

IntelliJ **IDEA** 4.0

Develop with pleasure!



OVERVIEW

Copyright © 2000-2004 JetBrains, Inc. All rights reserved.

© 2000 - 2004 JetBrains, Inc. All rights reserved.

JetBrains, IntelliJ, 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 consent of JetBrains, Inc.

Contact

International Sales

JetBrains s.r.o.
Klanova 9/506
14700 Prague
Czech Republic

Phone: +420 2 4172 2501
Fax: +420 2 6171 1724
sales@jetbrains.com

Evaluation support:
support@jetbrains.com

North American Sales

East coast

33-C South Main Street
Medford NJ 08055

Phone: +1 (609) 714 7883
Fax: +1 (609) 714 7886
sales.us@jetbrains.com

West coast

1670 So. Amphlett Blvd. -
Suite 214
San Mateo, CA 94402

Phone: +1 (650) 378 8571
Fax: +1 (650) 378 8591
sales.us@jetbrains.com

Canada

207 Barnes Street
Kempville ON K0G 1J0

Phone: +1 (613) 258 0575
Fax: +1 (613) 258 5979
sales.us@jetbrains.com

On the Web

Corporate Website:
IntelliJ IDEA Website:
IntelliJ Community:
IntelliJ Technology Network:

www.jetbrains.com
www.jetbrains.com/idea
www.intellij.org
www.intellij.net

Acknowledgements

Special thanks to all of the community members who offered suggestions and constructive feedback for this Overview, and who spent countless hours of their valuable time bug hunting and giving critical feedback to help make IDEA what it is today, this includes but is not limited to: Alain Ravet, Glen Stampoulzidis, Guillaume Laforge, Jacques Morel, Jonas Kvarnström, Oleg Danilov, Robert F. Beeger, Robert S. Sfeir, Thomas Singer, and the Team JetBrains members not mentioned specifically herein.

About IntelliJ IDEA

IntelliJ IDEA is the industry's smartest and most user friendly Java IDE in the market place today. No other IDE enhances developer productivity like IDEA, a fully loaded IDE with industry setting refactoring tools, intelligent code editing assistance, a diverse array of J2EE development features and integrations for rapid web-application development, a powerful Code-Inspection tool, advanced CVS support, an Open API for third-party plug-in support, GUI designer, support for JSR-14 (Generics), and a host of other features to secure this boast. If that wasn't enough, IntelliJ IDEA is the industry's only IDE that was built from the ground up to enhance productivity and provides an ergonomic development environment to suit individual tastes and coding styles.

Why Read This Overview?

IntelliJ IDEA 4.0 Overview was written for developers, project managers, architects, and even sales staff – those who have used prior version of IDEA or those who are new to the IDEA experience. While there is no better way to learn IDEA than by downloading it and actively trying it, this overview is targeted at individuals who wish to accelerate their learning curve through written print before sitting down in front of a computer and starting IDEA.

This overview assumes you have a rudimentary understanding of Java fundamentals and Object-orientated programming; it is not necessary for you to have an expert, zealot like understanding of all Java based or centered technologies. IDEA is a very user friendly tool and will quickly prove to 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.

We hope that this overview will be of assistance to you as you begin your journey on IDEA 4.0, and we are confident in saying that once you have given IDEA a fair hearing, it is unlikely you will want to replace it with anything else.

IntelliJ IDEA: The Intelligent, Usable Java Editor

Ever since IntelliJ IDEA made its debut into the Java IDE industry in 2001, the expectations for IDE performance have been forced up a notch. What then became the world's smartest and most usable Java IDE on the planet has continued to lead and push the envelope in terms of usability, refactoring, and code automation features that have remained unparalleled in the market by its imitators.

IDEA's mana has always been summed up in two words: Intelligence and Usability. IDEA has never promised to be the "everything editor" – it has always promised to be first and foremost a complete Java editor with the ability to understand the code in its editor as good as those who wrote it.

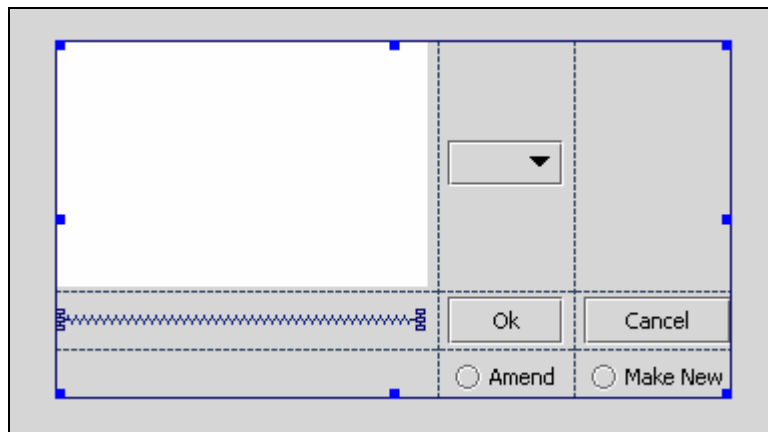
Every aspect of IDEA’s construction, its features, functions, UI layout, and integrations were all specifically designed to maximize developer ergonomics. As noted earlier, this ergonomic aspect is what has made, and continues to make, IDEA the industry’s most intelligent and user friendly IDE available.

The following sections of this overview will cover the main features available in IDEA, giving you an extensive taste of why IDEA has become the *de facto* pioneer in IDE innovation, and why we continue to unabashedly maintain our motto: “Develop with Pleasure!”

New Features in IntelliJ IDEA 4.0

GUI Designer

IDEA now comes powered with a new **GUI Designer** to let you rapidly create any type of interface for all of your development needs. Its component layout paradigm and intuitive user interface enable you to quickly and easily deal with dialogs and panels regardless of complexity. As shown in figure *GUI 1.1*, IDEA provides an easy to utilize point-and-click grid layout pane for GUI construction.



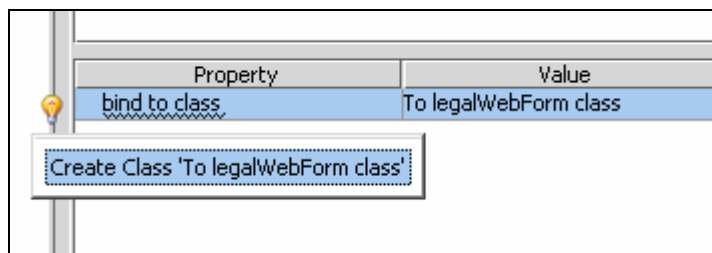
GUI 1.1

Moreover, IDEA completely separates you from working with the interface code itself. You can safely skip the details with developing layouts, Swing, etc. All interface coding is simplified to linking your class members with corresponding interface elements as shown in figure *GUI 1.2*.

Property	Value
binding	Testbutton
⊕ Horizontal Size Policy	Can Shrink, Can Grow
⊕ Vertical Size Policy	Fixed
fill	None
anchor	West
Row Span	1
Column Span	1
⊕ Minimum Size	[-1, -1]
⊕ Preferred Size	[-1, -1]
⊕ Maximum Size	[-1, -1]
enabled	<input checked="" type="checkbox"/>
horizontalAlignment	Leading
horizontalTextPosition	Trailing
selected	<input type="checkbox"/>
text	Amend
toolTipText	
verticalAlignment	Center
verticalTextPosition	Center

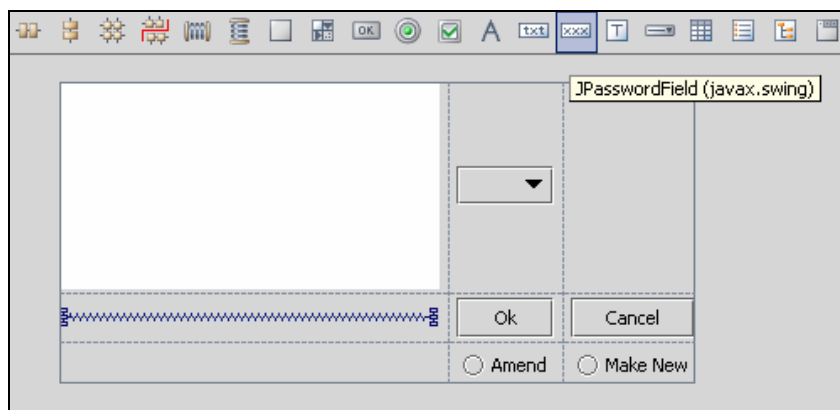
GUI 1.2

As shown in figure *GUI 1.3*, even IDEA's traditional code completion and intention action features work within the GUI designer tool.



GUI 1.3

As shown in figure *GUI 1.4*, Swing components are added to your grid from a simple to utilize Swing component tool bar. IDEA also allows you to add additional components and component libraries (from IDE Settings) to the existing palette to extend its power even more.



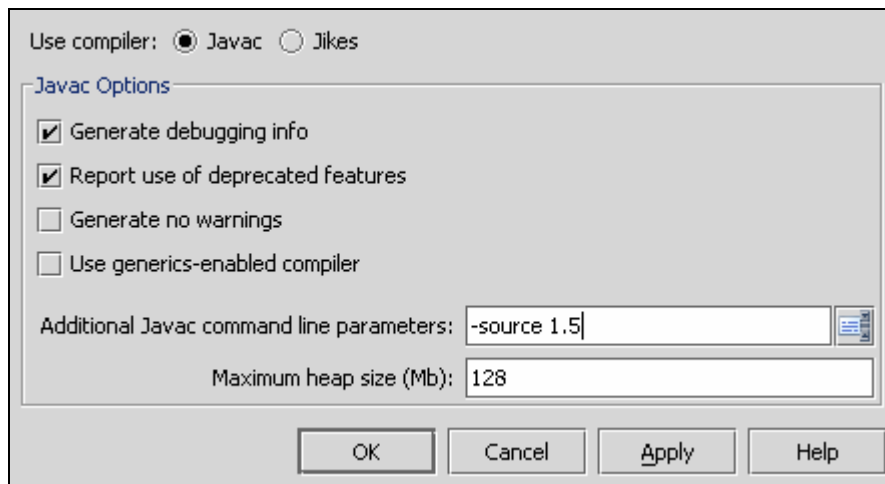
GUI 1.4

You can learn how to use IDEA's GUI Designer tool quickly by watching instructional videos on how to build, wire together, and deploy a form from this powerful tool, available here:

http://www.jetbrains.com/idea/training/ui_designer.html

Generics Support

Although at the time of this writing, no Sun finalized or stable implementation of Generics ([JSR-14](#)) exists, developers can still make use of Sun's beta implementation of Generics currently available in the beta version of JDK 1.5. To start using Generics, just utilize JDK 1.5 for the module/project with generics code, and then under **Settings | Compiler**, uncheck the **Use generics-enabled compiler** option and add `-source 1.5` to the **Additional Javac command line parameters** field as shown in figure *Generics 1.1*.



Generics 1.1

Nota Bene: IDEA version 4.1, which will be a free upgrade to all who purchase IDEA 4.0, will seamlessly integrate Generics once the implementation has been completed. The above mentioned settings will no longer be needed at that time.

Once IDEA has been setup to use Generics, these new Generics libraries will be indexed by IDEA's code completion tool. Once implemented into source, Generic code fragments will now be easily handled as shown in figure *Generics 1.2*.

```

24
25 List<String> geperifyMethodReturntype() {
26     List<String> list = new LinkedList<String>();
27
28     list.add(geperifyMethodReturntype().get(0));
29
30     return list;
31 }

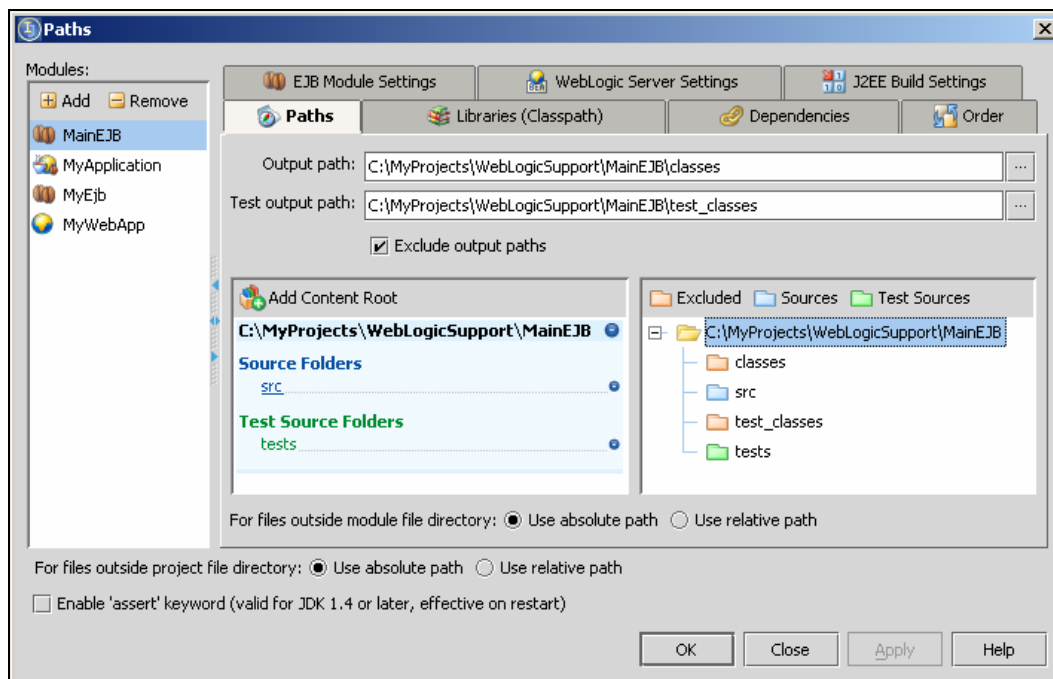
```

Generics 1.2

Modular Project Creation

IntelliJ IDEA's new modular project structure provides greater flexibility for configuring, setting up, and organizing your projects. In particular, it improves the management of complex projects with multiple internal dependencies and is especially useful for J2EE centered projects, where multiple projects have a tendency to share a lot of common code (connection handlers for example). Using modules not only eases project and code management by helping you divide and organize your projects with logically associated components, but also lets you share actual code, removing the need to make copies of the same package and classes because modules can be reused among separate projects.

Since a module exists as a separate logical part of a project, incorporating working sources, libraries, reference to a target JDK and more, it can thus be compiled, ran or debugged as a standalone entity. IDEA separates modules into the following specific module types: Java modules, EJB modules, Web modules and J2EE Application modules.

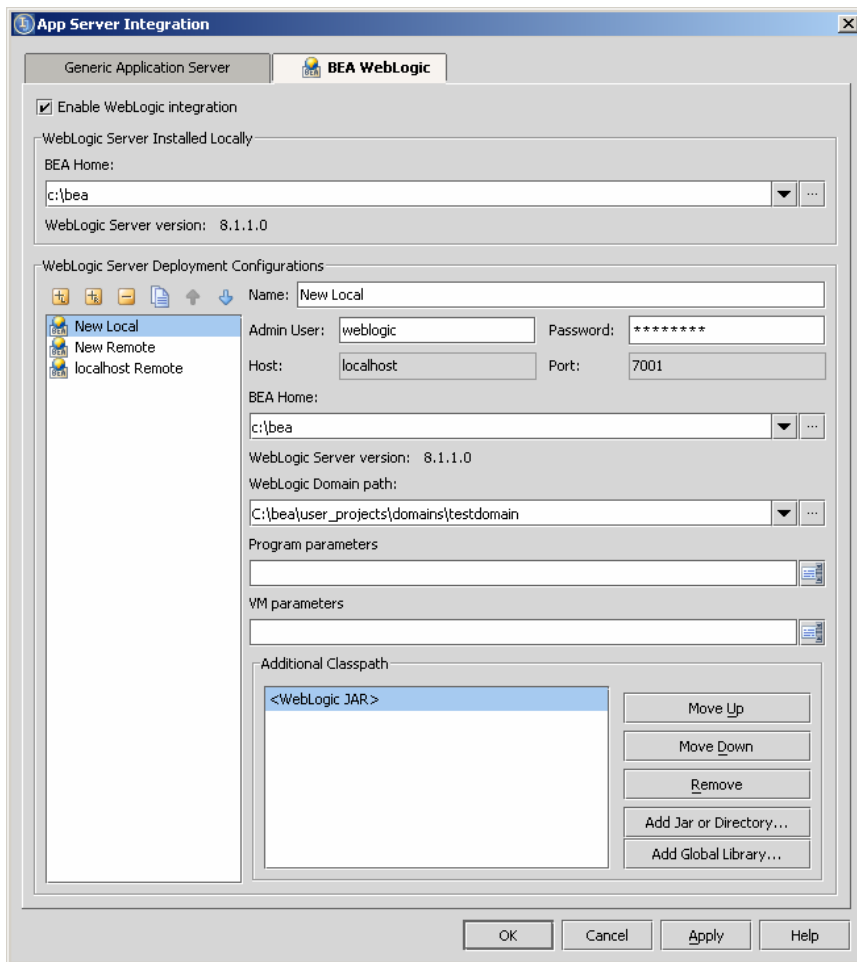


Modules 1.1

As shown in figure *Modules 1.1*, the modules **Path** pane allows you to assign module dependencies and organize (or re-organize) existing modules within an easy to use administration panel.

