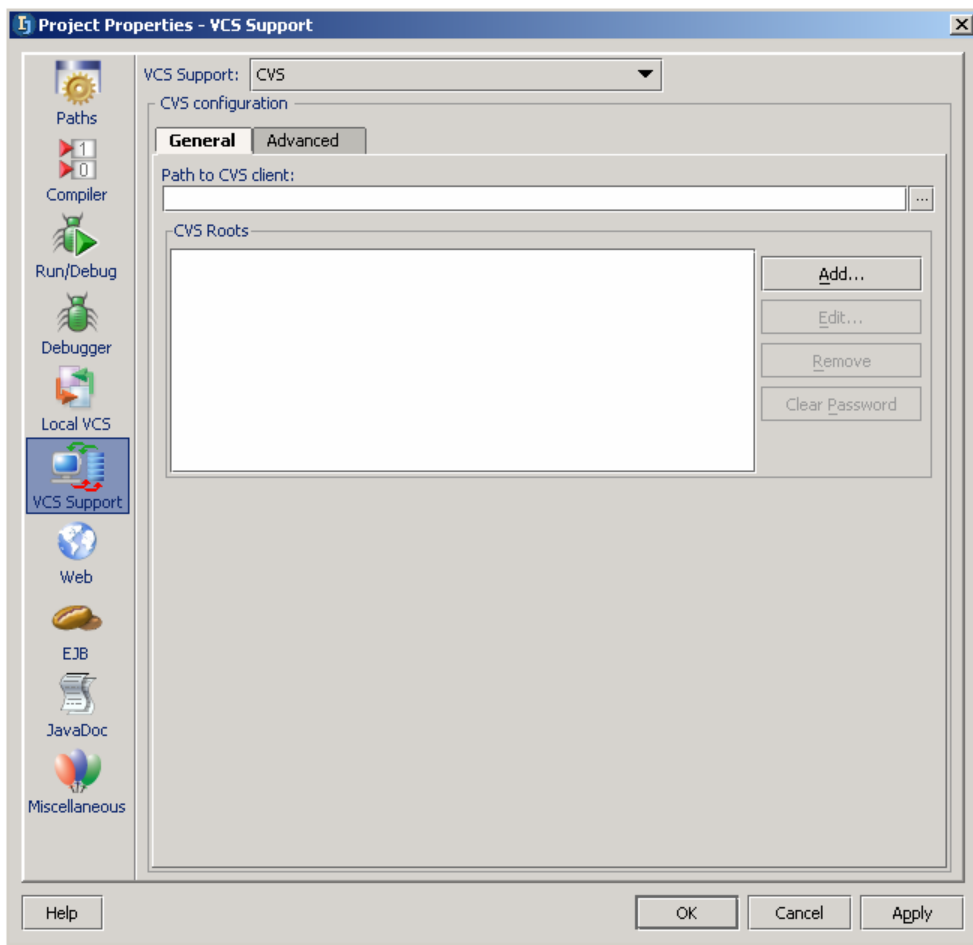
IDEA has evolved into the kind of IDE that simply cannot avoid incorporating a good thing, and therefore, IDEA has been forged to integrate seamlessly with some of today's most popular and most important open source development tools the world has come to know.

This section will briefly cover these various tools and point you in the right direction in regards to where you can download them.

### CVS Integration

IDEA not only helps you develop and design code quicker, more intelligently, and with greater ease - it also helps you manage and organize your projects for greater work efficiency. IDEA comes equipped with a powerful CVS (Concurrent Version System) to help you manage revisions to any project's source code files. As shown in figure *CVS Integration 1.1*, IDEA packs a user friendly CVS administration console to help you immediately begin tracking and backing up your documents

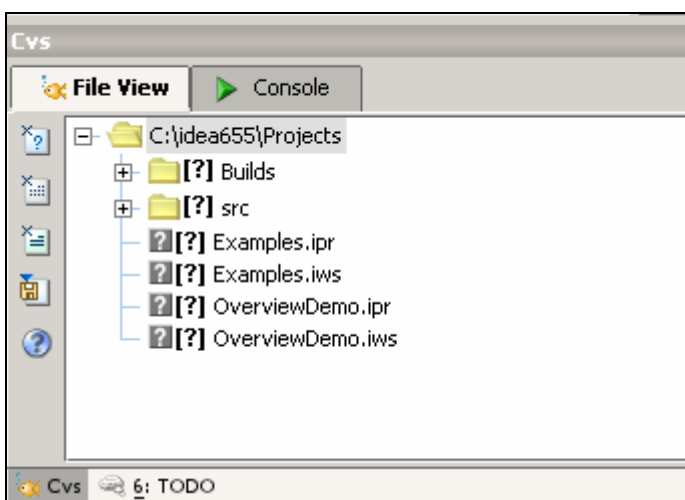
and source code.