BEA WebLogic Integration

Those who depend on **BEA's WebLogic Application Server** to run their enterprise applications will be happy to know that IDEA now includes integration support with this industry leading application server. In projects created to work with WebLogic, IDEA fully supports and makes it easy to implement WebLogic-specific tags and deployment descriptors, and more. In addition, you can work with several different WebLogic servers at the same time. To do so, you simply setup a new WebLogic instance in IDEA WebLogic integration control panel as shown in figure *WebLogic 1.1*. This easy and rather intuitive configuration pane makes deploying WebLogic applications a snap. In IDEA's additional WebLogic tool window, the Deployment View, the process of deploying a WebLogic application is as simple as pressing the single **Deploy** button. IDEA takes care of all corresponding details once this has been done.



WebLogic 1.1

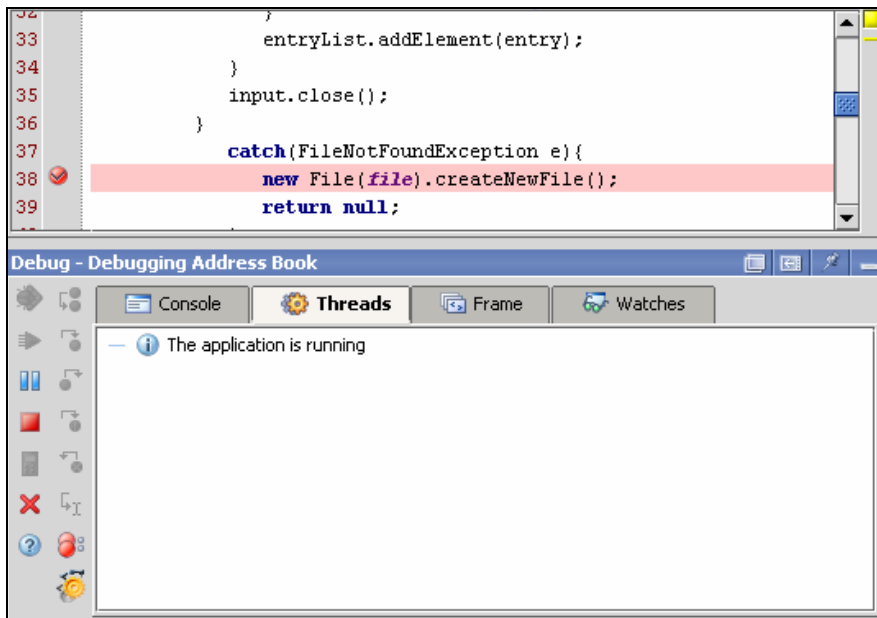
HotSwap Debugger

One of the more widely used and critical features needed in any IDE is a fast working and effective debugger. This is why IDEA has integrated a JPDA-based debugger that is not only extremely fast and simple to utilize, but also now includes the ability to perform **HotSwap Debugging**. A HotSwap enabled debugger allows you make changes to your code while the application is still running, removing the need to stop and start the application repeatedly to correct coding errors. Simply invoke the reload function to reload the changed object, and continue the debugging process.

Note Bene: HotSwapping allows you to edit existing methods, fields, and constructors. However, it does not allow you to add new methods, fields, or constructors; you also cannot change field types, method signatures, nor can you delete methods, fields, or constructors in the given class. Doing so will lead to hotswap errors being thrown. You can read a more in-depth overview of how hotswap debugging works from:

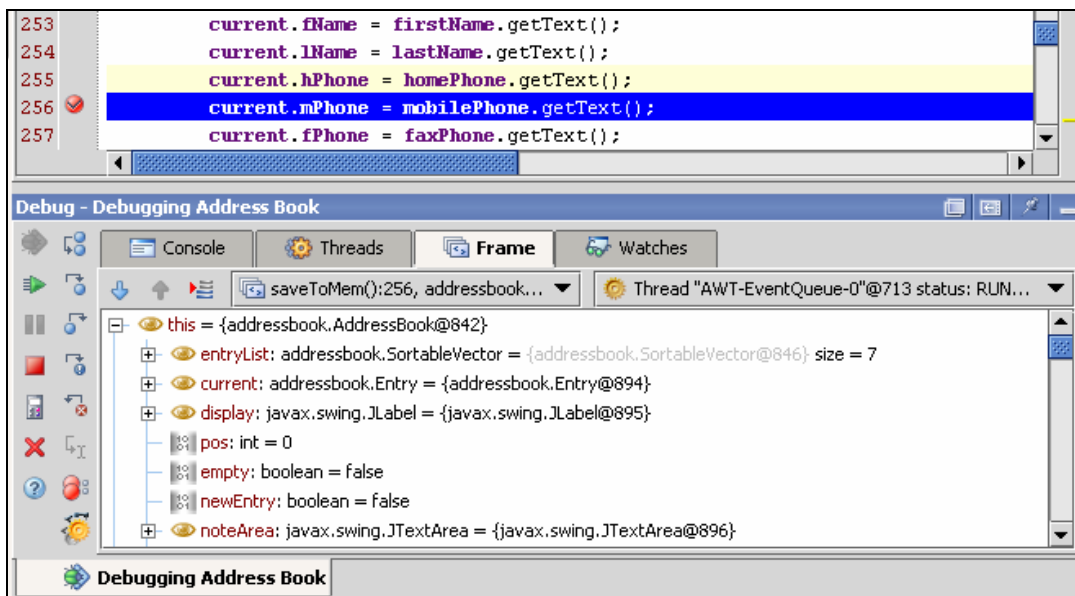
[http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi/com/sun/jdi/VirtualMachine.html#redefineClasses\(java.util.Map\)](http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi/com/sun/jdi/VirtualMachine.html#redefineClasses(java.util.Map))

As shown in figure *Debugging 1.1*, IDEA's powerful debugger incorporates a very friendly and intuitive user interface that allows you to quickly hunt down and correct code errors should they happen to arise.



Debugging 1.1

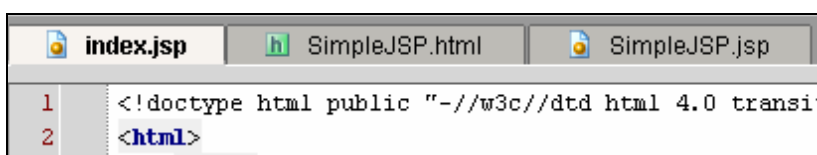
You can select an entire class file, block of code, or just single lines of code to observe during the debugging process, and you can access the debugger's output in an easy to read tree-view pane as shown in figure *Debugging 1.2*.



Debugging 1.2

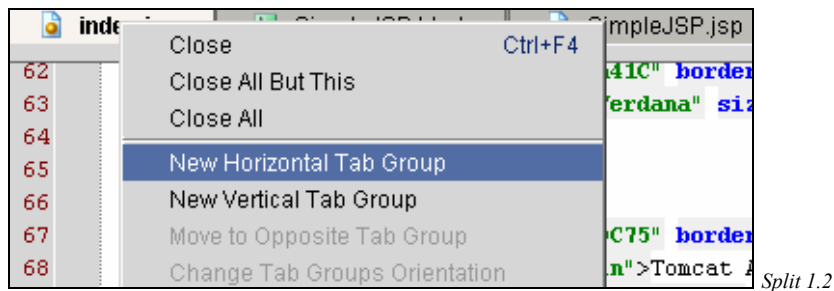
Split Editor

If you are the kind of developer who likes to work with multiple source files at one time (or just like to compare files side by side), you can now open two IDEA editors simultaneously for maximum viewing pleasure. To do so, simply select a source file's tab as shown in figure *Split 1.1*.

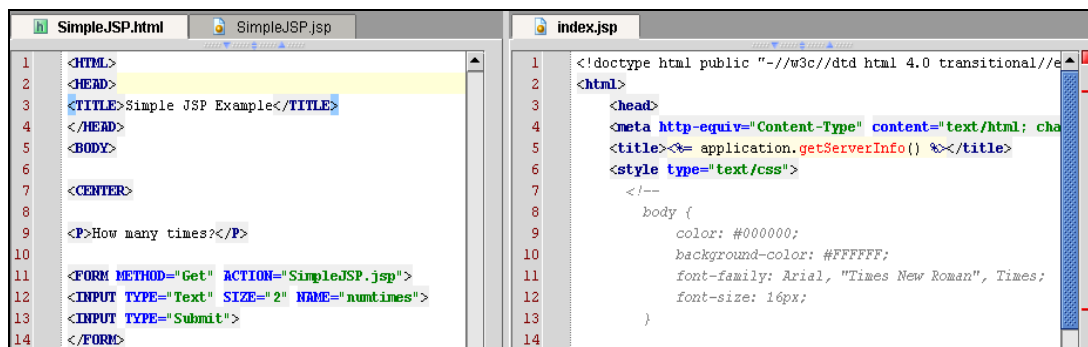


Split 1.1

Right-click on the selected tab and then choose either **New Horizontal Tab Group** or **New Vertical Tab Group** as shown in figure *Split 1.2*, and a new editor pane will be opened with the selected tab file according to the New Horizontal/Vertical Tab Group selection as shown in figure *Split 1.3*.

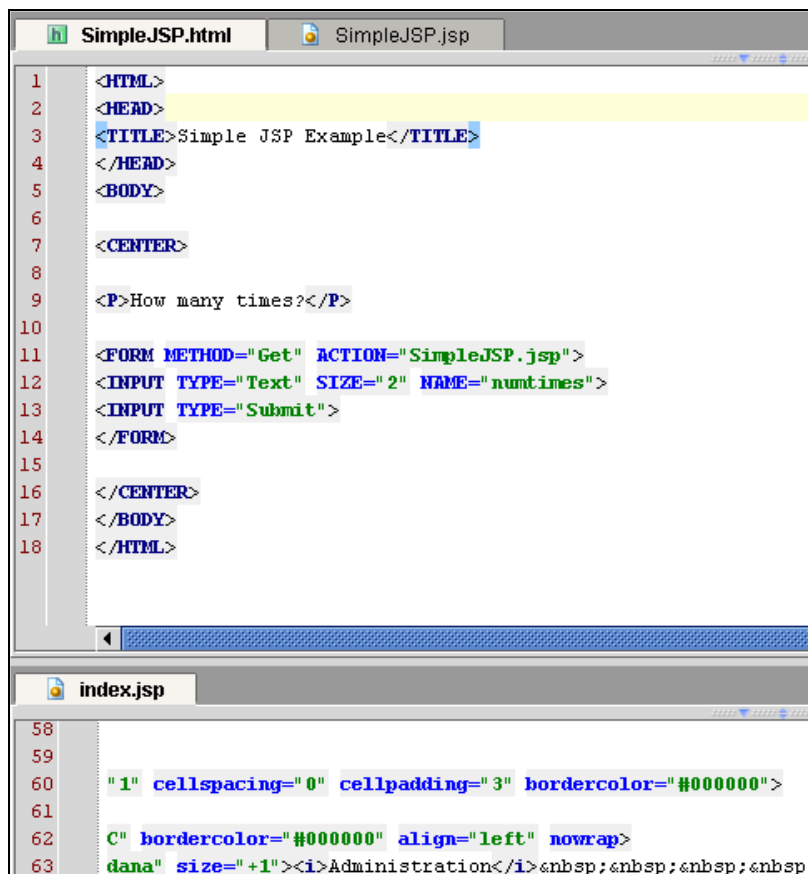


Split 1.2



Split 1.3

Alternatively, if New Vertical Tab Group is selected, the new editor is opened *column* wise as shown in figure *Split 1.4*.



Split 1.4

Code Completions

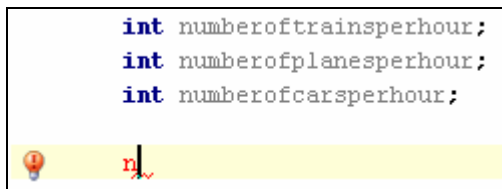
There has always been a dichotomy between code quality and coding quickly. Tradition has it that one is normally sacrificed to obtain the other and that it is almost an impossible task to bring the two together into a harmonious synthesis. Well, that was exactly what was happening until IntelliJ IDEA walked on the scene.

IDEA comes equipped with three robust types of code completion features to help you accurately speed up your coding responsibilities: **Basic**, **Smart-Type**, and **Class Name** completions.

Basic (Ctrl + Space)

IDEA's **Basic** code completion feature not only completes basic Java syntax for you, but will also dynamically create a selection list of usable code completions based upon what has been written in the editor. As shown in figures *Code Completions 1.1*, *1.2*, and *1.3*, as new objects are added to your source, IDEA memorizes and then allows you to recall complete variables by selecting **Ctrl + Space** on the keyboard.

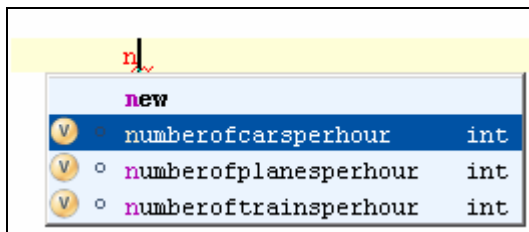
```
int numberOftrainsperhour;
int numberOfplanesperhour;
int numberOfcarsperhour;
```



Code Completions 1.1

```
n
```

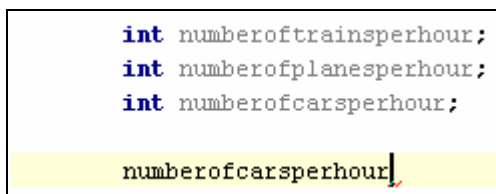
new	
numberofcarsperhour	int
numberofplanesperhour	int
numberoftrainsperhour	int



Code Completions 1.2

```
int numberOftrainsperhour;
int numberOfplanesperhour;
int numberOfcarsperhour;
```

```
numberofcarsperhour
```

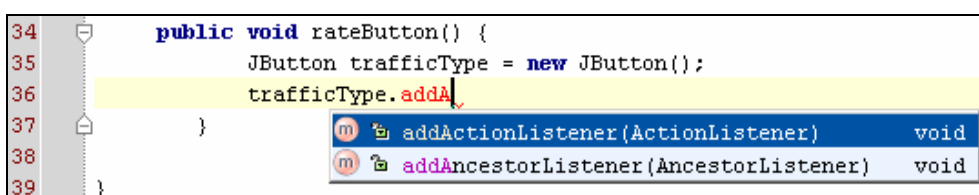


Code Completions 1.3

As shown in figure *Code Completions 1.4*, 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 has been previously used.

```
34 public void rateButton() {
35     JButton trafficType = new JButton();
36     trafficType.addAction
37 }
38
39 }
```

addActionListener	(ActionListener)	void
addAncestorListener	(AncestorListener)	void



Code Completions 1.4

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 coded. For example, as shown in figure *Code Completions 1.5*, when the Smart-Type completion feature is invoked, a popup with appropriate selections appears ready to implement the selected item.

```
37 public void rateButton() {
38     JButton trafficType = new JButton();
39     trafficType.addActionListener(new Ac
40 }
41
42 }
43
44
```

Code Completions 1.5

Once the desired item from the popup has been selected, IDEA will automatically implement it with all of its corresponding components as shown in figure *Code Completions 1.6*.

```
37 public void rateButton() {
38     JButton trafficType = new JButton();
39     trafficType.addActionListener(new ActionListener() {
40         public void actionPerformed(ActionEvent actionEvent) {
41             //To change body of implemented methods use Options | File Templates.
42         }
43     });
44 }
```

Code Completions 1.6

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).

```
15 public class initiationClass extends JP
16
17
18
19
20
21
22
23
24
25
26
27
```

Code Completions 1.6

List of selectable classes in popup after Class Name code completion function has been invoked

```
15 public class initiationClass extends JPanel
16
```

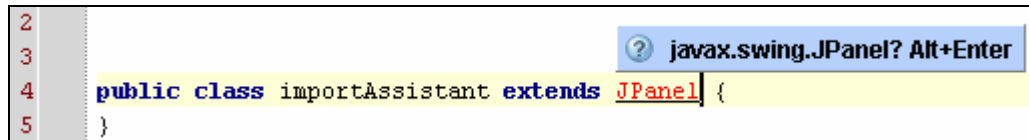
Code Completion 1.7

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

Import Assistant

One basic but very important feature that really enhances productivity is IDEA's **Import Assistant**. Just start typing the Java class short name into the editor and the import assistant will automatically launch a popup suggesting that you import its relevant corresponding Java class as shown in figure *Import Assistant 1.1*.