CVS Integration 1.1

CVS set-up administration console

Once the CVS integration feature has been enabled, simply fill in the requisite fields in the set-up administration console and then apply your settings. After the console has been set up, you will see “CVS” highlighted on the tab on the bottom feature bar shown in figure *CVS Integration 1.2*, which indicates that the CVS is now available for use.



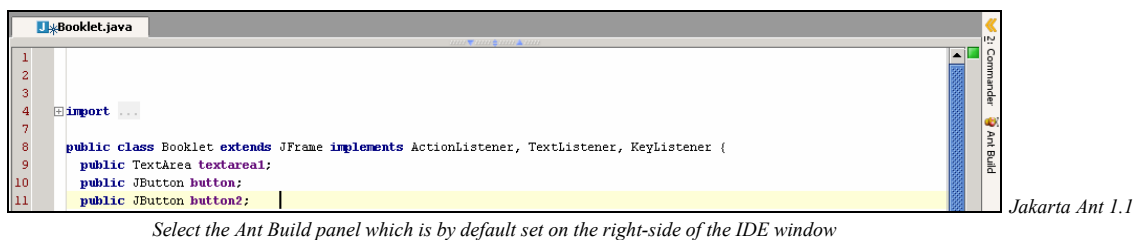
CVS Integration 1.2

Easy to read tree-view of CVS submitted files

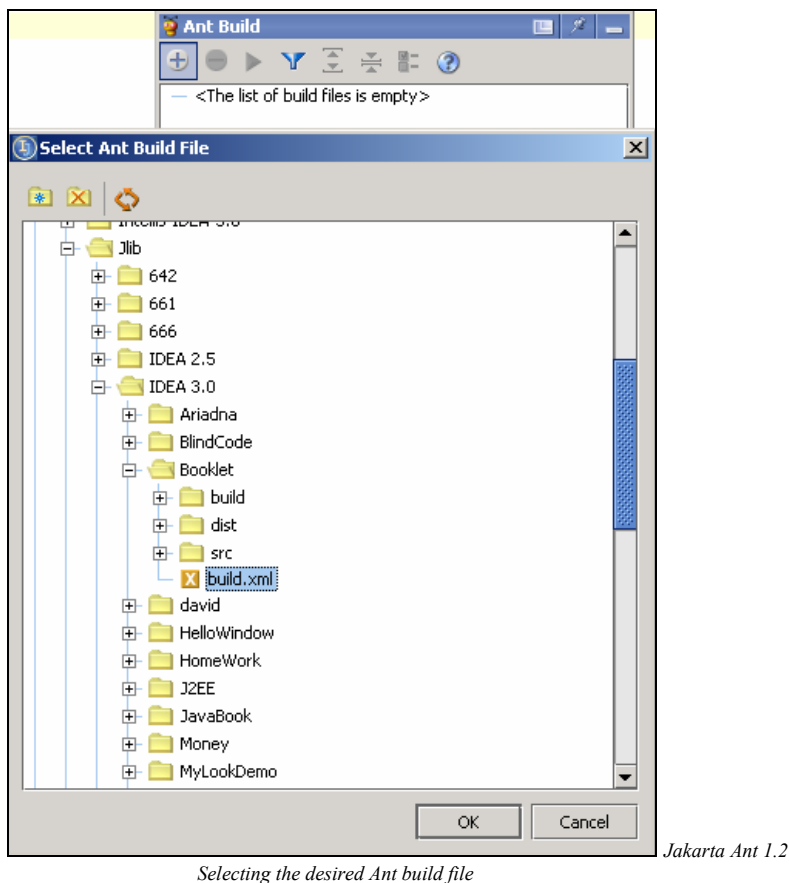
## Jakarta Ant

If you are a Java developer and you have never used Jakarta Ant (god forbid you haven't heard about it), it is probably a good time for you to get familiar with this open source, freely downloadable Java based build tool. After all, would you really enjoy repeating the same remedial time-consuming task day after day if you could avoid it? We didn't think so, and that is why IDEA has integrated Ant. Ant is one of the most popular and widely used build tools on the planet, and when used along with IDEA, development and deployment time respectively becomes almost frighteningly too easy. Take it for a test run and see for yourself.

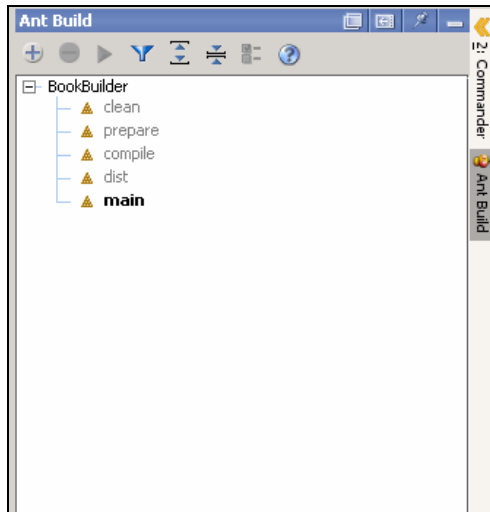
Open a project in IDEA, and then open the **Ant Build** panel shown in figure *Jakarta Ant 1.1*.



As shown in figure *Jakarta Ant 1.2*, once the Ant Build panel is open, simply select the + menu button and add the build file you want to initiate your build process.



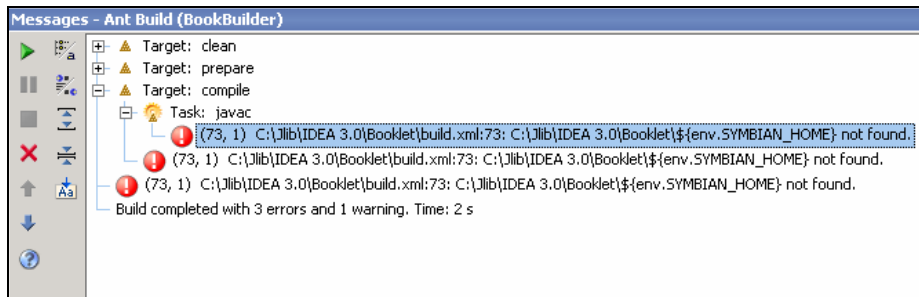
As shown in figure *Jakarta Ant 1.3*, once you have selected your build file (see figure *Jakarta Ant 1.2*), a navigation window will appear outlining the build file's sequence of events that it will initiate during the build process. To initiate the build process, just select the run menu item. Ant will begin its build process, and if any errors occur, IDEA's event window will display a detailed log of the final build results.



*Jakarta Ant 1.3*

*View of selected build file's contents*

As shown previously in prior sections, IDEA's standard tree-navigation window shows you the error messages in its output if any errors are thrown. In figure *Jakarta Ant 1.4*, you can see these throw error messages and quickly navigate to their respective locations in the source code, make corrections, and start the build process again.



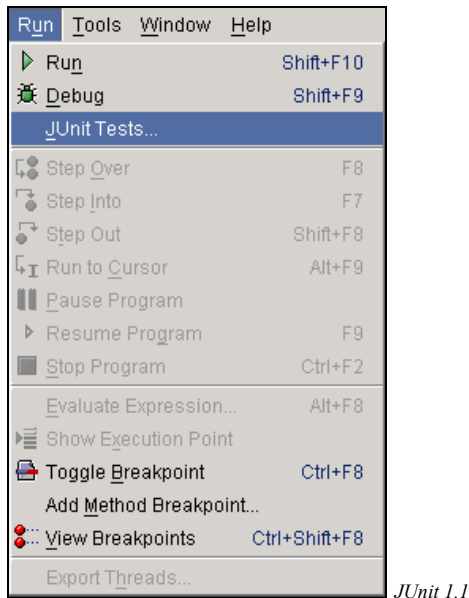
*Jakarta Ant 1.4*

## JUnit

Those who like to do things right the first time, will no doubt appreciate JUnit's integration into IDEA. JUnit is an open source testing framework for Java that provides users with a simple yet powerful way to express a written code's intention and then verify that code's behavior according to its associated intention. This is done by initiating unit tests (each test is normally associated with a specific class), and then testing the output of each unit.

This is done to ensure that all of your objects are doing what they are supposed to be doing. When each object does what it is supposed to be doing, then you won't have to waste time later debugging. It is a pretty straight forward philosophy.