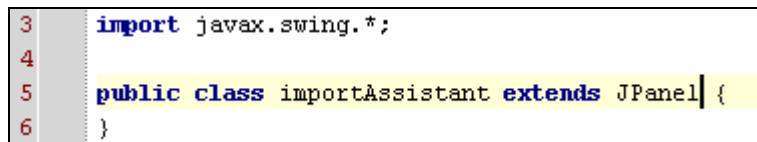
```
2
3
4 public class importAssistant extends JPanel {
5 }
```



Import Assistant 1.1

Not only will IDEA make the 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**.

```
3 import javax.swing.*;
4
5 public class importAssistant extends JPanel {
6 }
```

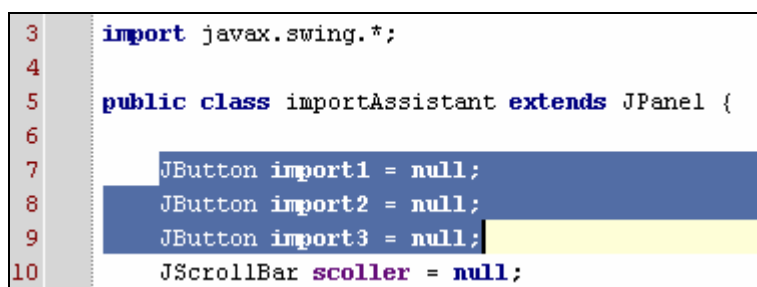


Import Assistant 1.2

In figure *Import Assistant 1.2*, after selecting **Alt + Enter** on the keyboard, the statement is imported, the popup disappears, and the red text on `JPanel` (error highlighting feature) vanishes. All of this is done without your caret position moving!

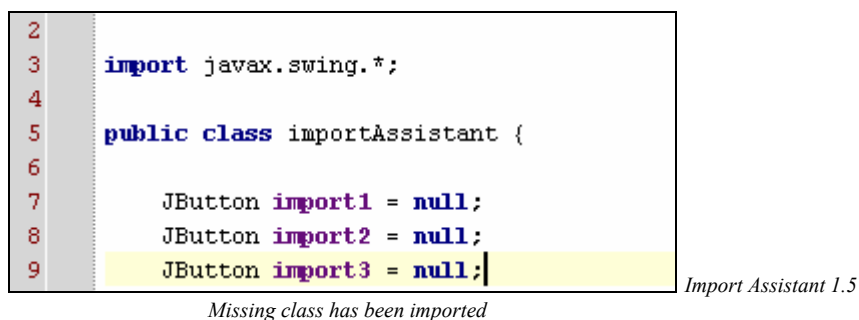
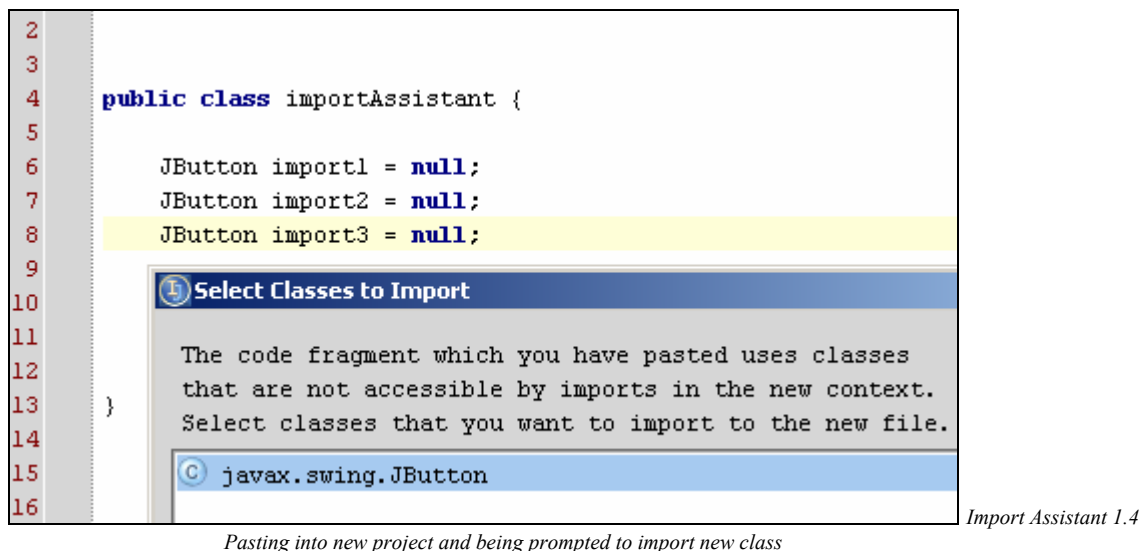
The import assistant also works when importing large blocks of code. For example, if you copy a block of code from one project file, and you paste this block of code into another project file, the import assistant will automatically prompt you for permission to import the relevant Java classes that are 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.

```
3 import javax.swing.*;
4
5 public class importAssistant extends JPanel {
6
7     JButton import1 = null;
8     JButton import2 = null;
9     JButton import3 = null;
10    JScrollBar scroller = null;
```



Copying from one project

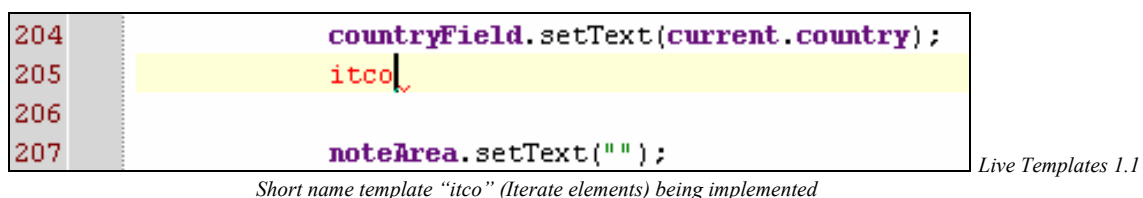
Import Assistant 1.3



Although for illustrative purposes we have used a very simple code example, IDEA does not have a problem handling large code selections and importing any requisite classes. When IDEA prompts you with the class import dialog, as shown in figure *Import Assistant 1.4*, you can select any class to import (or not to import), and should you miss anything once you have made your selection, IDEA will again prompt you asking if you wish to make any additional imports for any missing item.

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 single key strokes. 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 live template.



The entire live template index can be accessed by selecting **Code | Insert Live Template** from the main menu or by pressing **Ctrl + J** on the keyboard as shown in figure *Live Templates 1.2*. 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 selections quickly.

```

204     countryField.setText(current.country);
205     it
206     itar    Iterate elements of array
207     itco   Iterate elements of java.util.Collection
208     iten   Iterate java.util.Enumeration
209     itit   Iterate java.util.Iterator
210     itli   Iterate elements of java.util.List
211     // don't print new line after last line of notes
212     if(i < (Entry.maxNotes-1)){

```

Live Templates 1.2

Ctrl + J brings up the live template index

When an item from the index is selected, the corresponding template is implemented as shown in *Live Templates 1.3*

```

205     countryField.setText(current.country);
206     for (Iterator iterator = entryList.iterator(); iterator.hasNext();) {
207         Entry entry = (Entry) iterator.next();

```

Live Templates 1.3

Depending on your project’s requirements, you can edit the existing templates or add more templates to the index by creating your own. To call up the Live Templates editor, select **File | Settings** and then select the Live Templates icon on main settings pane. 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 as shown in figure *Live Templates 1.4*.

Abbreviation	Description	Active
collections		
html/xml		
iterations		
other		
output		
serr	Prints a string to System.err	<input checked="" type="checkbox"/>
sout	Prints a string to System.out	<input checked="" type="checkbox"/>
soutm	Prints current class and method names to System.out	<input checked="" type="checkbox"/>
soutv	Prints a value to System.out	<input checked="" type="checkbox"/>
plain		
surround		

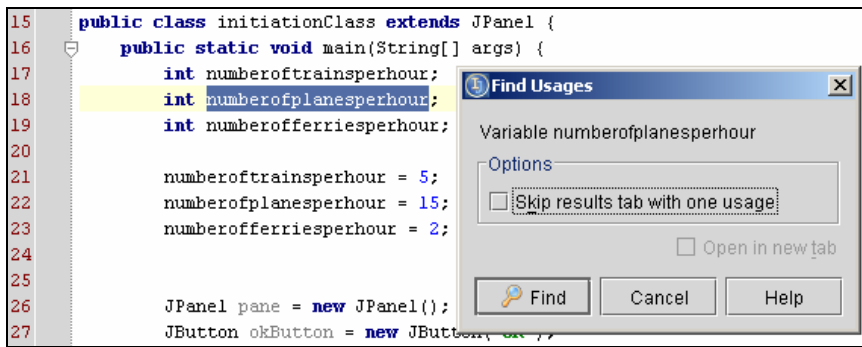
Live Templates 1.4

Live Templates editing / adding panel

Searching for Usages

Working in large projects can sometimes overwhelm your ability to keep track of every class, method, field, or variable in a project. To leverage this work, IDEA’s **Usage Search** function helps users quickly determine the functionality of any selected class, method, field, or variable in a project. Users simply click on the item they want to search, and IDEA will show all usages of that item in an entire project in an easy to read and navigate usages result tree.

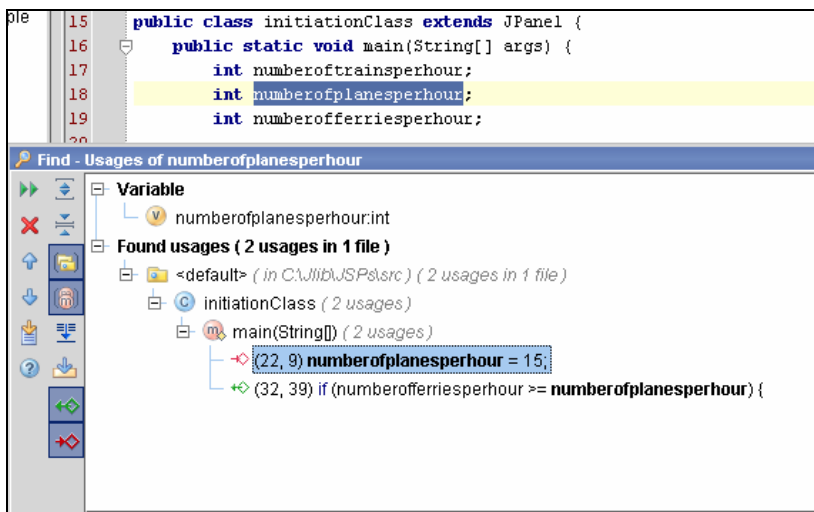
As shown in figure *Searching for Usages 1.1*, any item in a project can be searched in order to find out where that item is being used.



Searching for Usages 1.1

The project field "numberofplanesperhour" is being searched.

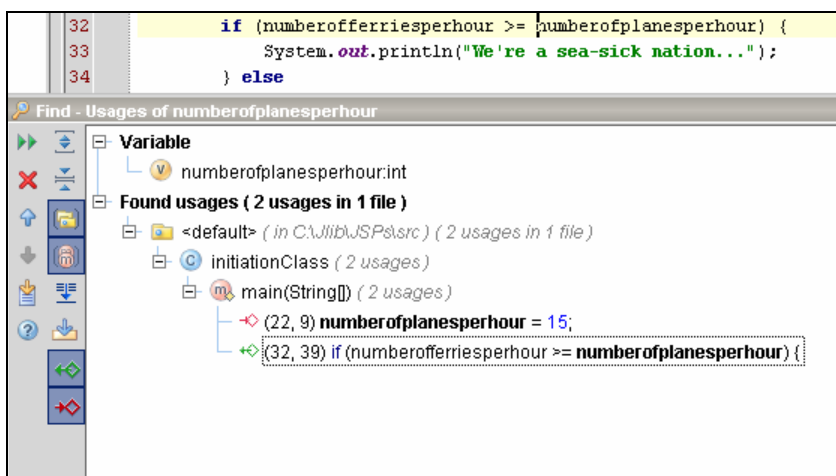
As noted above, all results are viewable in an easy to navigate tree panel as shown in figure *Searching for Usages 1.2*.



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 general usage search, IDEA also allows users to search for specific element types by employing a rich set of search options and filters. This usage search feature will not only search code within the immediate editor window, but you can also enable it to search entire packages and projects.

Code Layout Manager

If you have ever been involved in a project with multiple developers and found yourself reading the code of somebody else, you know that it can appear at times as a foreign language you have yet to learn. If this is the case, or you are the guilty one who writes illegible code for your colleagues, you no longer have to fear. IDEA's **Code Layout Manager** tool is perfect for creating, optimizing, controlling, and directing a uniform approach to code development layout.

```
14 public class initiationClass extends JPanel {
15     public static void main(String[] args) {
16         int numberoftrainsperhour;
17         int numberofplanesperhour;
18         int numberofferriesperhour; numberoftrainsperhour = 5; numberofplanesperhour = 15;
19         numberofferriesperhour = 2;
20         JPanel pane = new JPanel();
21         JButton okButton = new JButton("OK"); JButton cancelButton = new JButton("Cancel");
22         JButton ignoreButton = new JButton("Ignore");
23         if (numberofferriesperhour >= numberofplanesperhour) {
24             System.out.println("We're a sea-sick nation...");
25         } else System.out.println(numberoftrainsperhour);
26     }
27
28     public void rateButton() {
29         JButton trafficType = new JButton();
30         trafficType.addActionListener(new ActionListener() {
31             public void actionPerformed(ActionEvent actionEvent) {
32                 //To change body of implemented methods use Options | File Templates.
33             }
34         });
35     }
36 }
```

Code Layout Manager 1.1

Select the block of code you want to format

Utilizing this powerful feature is initiated with the touch of a key. As shown in figure *Code Layout Manager 1.1*, you only have to highlight a 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 reorganized as shown in figure *Code Layout Manager 1.2*.

```
14 public class initiationClass extends JPanel {
15     public static void main(String[] args) {
16         int numberoftrainsperhour;
17         int numberofplanesperhour;
18         int numberofferriesperhour;
19         numberoftrainsperhour = 5;
20         numberofplanesperhour = 15;
21         numberofferriesperhour = 2;
22         JPanel pane = new JPanel();
23         JButton okButton = new JButton("OK");
24         JButton cancelButton = new JButton("Cancel");
25         JButton ignoreButton = new JButton("Ignore");
26         if (numberofferriesperhour >= numberofplanesperhour) {
27             System.out.println("We're a sea-sick nation...");
28         } else
29             System.out.println(numberoftrainsperhour);
30     }
31
32     public void rateButton() {
33         JButton trafficType = new JButton();
34         trafficType.addActionListener(new ActionListener() {
35             public void actionPerformed(ActionEvent actionEvent) {
36                 //To change body of implemented methods use Options | File Templates.
37             }
38         });
39     }
40 }
```

Code Layout Manager 1.2

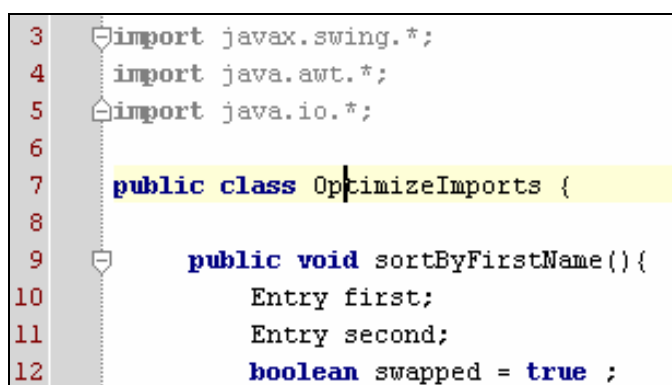
Code after IDEA's Reformat Code option has been used

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

Optimize Imports

An additional 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 an eyesore.

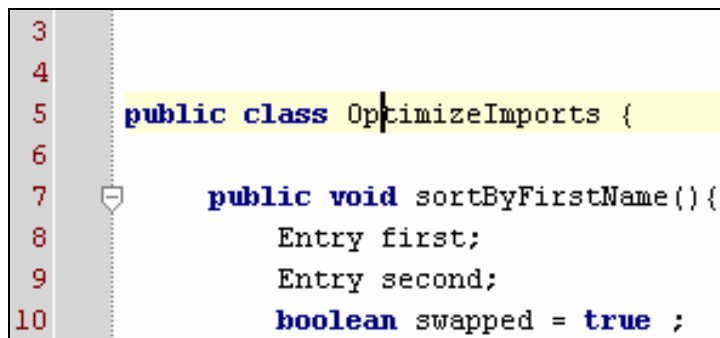
As shown in figure *Optimize Imports 1.1*, there are three grayed-out imports that are not currently being used by the open class (they may have been used, but are no longer needed). Simply select the Optimize Imports function (menu **Tools | Optimize Imports** or **Ctrl + Alt + O**) and these imports will be safely removed as shown in figure *Optimize Imports 1.2*.



```
3  import javax.swing.*;
4  import java.awt.*;
5  import java.io.*;
6
7  public class OptimizeImports {
8
9      public void sortByFirstName(){
10         Entry first;
11         Entry second;
12         boolean swapped = true ;
```

Optimize Imports 1.1

Three grayed out imports to be removed



```
3
4
5  public class OptimizeImports {
6
7      public void sortByFirstName(){
8         Entry first;
9         Entry second;
10         boolean swapped = true ;
```

Optimize Imports 1.2

Three grayed out imports have been removed

Intention Actions

Sometimes our greatest ideas suddenly hit us in the 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 goodies into your 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.

```

13 public class Server extends HttpServlet {
14     void processRequest (Request request, int flags){
15         allocateResources();
16
17         createAndStartProcessingThread(request, flags);
18
19         clientCall();
20     }
21
22
23

```

Intention Actions 1.1

Unknown usage import correction dialog

As the unknown usage popup 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*.

```

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

```

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 found in existing packages where these methods have previously been implemented. As shown in figure *Intention Actions 1.3*, a known Java identifier has been entered into the editor, and IDEA will suggest that you implement this keyword's corresponding items (in this case, the *MouseListener* methods).