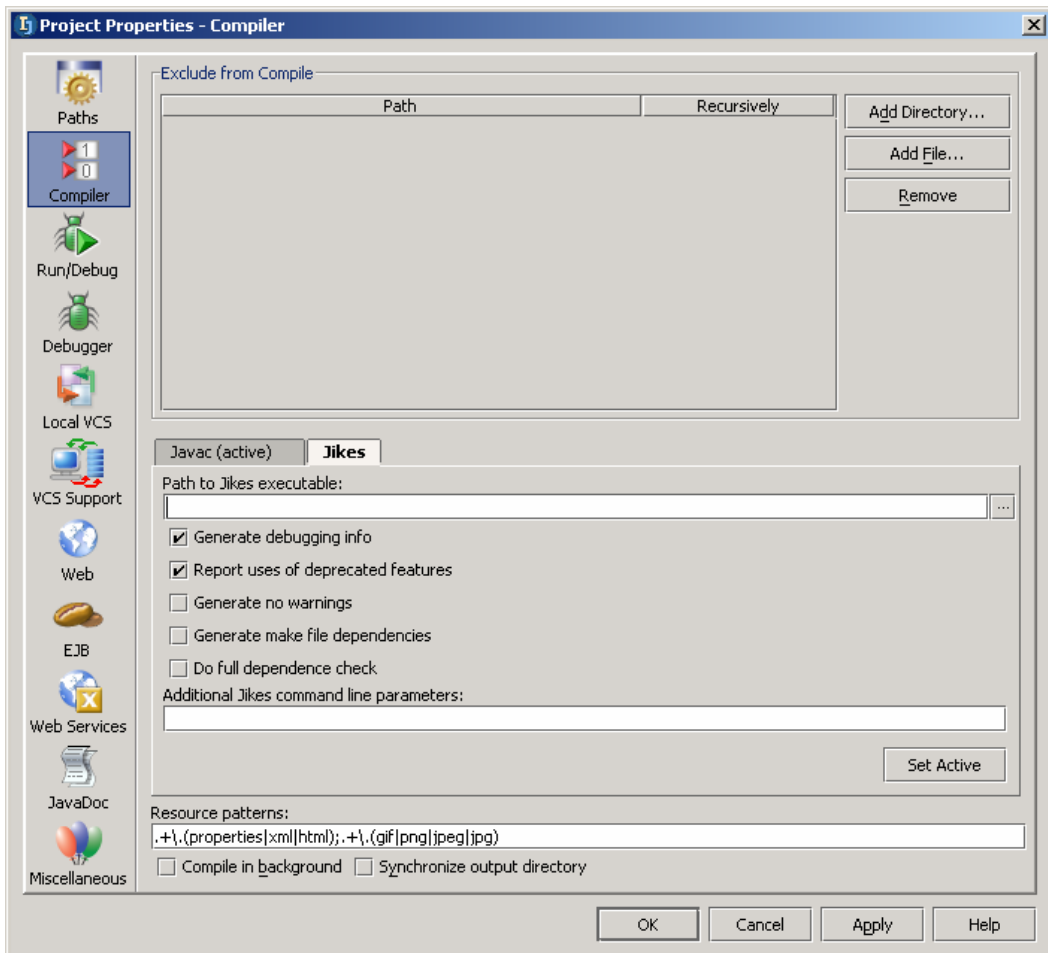
It is for this highly practical (and rather obvious) reason that IDEA has integrated JUnit. You can run unit test directly from IDEA controlled by an easy to configure JUnit control panel. As shown in figure *JUnit 1.1*, you can easily run a unit test from IDEA's tool bar menu. You just simply have to invoke a test case method near your intended target object and the results of the test will be visible in the output window.



## Jikes

If you require a Java compiler with a little more juice and packs the compilation speed of a super-sonic jet, then Jikes is the compiler you need to use. Jikes is a Java source to bytecode compiler written in C++ that starts and compiles faster than your standard javac compiler. However, this open source IBM production is noted not just for its speed, but also because it has the uncanny ability to offer alternative selections to misspelled identifiers and it is equipped with an incremental compiling feature along with an automatic makefile generation function. This is a jet that comes fully-armed!

If you want to test drive Jikes through IDEA, you won't find setting it up a problem. Simply download and install Jikes, change the *Compiler* properties to your liking, and set Jikes as your active compiler and point to its path. As shown in figure *Jikes 1.1*, the control console is pretty straight forward.