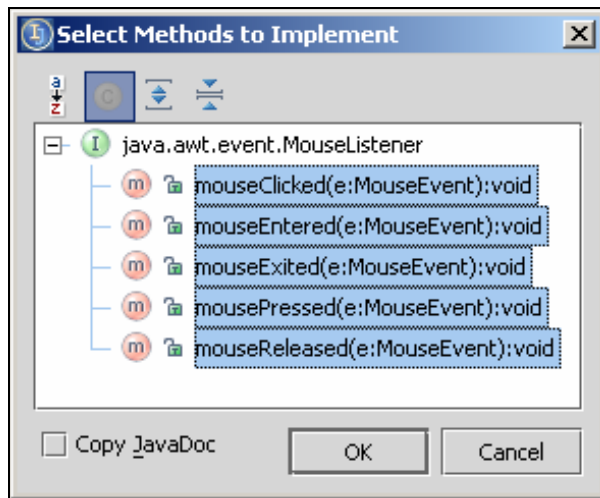
```

3 import java.awt.event.MouseListener;
4
5 public class BasicCodeCompletion extends ClassLoader implements MouseListener{
6
7
8
9

```

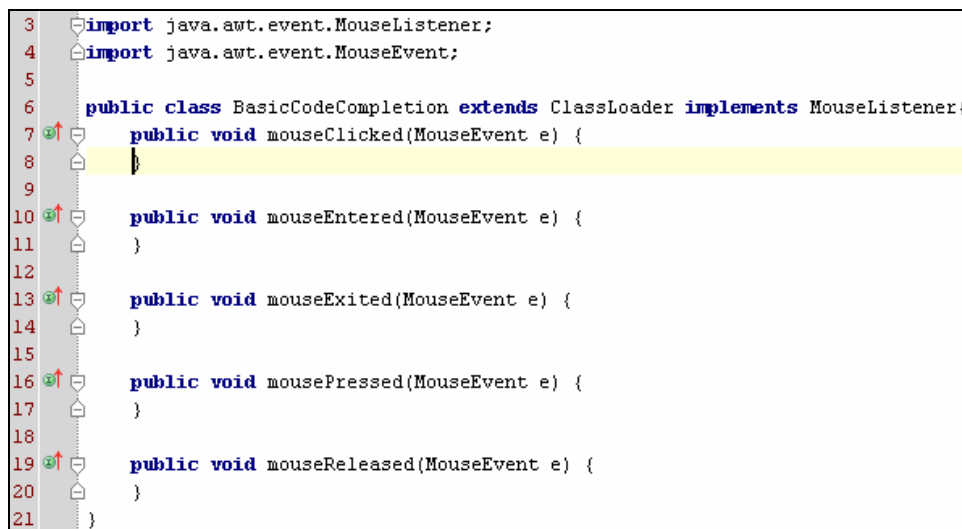
Intention Actions 1.3

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 the 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 the **OK** button.



Intention Actions 1.4

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

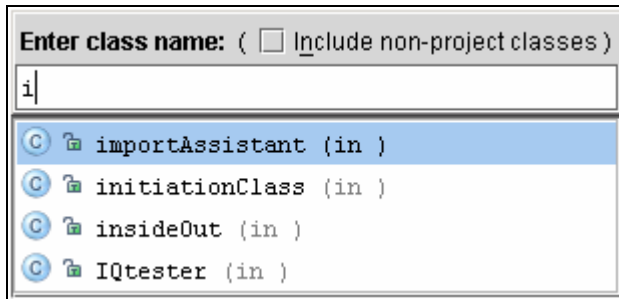


Intention Actions 1.5

Opening Class by its Short Name

When you are working with large projects that contain a large number of classes, finding a specific class can waste a lot of valuable of time. To ensure that you spend your valuable time actually coding, IDEA lets you **open any class by its short name**, eliminating the need to perform time consuming searches for a particular class.

Simply select **Ctrl + N**, and once you start to enter the first letter of the desired class, IDEA will dynamically begin to limit your possible selections as shown in figure *OCSN 1.1*. Once your choice has been selected, the desired class will be viewable in the source code editor panel.

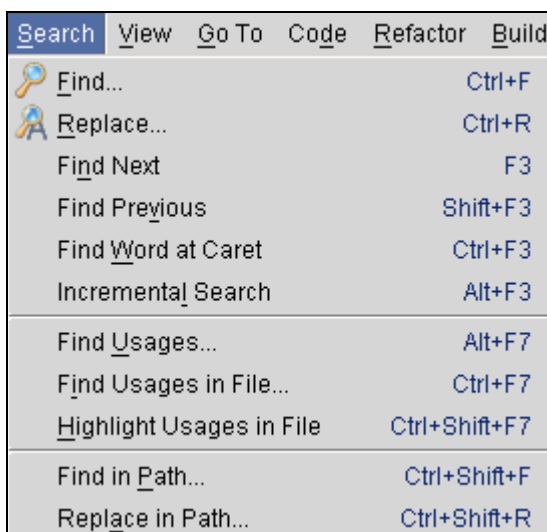


OCSN 1.1

Class search by short name

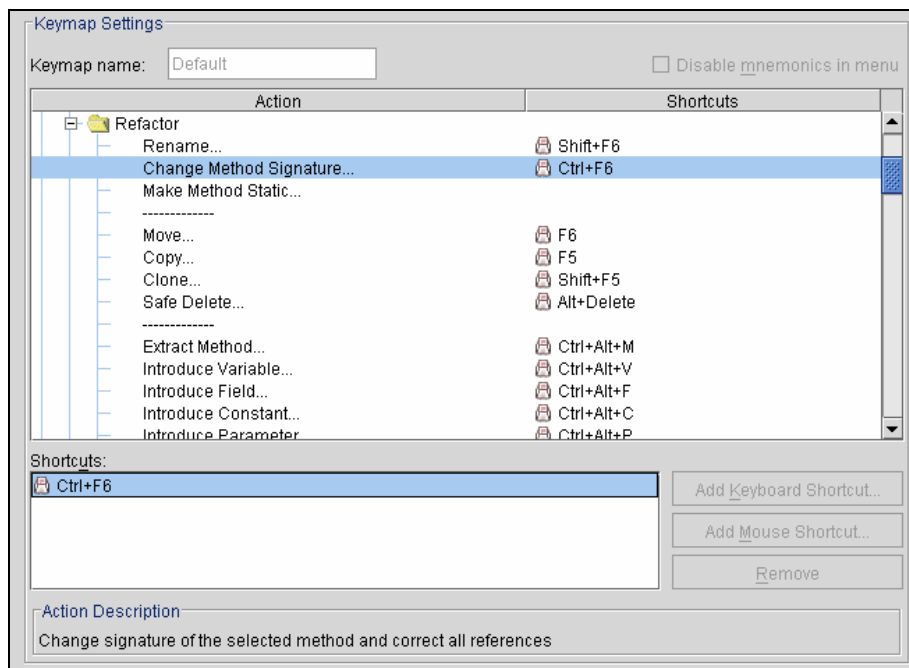
Keymapping

This IDEA feature allows you to ergonomically adjust how the IDE's functions are invoked. As you can see in figure *Keymapping 1.1*, shortcuts to most IDEA functions appear on the right side of menu items located under main menu categories.



Keymapping 1.1

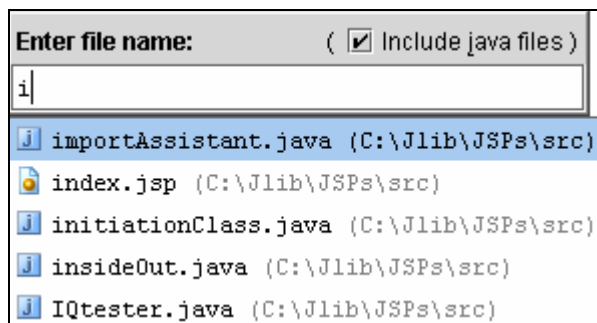
IDEA comes with a default set of shortcuts ready to use immediately after software installation. However, this default setup can be replaced with your own customized mapped selections. As shown in figure *Keymapping 1.2*, you can access the keymap index and change the default keymap selections and replace them with your own customized versions. You may also create additional keymap profiles, in case you prefer to have more than one set or more than one person uses the same computer. You can also save mapped configuration settings and export them to other machines running IDEA.



Keymapping 1.2

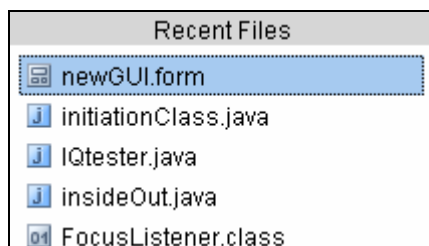
Navigation

As with any project, knowing what your project's internal components consist of is important. More importantly however, is how fast you can find them to make changes or add improvements. Not only does IDEA allow you to quickly open a class by its short name, but you can navigate to any file in a project by its short name as shown in figure *Navigation 1.1*, allowing you to quickly find declarations and type declarations, implementations, and super methods quickly.



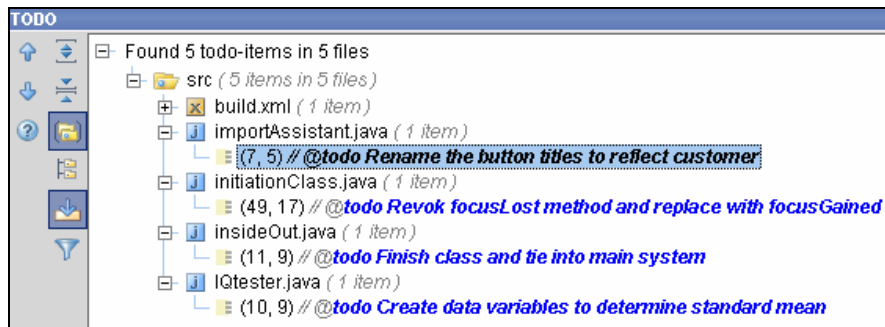
Navigation 1.1

In addition, IDEA allows you to quickly jump to the last change made in a file (**Ctrl + Shift + Backspace**) and even view the list of previously viewed files as shown in figure *Navigation 1.2*.



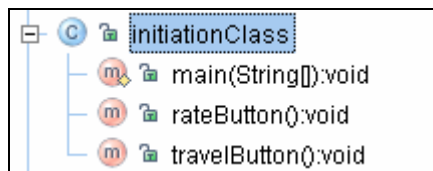
Navigation 1.2

Another feature to help keep you organized and manage your code efficiently is the *Bookmark* and *ToDo* functions. The **Bookmark** function allows you to mark lines of code in your project, and then allows you to quickly navigate back to those locations. The **ToDo** function allows you to see your ToDo comments in your source code in an easy to read tree-view panel as shown in figure *Navigation 1.3*. You can then navigate to the actual places in the source code by clicking on the specific ToDo in the tree-panel.



Navigation 1.3

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



Navigation 1.4

Code Inspection

Those who pride themselves on producing meticulously clean code are always surprised at what IDEA's code inspection feature is able to find. This feature empowers you with the ability to analyze your source code for irregularities and informs you when your code's design logic is "fuzzy." It highlights and navigates you to 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, take notice of the source code in figure *Code Inspection 1.1*. On **line 27** we have commented out and noted a deliberate error we have thrown in the source code.

```

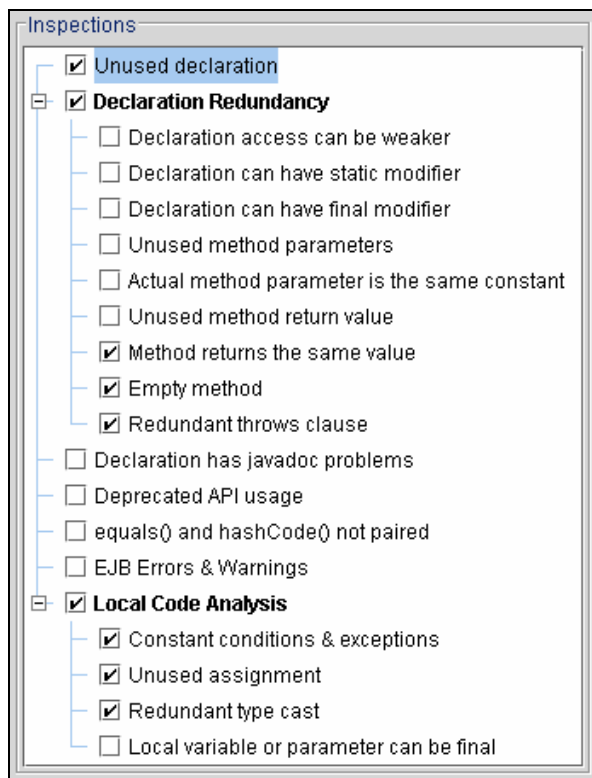
19 private class MyKeyListener extends KeyAdapter {
20     public void keyTyped(KeyEvent e) {
21         int keyCode = e.getKeyCode();
22
23         if (keyCode == KeyEvent.VK_F1) {
24             showHelp();
25         } else if (keyCode == KeyEvent.VK_F2) {
26             showClients();
27         } else if (keyCode == KeyEvent.VK_F2) { // Should be F3
28             showDevices();
29         } else if (keyCode == KeyEvent.VK_F4) {
30
31     }

```

Code Inspection 1.1

Example source code with conditional error

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 tool.

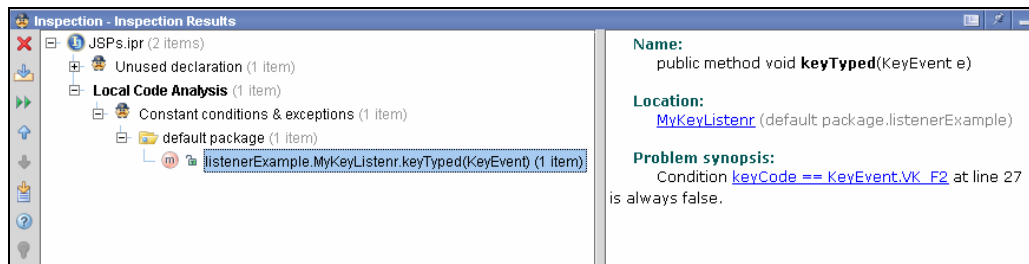


Code Inspection 1.2

Code Inspection Analyze and Search criteria option panel

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 the way it was intended. A quality assurance 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 source code example could be easily debugged manually without much fanfare; however, in a project with hundreds or even thousands of classes, interfaces, methods, and fields, it would be almost unimaginable to search for these errors manually. You just simply invoke the code inspection tool and let IDEA 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 and verifications, the code inspection's results will be viewable in an easy to read tree-like navigation window in an output control panel as shown above 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 newer staples to take hold in the world of development and push the paradigm of conventional programming has been the process of refactoring. What is refactoring? One industry refactoring maven, Martin Fowler, describes 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.”*¹

This of course is the technical definition of a process in theory; however in practice this art form can be extremely time consuming and difficult to perform when attempted manually. In addition, if one is actually crazy enough to try complicated refactorings manually, they inherit the risk of crippling working systems and turning them into unstable and non-functional gobs of code. This is why IDEA comes fully equipped with the most powerful refactoring tools available in the market. Refactoring processes such as *Renaming, Extract Method, Change Method Signature, Make Method Static, Extract Interface, Introduce Constant, Move*, and many more are bundled with IDEA for more than 25 different refactoring tools in total.

This section of the overview will briefly introduce some of the 25 plus refactoring tools provided with 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 key.

Renaming

One of the more common yet most used and useful refactoring tools integrated into IDEA is the **Renaming** refactoring. Renaming allows you to safely change the name of any package, class, method, field or variable in a specific file or desired project.