Jikes 1.1

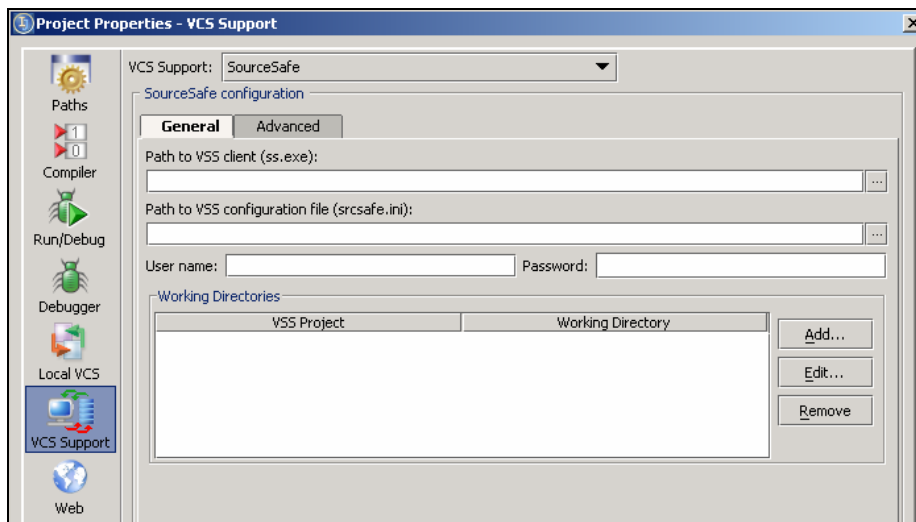
Select "Jikes" tab and point to Jikes path to set up

## Visual SourceSafe

When multiple people form a work group with specific goals in mind, regardless of the endeavor, their success nearly always depends on their ability to communicate and work together in a concerted and effective effort to achieve those common goals. When this scenario is applied to the development world, we see that projects are completed timely and efficiently when project managers, developers, and other essential parts of these groups are well informed of each other's progress.

This is why IDEA was developed to be easily integrated with Microsoft's Visual SourceSafe, an industry leading document management and versioning control system application.

As shown in figure *Visual SourceSafe 1.1*, IDEA incorporates an easy to use and set up SourceSafe control panel allowing you to quickly set up and begin to utilize your SourceSafe installation within minutes.



Visual SourceSafe 1.1

## Resources

JUnit: <http://www.junit.org>

Jakarta Ant: <http://jakarta.apache.org/ant/index.html>

Jikes: <http://oss.software.ibm.com/developerworks/opensource/jikes/>

Visual SourceSafe: <http://msdn.microsoft.com/ssafe/>

CVS: <http://www.cvshome.org>

## Open API

Not only is IDEA equipped with the development features mentioned previously in this book, but IDEA has also launched an Open API initiative that allows third party developers to seamlessly integrate their own tools into IDEA. Therefore, there is a good chance that if IDEA doesn't have a standard feature you would like to utilize, it will probably be coming in the near future.

If you are a third party developer, you will be happy to know that your application's functions will be able to be called directly from IDEA. In addition, you will be able to incorporate a number of IDEA's features directly into your own application. From a user perspective, this Open API is going to make IDEA more versatile, because once you have IDEA, you will have access to a whole new "eco-system" of development tools that accommodate and enhance IDEA's already second-to-none capabilities.

IDEA users are encouraged to check out: [www.intellij.org](http://www.intellij.org). Here you can find, among other things, a large and growing list of the newest plug-ins for IDEA, most of which are free to the public and open source.

## Default Keystroke Index

The following is a list of default keystrokes that initiate many key features in IDEA. However, you may modify these keystrokes' associations to your liking at anytime.

### Editing

Alt + Enter	Show Intention Actions
Alt + Q	Context Info
Ctrl + Alt + F11	Full Screen
Ctrl + F4	Close Active Editor
Ctrl + Space	Basic code completion ( <i>the name of any class, method or variable</i> )
Ctrl + Shift + Space	Smart code completion ( <i>filters the list of methods and variables</i> )
Ctrl + Alt + Space	Class name completion ( <i>the name of a class in the project or libraries</i> )
Ctrl + W	Select successively increasing code blocks
Ctrl + /	Comment with line comment
Ctrl + Shift + /	Comment with block comment
Ctrl + P	Parameter Info ( <i>within method call arguments</i> )
Ctrl + O	Override Methods
Ctrl + I	Implement Methods
Ctrl + Q	Quick JavaDoc
Alt + Insert	Generate code ... ( <i>Getters, Setters, Constructors, more...</i> )
Ctrl + Alt + T	Surround with ... ( <i>else / if, try / catch, for, Runnable, more...</i> )
Ctrl + Z	Undo
Ctrl + Shift + Z	Redo
Tab	Shift the selected lines to the right
Shift + Tab	Shift the selected lines to the left
Ctrl + Shift + F7	Highlight usages within file
Alt + F3	Incremental search within a file
Shift +	Click on file tab to close a file in the editor
Ctrl + Alt + O	Optimize Imports
Ctrl + Alt + I	Auto-indent Lines
Ctrl + Alt + L	Reformat Code
Ctrl + D	Duplicate unselected line or selected block
Ctrl + C	Copy unselected line or selected block to buffer
Ctrl + X	Cut unselected line or selected block to buffer
Ctrl + V	Paste unselected line or selected block from buffer
Ctrl + Shift + V	View buffer content to paste
Ctrl + Y	Yank the unselected line or selected block, does not affect the buffer