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

What would be the reason for doing this? Simple: to clean your code up. When naming methods, for example, a good programmer will reveal the purpose of that method by its name. As is shown in figure *Renaming 1.1*, the name refers to a general function.

```
34 public void rateButton() {
35     JButton trafficType = new JButton();
```

Renaming 1.1

In figure *Renaming 1.2*, the method has been renamed to a label more fitting to its specific function.

```
34 public void calculatetotaltravelratingButton() {
35     JButton trafficType = new JButton();
```

Renaming 1.2

During the renaming process, this tool automatically finds and corrects all references to a specific element (in both the working class and the rest of the entire project). As figure *Renaming 1.3* shows, an easy to read prompt will ask you to verify your changes – either by each individual instance or entire project.

```
TextArea textareal;
JButton button;
JButton button2;

public FourthQuarter(String report) {
    super(report);
    JPanel pane = (JPanel)
        getContentPane();

    button = new JButton("Print Out Sales Report");
    button.addActionListener(this);
```

Renaming 1.3

Once you have determined the appropriate items to refactor, and you have refactored them by selecting the **Do Refactor** button, the new results from the refactoring process are shown back in the editor as shown in figure *Renaming 1.4*.

```

TextArea textArea1;
JButton calculate;
JButton button2;

public FourthQuarter(String report) {
    super(report);
    JPanel pane = (JPanel)
        getContentPane();

    calculate = new JButton("Print Out Sales Report");
    calculate.addActionListener(this);
}

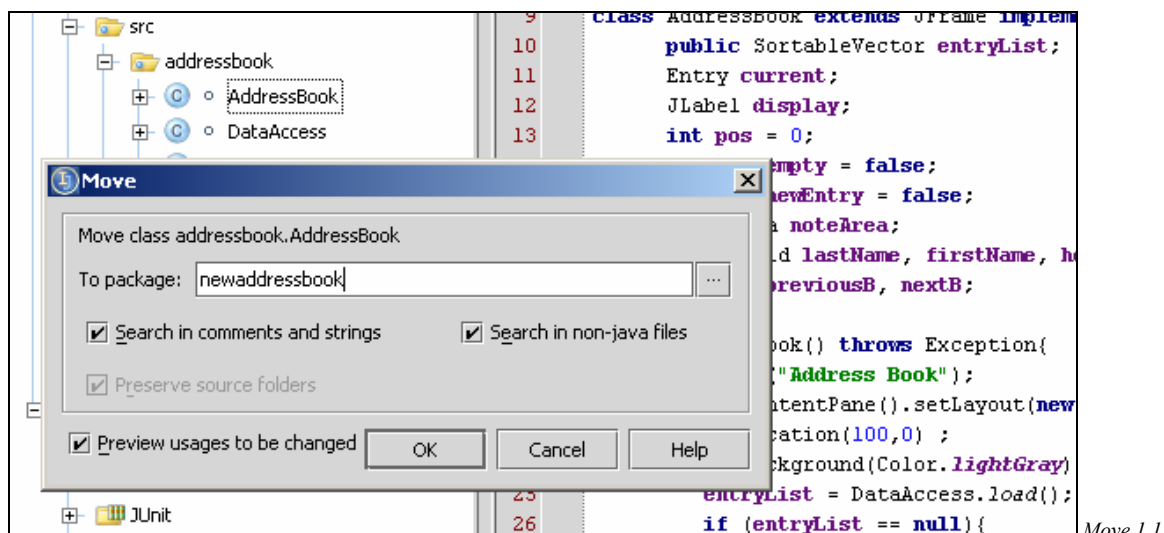
```

Renaming 1.4

Move

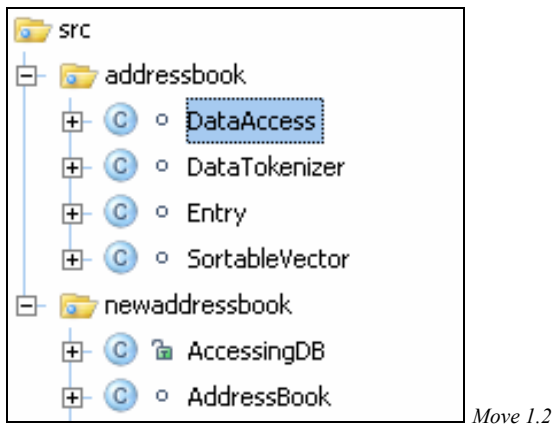
Along with the Rename refactoring, IDEA's **Move** refactoring tool is another straight forward yet highly powerful and widely used refactoring process that allows you to correct, improve, or transfer misplaced responsibilities in source code without a lot of hassle. It also enables you to quickly move 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 automated process eliminates any chance of introducing bugs into your code when moving items from place to place.

For example in figure *Move 1.1*, the Java class file (*AddressBook*) shown in the project view can be easily moved into a new location (or a previously existing one) as shown in the **To package:** field. All references to this class within the entire project will be changed to accommodate such change.



Move 1.1

Once the move process has been completed, you will see the previously mentioned Java class file has now been moved from the *addressbook* package into a new package called *newaddressbook* as shown in figure *Move 1.2*.



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 *AddressBook*, in this case *saveToMem()* on **line 255**, and invoke the Move refactoring (you can right-click your mouse and select **Refactor | Move** or press **F6** on the keyboard).

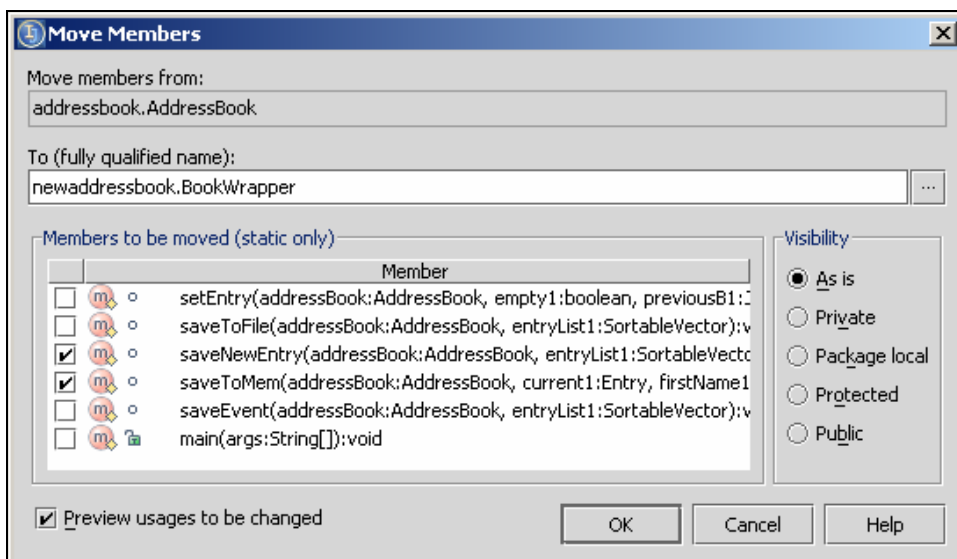
```

255 static void saveToMem(AddressBook addressBook, Entry current1,
256     int counter=0;
257     current1.fName = firstName1.getText();
258     current1.lName = lastName1.getText();
259     current1.hPhone = homePhone1.getText();
260     current1.mPhone = mobilePhone1.getText();
261     current1.fPhone = faxPhone1.getText();

```

Move 1.3

Once the refactoring has been started, you will be shown a control dialog informing you of your selection, and more importantly, a list of other members that should be moved along with your initial member as shown in figure *Move 1.4*.



Move 1.4

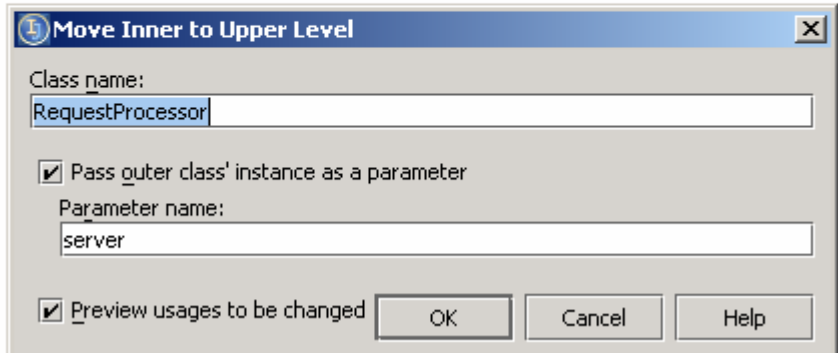
After the appropriate desired member selections have been made, and the Move refactoring has been completed (including your verification of the members to be moved), a new class will then be made in the newly mentioned location with your previously selected members to be moved as shown in figure *Move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

```
17 public class BookWrapper {
18     int field;
19
20     static void saveNewEntry(AddressBook addressBook,
21         entryList1.addElement(current1);
22     pos1 = entryList1.size()-1;
```

Move 1.5

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

```
10 public class StringUtil {
11     public static String toPlural(String word) {
12         if(word.length() == 0) return word;
13
14         if(word.charAt(word.length() - 1) == 'x' || word.charAt(word.length() - 1) == 's') {
15             return word + "es";
16         } else if (word.charAt(word.length() - 1) == 'y') {
17             return word.substring(0, word.length() - 1) + "ies";
18         } else {
19             return word + 's';
20         }
21     }
```

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 Introduce Variable is invoked on the expression `word.charAt(word.length() - 1)`.

```
10 public class StringUtil {
11     public static String toPlural(String word) {
12         if(word.length() == 0) return word;
13
14         if(word.charAt(word.length() - 1) == 'x' || word.charAt(word.length() - 1) == 's') {
15             return word + "es";
16         } else if (word.charAt(word.length() - 1) == 'y') {
17             return word.substring(0, word.length() - 1) + "ies";
18         } else {
19             return word + 's';
20         }
21     }
```

Variable 1.2

In figure *Variable 1.3*, the above mentioned complicated expression (and all of its occurrences) has now been changed into the expression `lastChar`.

```

10 public class StringUtil {
11     public static String toPlural(String word) {
12         if(word.length() == 0) return word;
13
14         char lastChar = word.charAt(word.length() - 1);
15         if(lastChar == 'x' || lastChar == 's') {
16             return word + "es";
17         } else if (lastChar == 'y') {
18             return word.substring(0, word.length() - 1) + "ies";
19         } else {
20             return word + 's';
21         }
22     }

```

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*.

```

10 public class StringUtil {
11     public static String toPlural(String word) {
12         if(word.length() == 0) return word;
13
14         int lastCharIndex = word.length() - 1;
15         char lastChar = word.charAt(lastCharIndex);
16         if(lastChar == 'x' || lastChar == 's') {
17             return word + "es";
18         } else if (lastChar == 'y') {
19             return word.substring(0, lastCharIndex) + "ies";
20         } else {
21             return word + 's';
22         }
23     }

```

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, your 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 refactorings 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 / Extract Superclass** from the main menu.

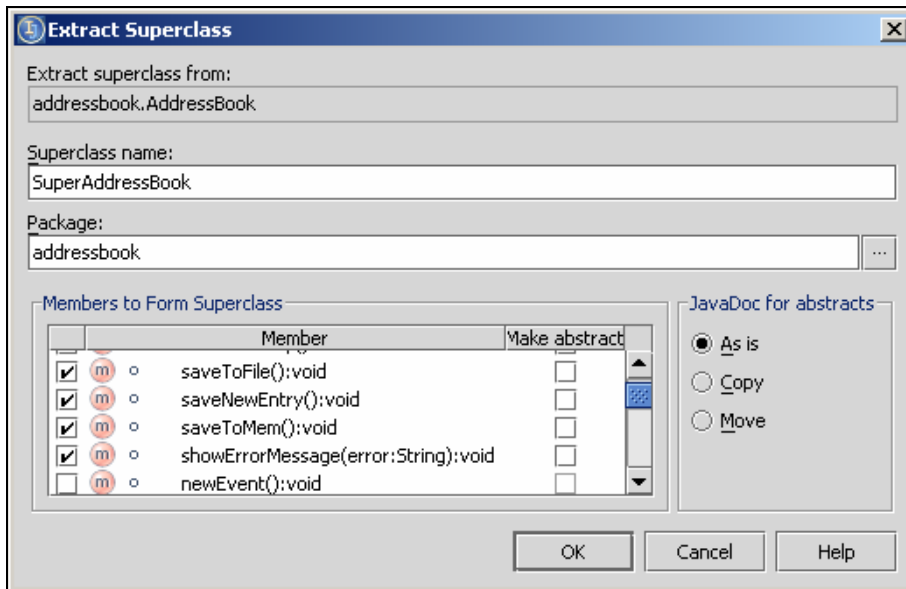
```

9   class AddressBook extends JFrame implements ActionListener{
10      public SortableVector entryList;
11      Entry current;
12      JLabel display;

```

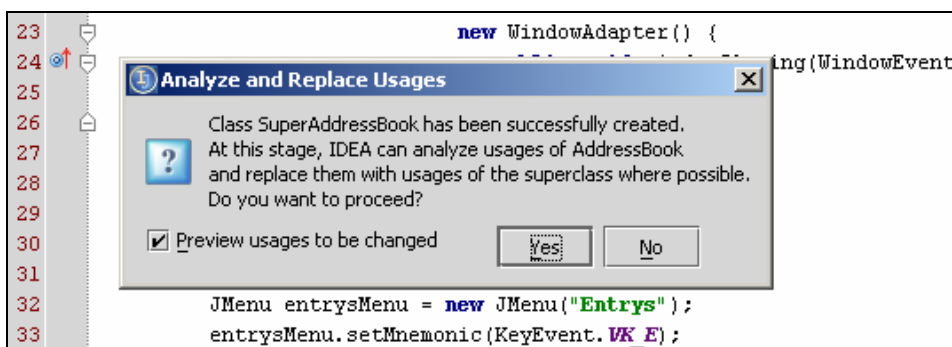
Extract 1 / S 1.1

Figure *Extract 1 / 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 associated objects, into a new interface.



Extract 1 / 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 new and improved ones as shown in figure *Extract Interface 1.3*. Like other refactorings in IDEA, a tree-view will be shown allowing you to approve your individual selections before making any 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 fixing 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 tool 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 tool.

```
12 public class BookletLibrary {
13     ArrayList myBooklets;
14
15     void renameEntry(String oldName, String newName){
16
17         Booklet bookletToRename = null;
18         for (int i = 0; i < myBooklets.size(); i++) {
19             Booklet booklet = (Booklet) myBooklets.get(i);
20             if(booklet.getBookletName().equals(oldName)) {
21                 bookletToRename = booklet;
22                 break;
23             }
24         }
```

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 content being referenced somewhere else. Now the resulting new method is easily identifiable and easily referenced by other methods and classes.

```
12 public class BookletLibrary {
13     ArrayList myBooklets;
14
15     void renameEntry(String oldName, String newName){
16
17         Booklet bookletToRename = bookletNewMethod(oldName);
18     }
```

Extract Method 1.2

Inline Method

The refactoring tool **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 tool 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 unneeded delegation in the *getEnteredName* method.

```

364     public String getEnteredName(){
365         return myNameField.getText();
366     }
367
368     public String getEntryName(){
369         return getEnteredName();
370     }

```

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 tool to remove the indirection chaff.

```

363
364     public String getEntryName(){
365         return myNameField.getText();
366     }

```

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 stated, sometimes there are methods that are simply factored in a sloppy manner, and the quickest way to fix them is to first 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 tool **Encapsulate Field**. This refactoring 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 Encapsulate Field has been invoked, it helps you create the appropriate getter and setter methods which hide the initial content of any selected field.

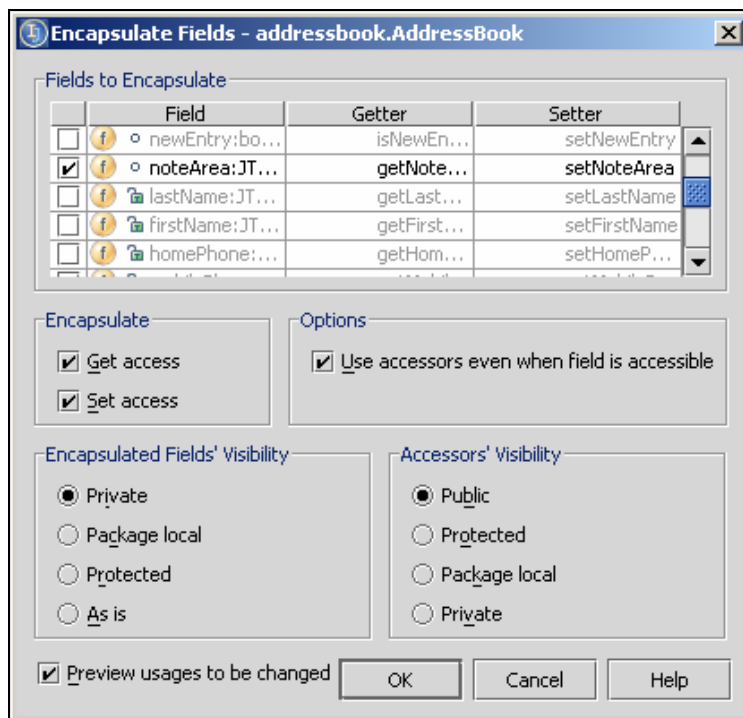
```

12     boolean empty = false;
13     boolean newEntry = false;
14     JTextArea noteArea;
15     public JTextField lastName, firstName,
16     JButton previousB, nextB;

```

Encapsulate Field 1.1

Select the public field you wish to encapsulate



Encapsulate Field 1.2

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 that encompasses a multitude of options for making a number of cosmetic and design changes to any desired method signature. Specifically, IDEA lets you perform 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 refactorings in greater detail in this overview, because by their names alone, their functions are pretty obvious. Some of these above mentioned refactorings can be read about in greater detail in Martin Fowler's book on refactoring previously mentioned in the Refactoring introduction page.

J2EE Support

Creating component based J2EE modules has become the *de facto* standard in today's highly competitive, rapidly 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.

Whether you are a small developer or part of a large corporate development team, the success of any project is defined, to a greater or lesser degree, by its relation to its completion schedule and budget. Working with enterprise applications 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, develop, and launch this compendium of technologies in an intelligent, fast, efficient, and timely fashion.

To ensure that all of your J2EE development needs are met, IDEA comes completely stocked with a vast selection of robust and usable features, including:

- 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

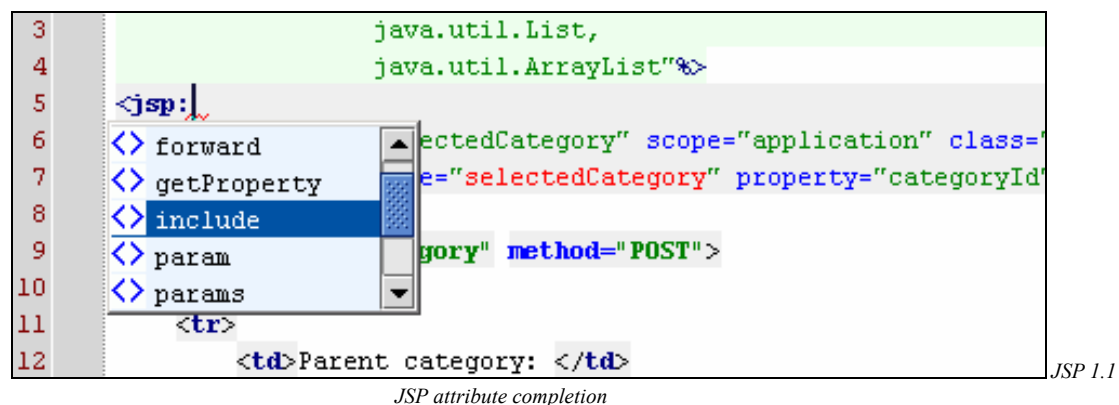
Web Application Development

If you are a Java developer and have experience creating web based applications, then no doubt **JSPs (JavaServer Pages)** have been an integral part of your development arsenal. Those who are yet to use JSPs, here is a quick run down: JSPs are HTML pages with inserted Java code that allow web developers and designers to quickly deploy and easily maintain dynamic and information-rich web content that is platform and server independent.

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

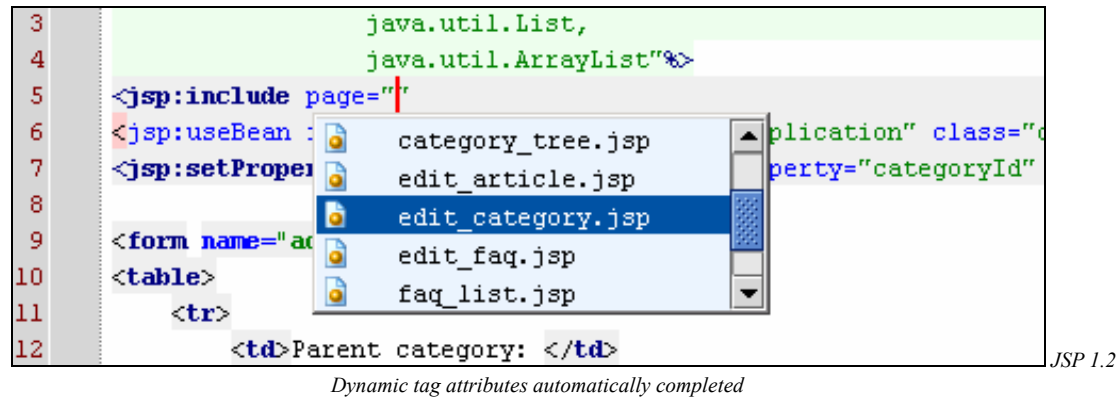
Having said this, if you are looking to utilize your limited time and resources to maximum efficiency, not to mention code for future scalability, then IDEA is the ideal development tool to use for JSP development. IDEA comes standard with JSP tag library and attribute code completions, code refactorings, error highlighting, on-the-fly debugging, and even JSP deployment capabilities all from within a single development environment.

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



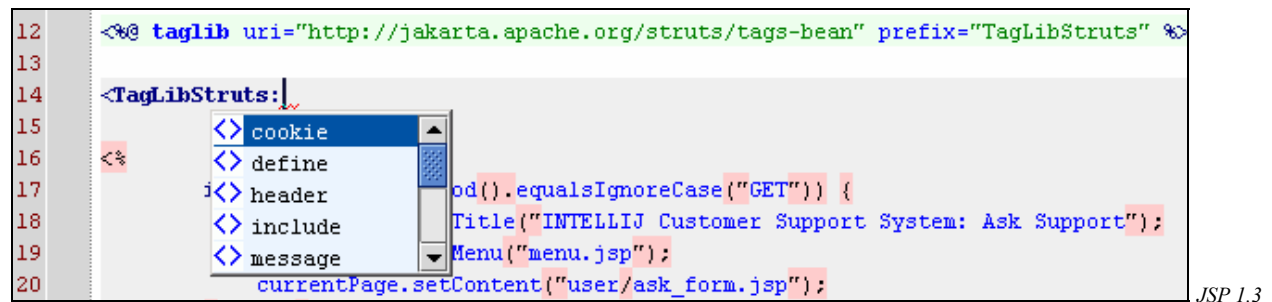
Once the selected attribute has been chosen from the automated attribute list, IDEA will automatically complete the JSP tag by filling in all necessary static data.

As shown in figure *JSP 1.2*, any part of a tag that allows 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 1.3*.

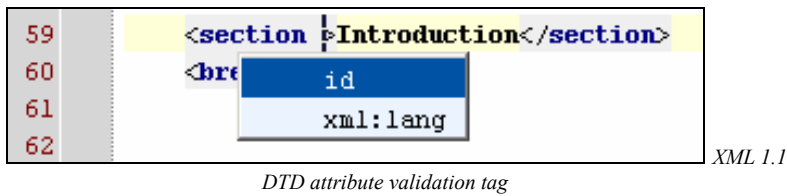


XML Development

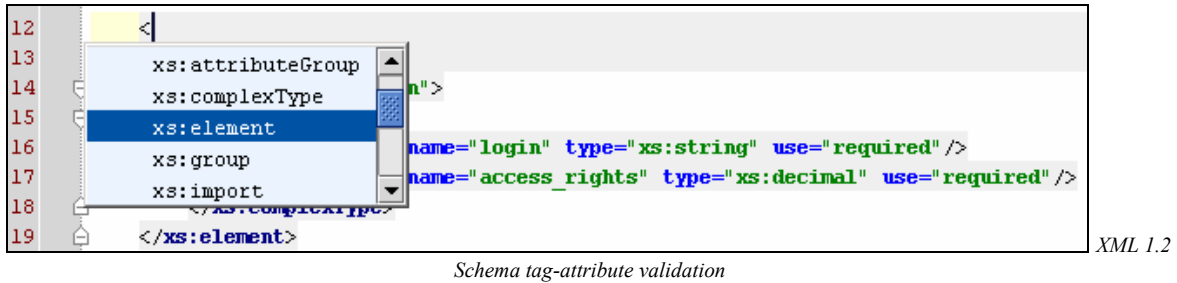
XML needs no introduction, or it shouldn't anyway. If you have ever done any extensive programming in Java, you have probably run into and used XML, if for nothing else to create Ant build files for faster application deployment. For more extensive J2EE development XML is utilized for multiple purposes: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

Whatever your specific case may be, if you are going to be deploying Java applications that work in conjunction with XML, IDEA comes equipped to help you create applications quicker and more efficiently. How is this possible? Simple: Not only does IDEA's editor know Java, it also enables you to meet the demands of XML coding with its smart editing features (including automated error highlighting).

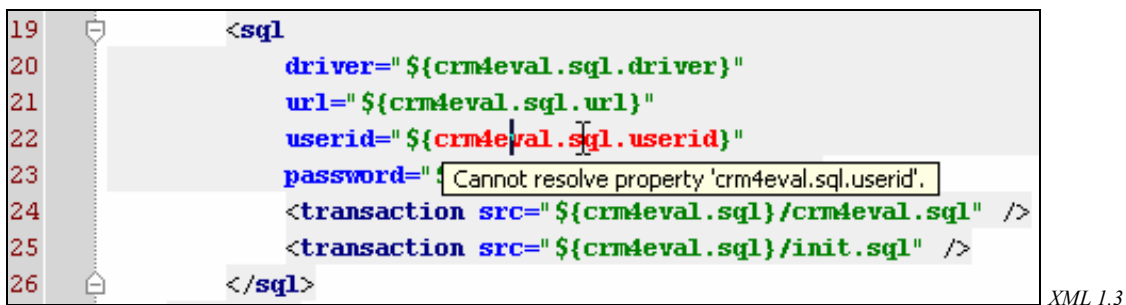
For example, IDEA allows you to quickly edit XML documents that support both DTD and Schema validation. As shown in figure *XML 1.1*, IDEA can digest any given DTD's specification and automatically include these special attributes into the editor's intelligent XML attribute completion function.



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



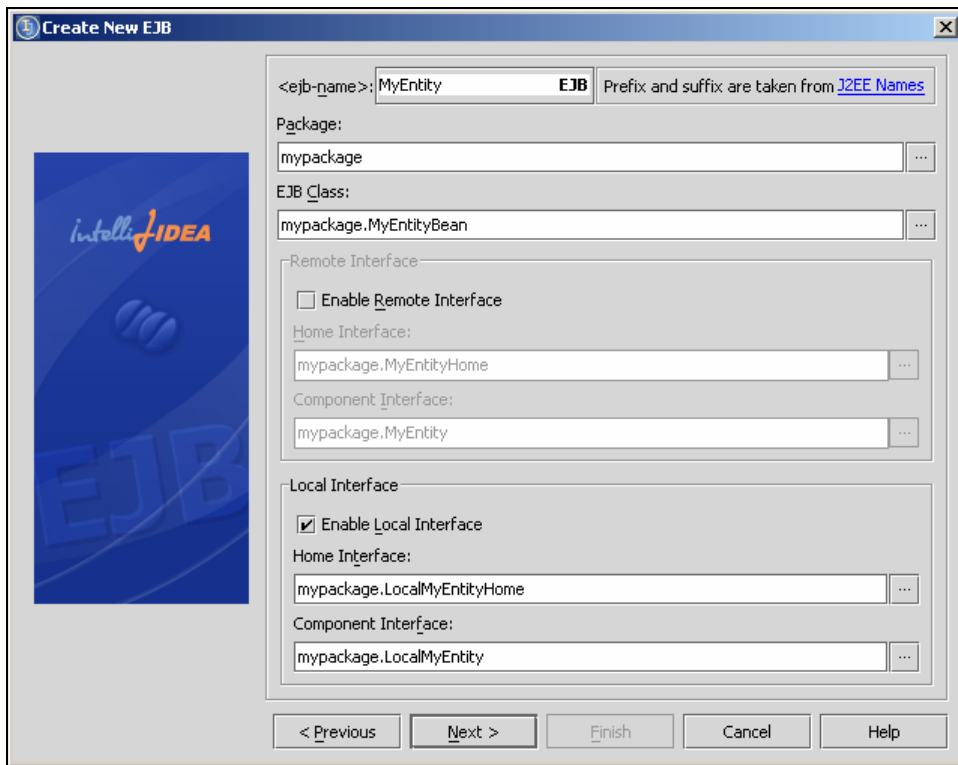
In addition to the aforementioned features, IDEA also incorporates a XML error high-lighting function. As shown in figure XML 1.3, if an error occurs in the XML code, IDEA will color-code the errors making them easy to find and fix.



EJB Development

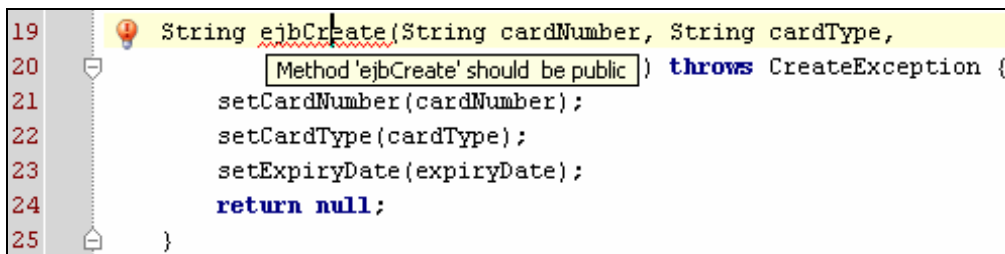
For those developers who are looking for a set of tools to aid you in much more complicated, robust, and over all time consuming enterprise centered development projects – or – in other words, you need to crank out a plethora of EJBs under a deadline or simply want to create EJBs that are flexible, scalable,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

For starters, IDEA's EJB wizard helps you create new beans to get you up and going quickly as shown in figure EJB 1.1.



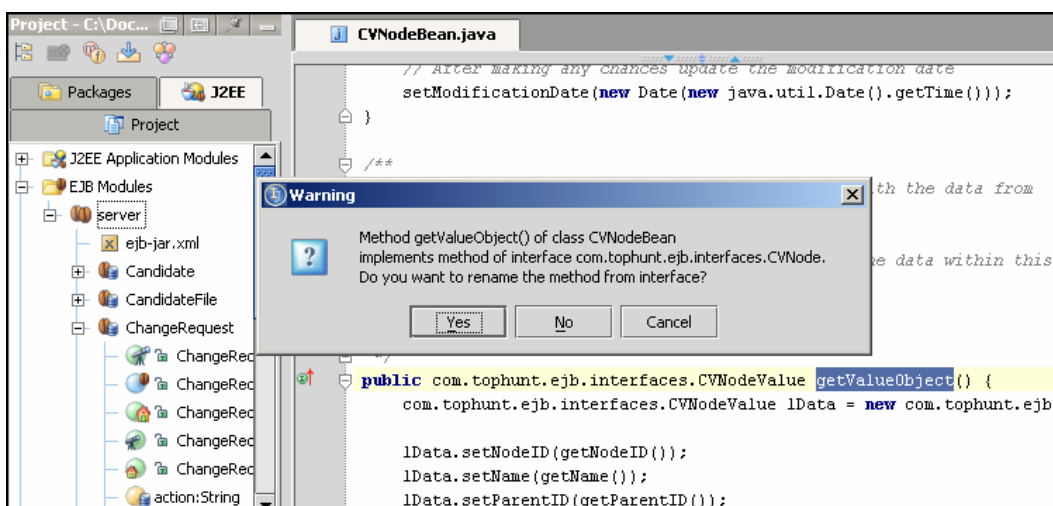
EJB 1.1

Once your bean has been created, IDEA will also monitor your EJB code with its integrated error highlighting function. Red is the magic color: major errors that prevent the deployment of your EJB will be shown in red, including compatibility errors and errors in any of the deployment descriptors.



EJB 1.2

IDEA's advanced refactoring support also works during EJB development as shown in figure EJB 1.3.



EJB 1.3

Collaboration Tools

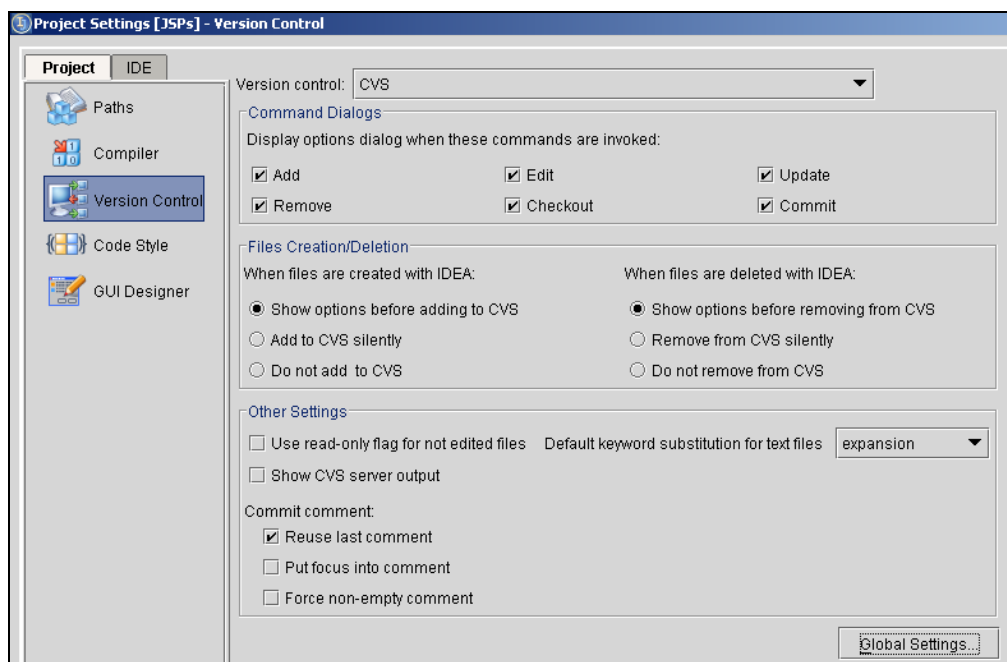
If you have read through the Overview up to this point, it is probably safe for us to assume that you are now pretty familiar with IDEA and have a grasp of the firepower it packs with its multitude of powerful features and functions that, among a gazillion other things, hasten development, clean up your code, and increase 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 industry has come to depend on.