Ctrl + Shift + J                      Join Lines

## **Compile and Run**

Ctrl + F9	Make Project
Ctrl + Shift + F9	Compile "Variable.java"
Shift + F10	Run
Shift + F9	Debug

## **Debugging**

F8	Step Over
F7	Step Into
Shift + F8	Step Out
Alt + F9	Run to Cursor
Alt + F8	Evaluate Expression
Ctrl + F2	Stop Program
F9	Resume Program
Ctrl + F8	Toggle Breakpoint
Ctrl + Shift + F8	View Breakpoints

## **Navigation**

Alt + F1	Select View
Ctrl + N	Go to Class
Ctrl + Shift + Backspace	Last Edit Location
Ctrl + Shift + N	Go to File
Ctrl + E	Recent files popup
Ctrl + G	Go to Line
Ctrl + H	Type Hierarchy
Ctrl + Shift + H	Method Hierarchy
Ctrl + Alt + H	Call Hierarchy
Ctrl + Alt + Left	Navigate one item back
Ctrl + Alt + Right	Navigate one item forward
Ctrl + B	Go to Declaration
Ctrl + Alt + B	Go to Implementation(s)
Ctrl + Shift + B	Go to Type Declaration
Ctrl + U	Go to Super Method

Alt + Up	Go to Previous Method
Alt + Down	Go to Next Method
F2	Next Highlighted Error
Shift + F2	Previous Highlighted Error
F4	Edit Source
Ctrl + Enter	View Source
Alt + F7	Find Usages ...
Ctrl + F	Find
F3	Find Next
Shift + F3	Find Previous
Ctrl + R	Replace Text
Alt + F3	Incremental Search
Ctrl + Shift + F	Find in Path
Ctrl + Shift + R	Replace in Path
F11	Toggle Bookmark
Shift + F11	Show Bookmarks

## *Refactoring*

F5	Copy
F6	Move
Alt + Delete	Safe Delete
Shift + F6	Rename
Ctrl + F6	Change Method Signature
Ctrl + Alt + N	Inline
Ctrl + Alt + M	Extract Method
Ctrl + Alt + V	Extract Variable
Ctrl + Alt + F	Introduce Field
Ctrl + Alt + C	Introduce Constant
Ctrl + Alt + P	Introduce Parameter

## *Live Templates*

Ctrl + Alt + J	Surround with Live Template
Ctrl + J	Insert Live Template

<b>inst</b>	Checks object type with <code>instanceof</code> and down-casts it
<b>itar</b>	Iterate elements of array
<b>itco</b>	Iterate elements of <code>java.util.Collection</code>
<b>iten</b>	Iterate elements of <code>java.util.Enumeration</code>
<b>itit</b>	Iterate elements of <code>java.util.Iterator</code>

<b>itli</b>	Iterate elements of <code>java.util.Elements</code>
<b>ittok</b>	Iterate tokens from <code>String</code>
<b>itve</b>	Iterate elements of <code>java.util.Vector</code>
<b>lst</b>	Fetches the last element of an array
<b>mn</b>	Sets lesser value to a variable
<b>mx</b>	Sets greater value to a variable
<b>psf</b>	<code>public static final</code>
<b>psfi</b>	<code>public static final int</code>
<b>psfs</b>	<code>public static final String</code>
<b>psvm</b>	<code>main()</code> method declaration
<b>ritar</b>	Iterate elements in an array in reverse order
<b>serr</b>	Prints a string to <code>System.err</code>
<b>sout</b>	Prints a string to <code>System.out</code>
<b>soutm</b>	Prints current class and method names to <code>System.out</code>
<b>soutv</b>	Prints a value to <code>System.out</code>
<b>St</b>	<code>String</code>
<b>thr</b>	<code>throw new</code>
<b>toar</b>	Store elements of <code>java.util.Collection</code> into an array