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

CVS Integration

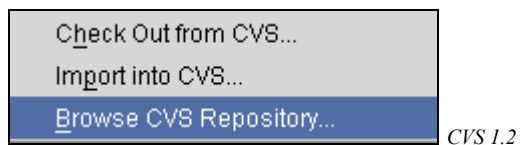
IDEA not only helps you develop and design code faster and more intelligently -- it also helps you manage and organize your projects for greater work efficiency. IDEA comes standard with a powerful **CVS (Concurrent Version System)** to help you manage revisions to any project's source code files. As shown in figure *CVS 1.1*, IDEA's CVS control panel is very user friendly. The administration console allows you to set various criteria related to CVS operation.



CVS 1.1

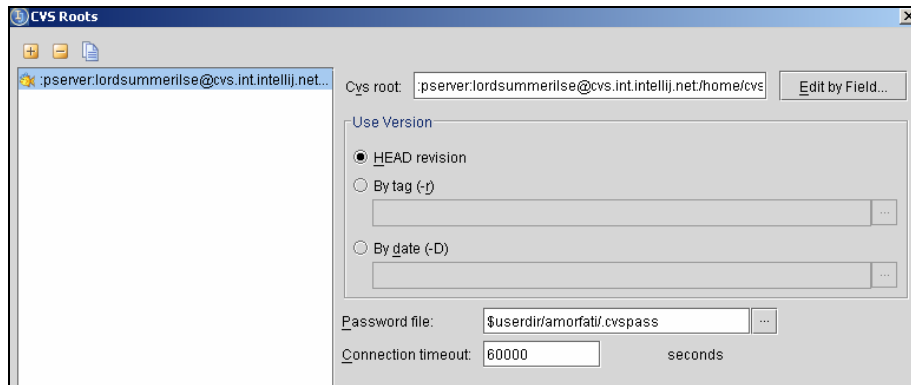
CVS preference administration console

Once your CVS preferences have been set, you are now ready to use the CVS tool itself. Under **File** on the main menu, you can see three CVS menu items as shown in figure *CVS 1.2*.



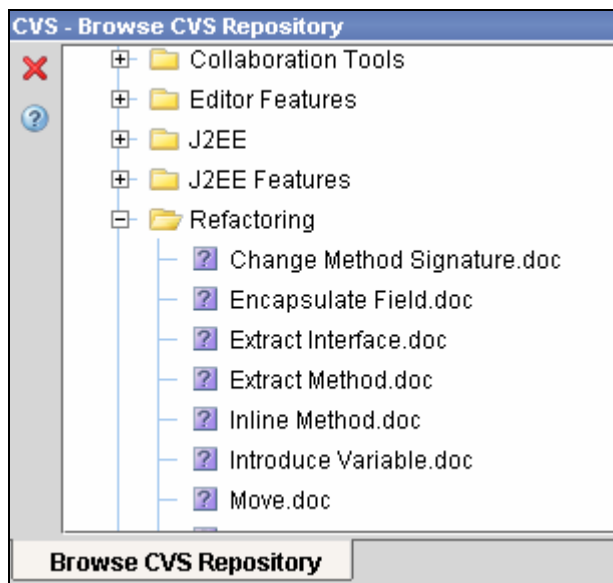
CVS 1.2

During initial set up, you may select any of the 3 menu items to invoke the configuration panel as shown in *CVS 1.3*. Once shown, you should configure the panel according to your personal set up and then select the **Test Configuration** button to test and confirm the connection to the CVS system.



CVS 1.3

If the connection test is successful, select **OK** to close the configuration panel. You may now connect to the CVS by again selecting **OK** on the CVS Root Configuration pane. Once you are successfully connected, you will see the directory hierarchy of your CVS folders as shown in figure *CVS 1.4*.



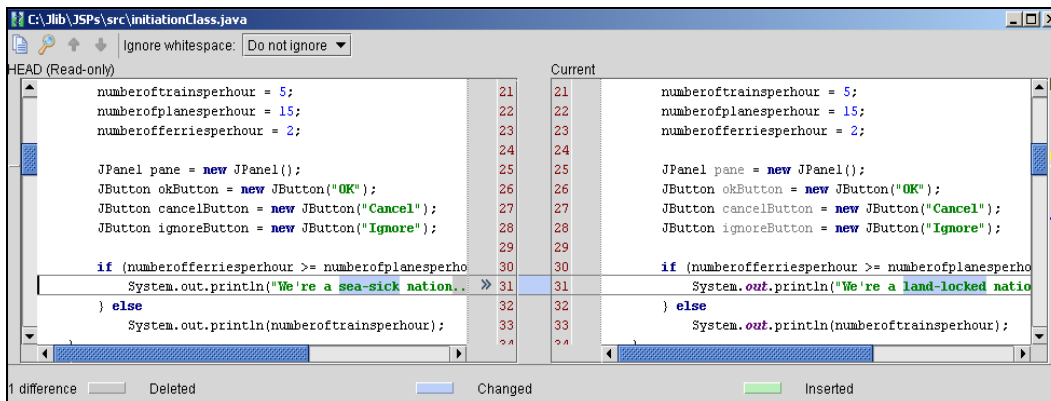
CVS 1.4

While browsing, you can right-click any file item in the hierarchy to read the comments associated with the last version checked in, or you can check the file out directly. By following the previous other two main menu items, *Import into CVS* and *Check Out from CVS*, you will be presented with similar options that are easy to understand and follow.



CVS Integration 1.5

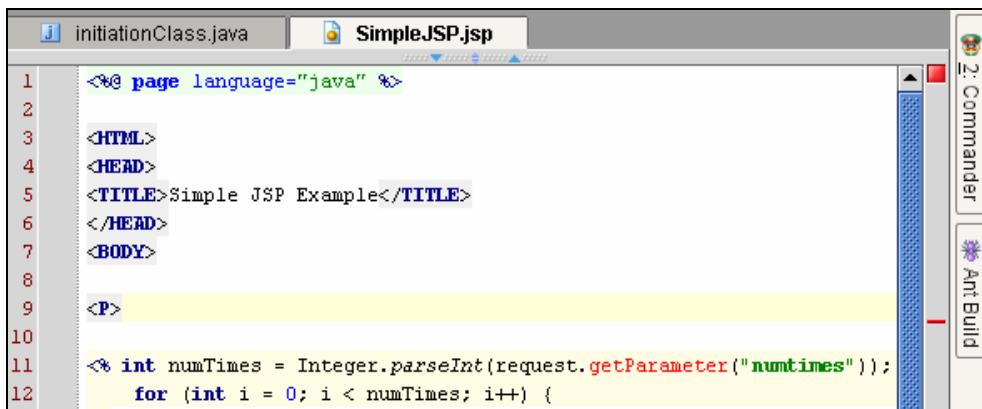
Any change made to source files that have been added to the CVS will be highlighted blue as shown in figure *CVS 1.5*. When something has been changed, but not yet checked-in, you can compare versions and be alerted to any changes that may have taken place by color coded highlights as shown in figure *CVS 1.6*.



CVS 1.6

Jakarta Ant

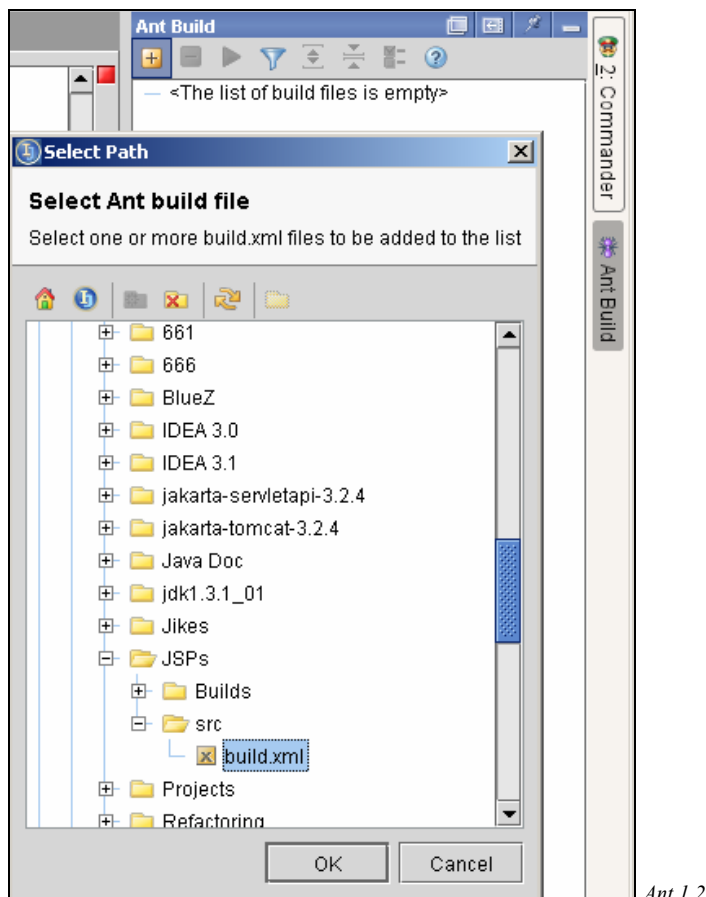
Those of you who depend on Jakarta Ant will find seamless integration of this powerful build tool into IntelliJ IDEA. To utilize Ant, open a project in IDEA and then open the Ant Build panel as shown in figure *Ant 1.1*.



Ant 1.1

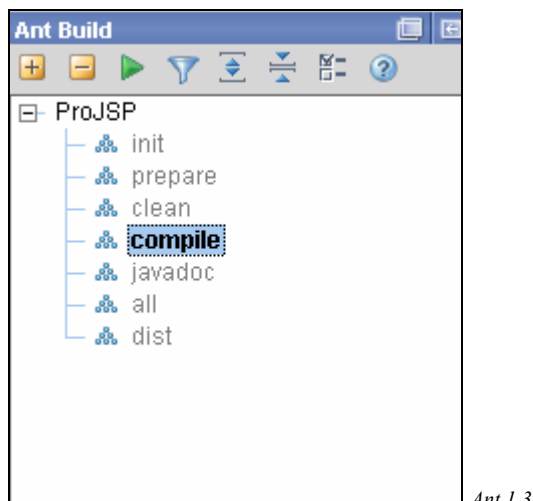
Select the Ant Build panel which is by default set on the right side of the IDE editor pane vertically

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



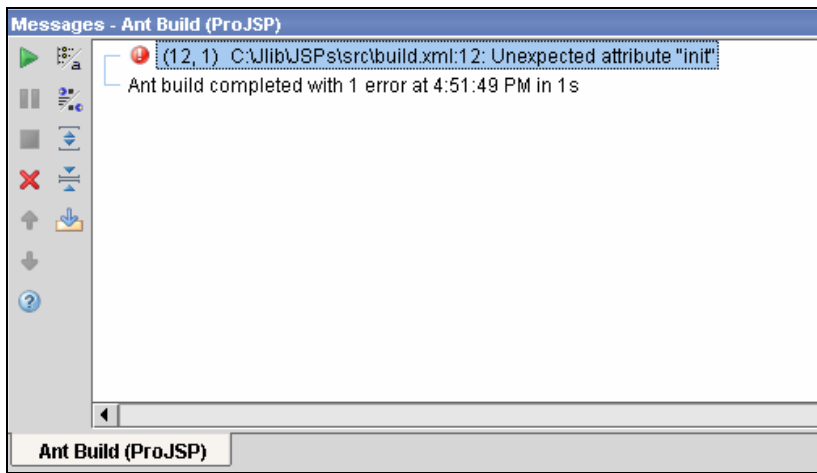
Selecting the desired Ant build file

As shown in figure *Ant 1.3*, once you have selected your build file (see figure *Ant 1.2*), a navigation window will appear outlining the build file's sequence of events that 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.



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 *Ant 1.4*, you can see these error messages and quickly navigate to their respective locations in the source code, make corrections, and restart the build process again.



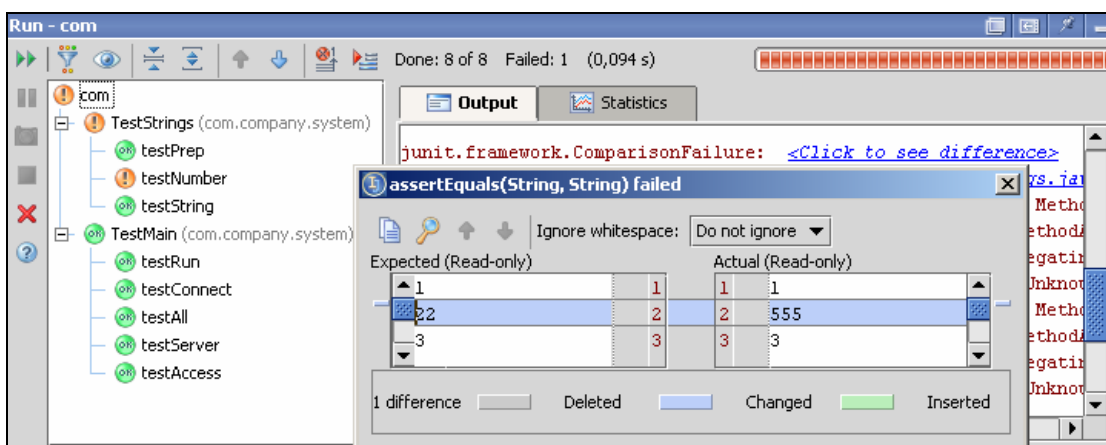
Ant 1.4

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 but 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. IDEA has an easy to setup and configure JUnit control panel that helps you quickly run unit tests directly from IDEA. You just simply invoke a test case method near your intended target object and the results of the test will be visible in an output pane. As shown in figure *JUnit 1.1*, the results in the output pane are easy to read and interpret. The output pane also allows you to quickly navigate to troubled areas and make immediate corrections in your source.



JUnit 1.1

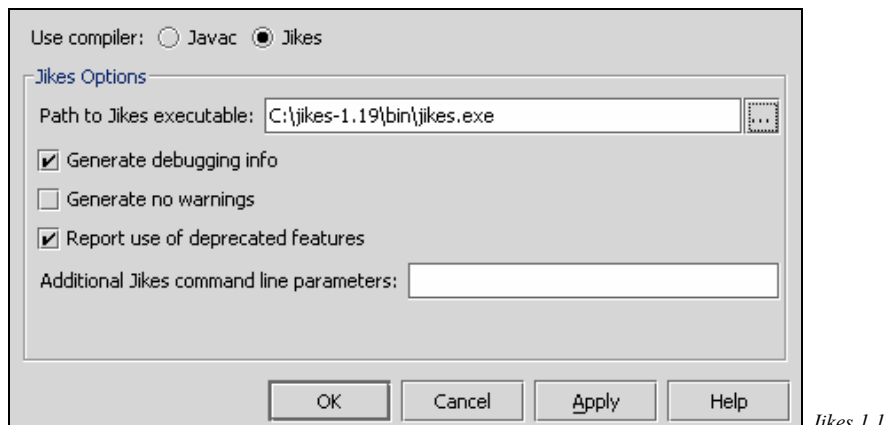
Jikes

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 compiler that translates Java™ source files as defined in The Java Language Specification into the bytecoded instruction set and binary format defined in The Java Virtual Machine Specification.

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 you 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.



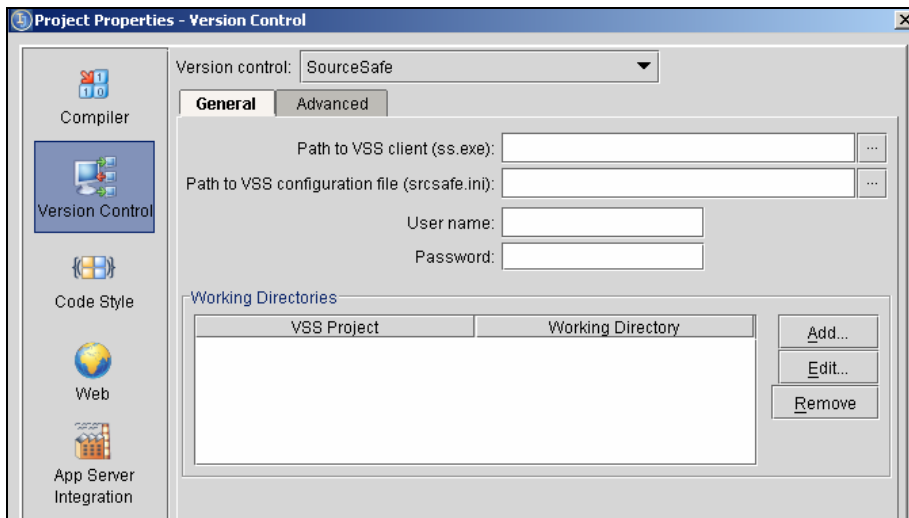
Select Jikes radio button and point to Jikes path to set up

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.

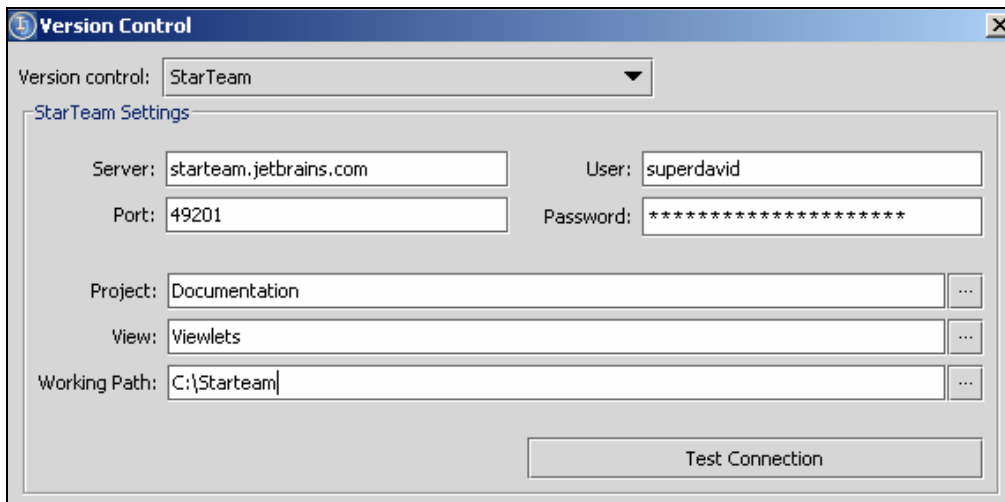
As shown in figure *Visual SourceSafe 1.1*, IDEA incorporates an easy to use and set up SourceSafe control panel allowing you to quickly set up and begin to utilize your SourceSafe installation within minutes.



Visual SourceSafe 1.1

StarTeam

Borland's StarTeam® is another handy application for sharing and managing development code and project responsibilities, and like Visual SourceSafe, it easily integrates with IDEA for effective cross-application collaboration. As with the previous figure *Visual SourceSafe 1.1*, simply select StarTeam under the *Version control:* drop down. Here you will be instructed to copy the **starteam-sdk.jar** file into your IDEA *distribution/lib* folder. Once the starteam-sdk.jar has been copied over, you will be able to configure your StarTeam setup as shown in figure *StarTeam 1.1*.



StarTeam 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>

StarTeam: <http://www.borland.com/starteam/>

Open API

After getting acquainted with IDEA, you will quickly realize that it comes with a hefty selection of development features and integrations that will more than satisfy the most “tool hungry” developers out there. However, in case IDEA doesn’t have a feature you want or lacks integration with some obscure tool, you have the opportunity to add such feature or integration yourself.

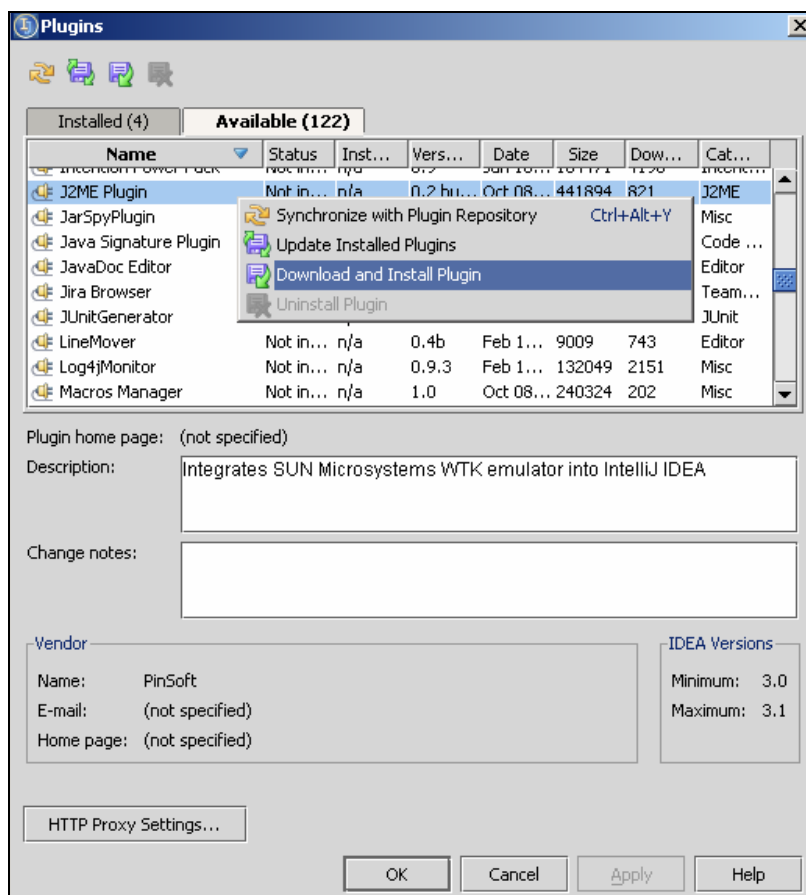
Third party developers will be happy to know that their application’s functions can be called directly from IDEA. In addition, they can incorporate a number of IDEA’s features directly into their own applications. From a developer’s perspective the Open API gives access to a whole new “eco-system” of development tools that accommodate and enhance IDEA’s already industry setting capabilities.

IntelliJ IDEA Developer Community

Users interested in coding their own plug-ins for IDEA or extending some of IDEA’s functionality into their own applications, are encouraged to check out the IntelliJ Community website at: www.intellij.org Here one can find among other things a large and growing list of plug-ins for IDEA, most of which are free to the public for use and many times open source.

Automated Plugin Installation and Update Tool

In case IDEA does not come with a specific tool you desire, you are now aware that you can always code your own plugin for IDEA, or alternatively download one of the many free tools available from the **IntelliJ Developer Community**. If you wish to download additional tools, you can now do so with IDEA’s built-in automated plugin installation and update tool. As shown in figure *Plugins 1.1*, just open IDEA’s **File | Setting | IDE Settings | Plugins** tab, and see the plugin icon on the left menu.



Plugins 1.1

After the **Available** tab has been selected, a list of all the downloadable plugins for IDEA will appear. Once the list has loaded, right-click on a specific plugin you want, and select either *Download and Install* or *Update Installed* plugin.

Once the plugins you have installed have been downloaded, restart IDEA and they will then be automatically deployed by IDEA and ready for use.

Free Community Plugins

Just to give you a taste of some of the real cool and useful plugins that members of the IntelliJ community are developing, we would like to introduce 3 of the more widely used and popular plugins available through IDEA's automated plugin installer:

InspectionGadgets

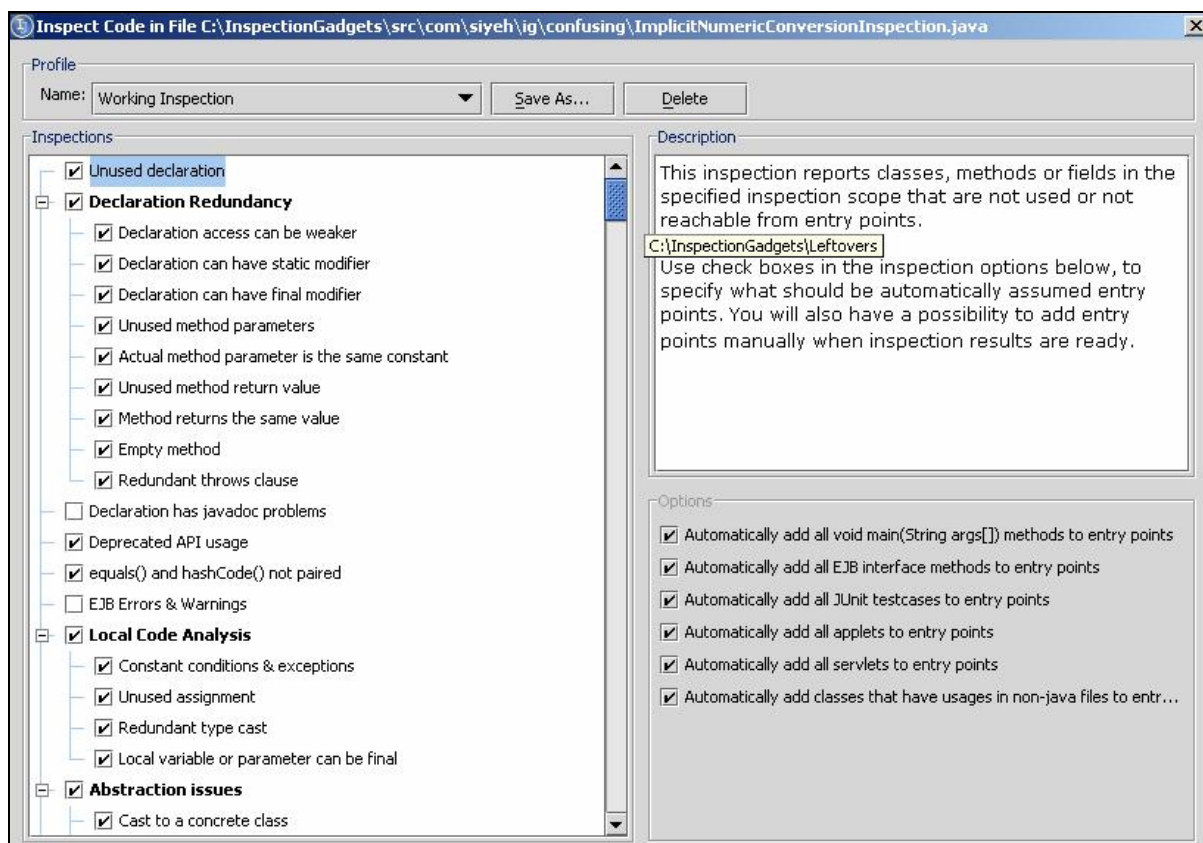
The **InspectionGadgets** plug-in extends IDEA's built-in code inspection and error reporting functionality with over 270 new code inspections, creating a super powerful and productive code analysis environment.

Once InspectionGadgets has been installed, IDEA will automatically search through your code for common errors, code weaknesses, and places for improvement. InspectionGadgets provides further inspections in the following categories:

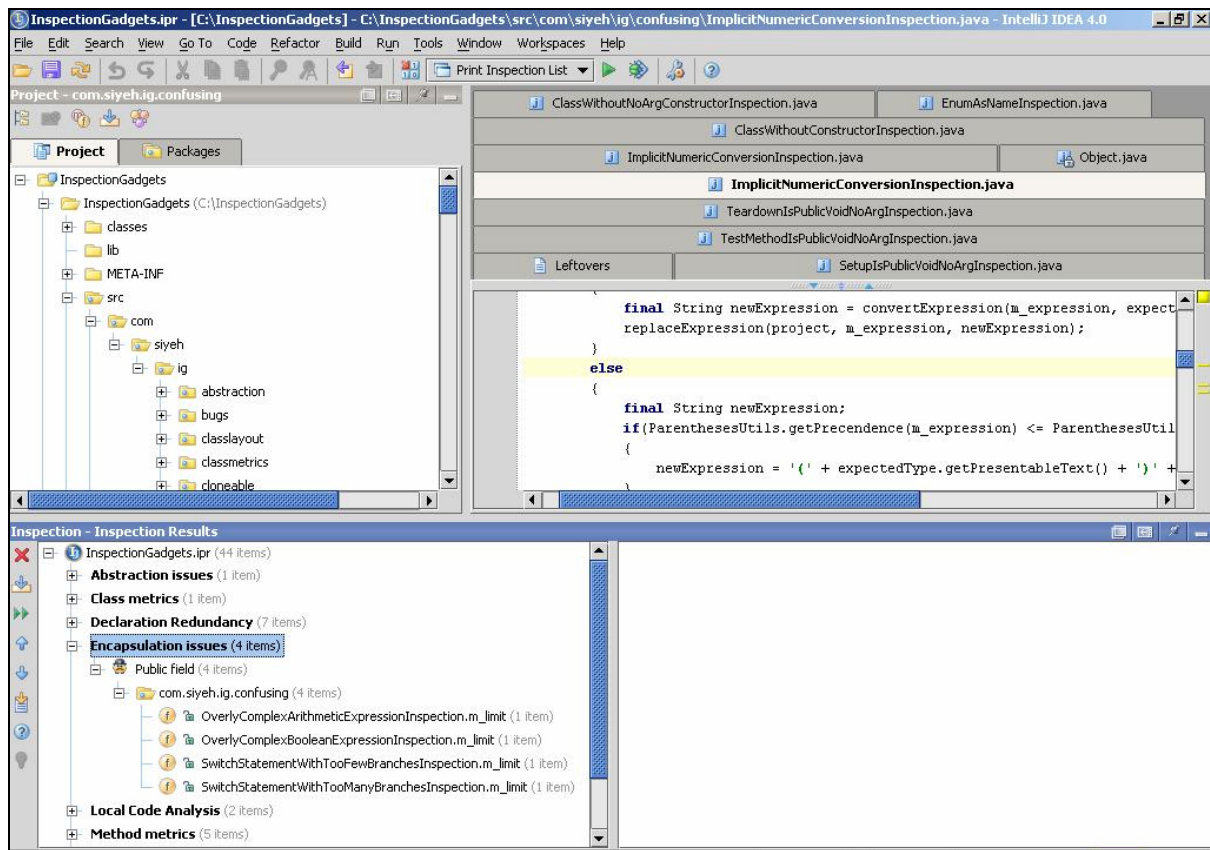
- performance problems
- confusing and error-prone code constructs
- common bugs
- design flaws
- initialization problems

- naming conventions
- threading issues
- internationalization problems
- style issues
- portability concerns
- Common JUnit errors
- class and method metrics (size, algorithmic complexity, coupling)
- many more...

While InspectionGadgets is comparable with the best commercial and open-source static-analysis tools in terms of number of inspections reported, its real value comes in the tight coupling of static analysis and code editing. With InspectionGadgets and IDEA, errors can be shown during editing as “yellow-line” warnings, with tool-tips describing each error for easy comprehension. Errors can be quickly navigated to using “Find next error” (F2) function, so that they can be easily fixed. Even more impressive, fifty of the inspections come with “quick fixes”, which let the user automatically fix the error with a keystroke. Add it all up, and the pairing of IDEA and InspectionGadgets can’t be beat for either finding bugs or preventing them.



Over 270 additional inspections with InspectionGadgets



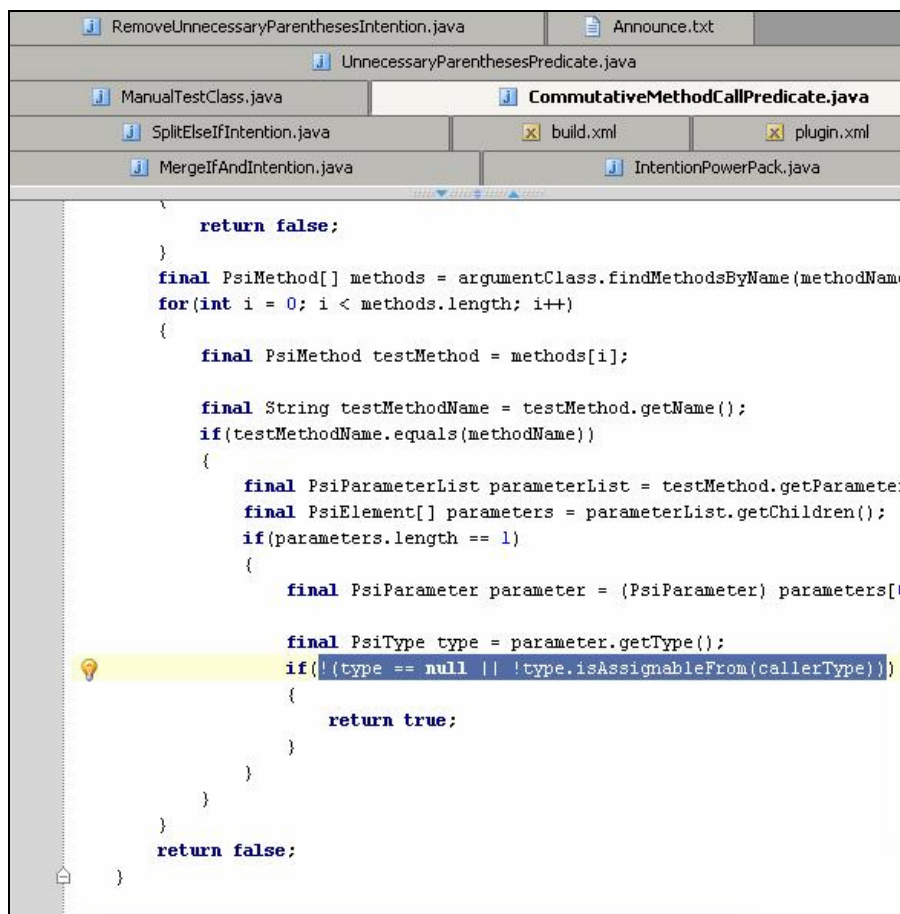
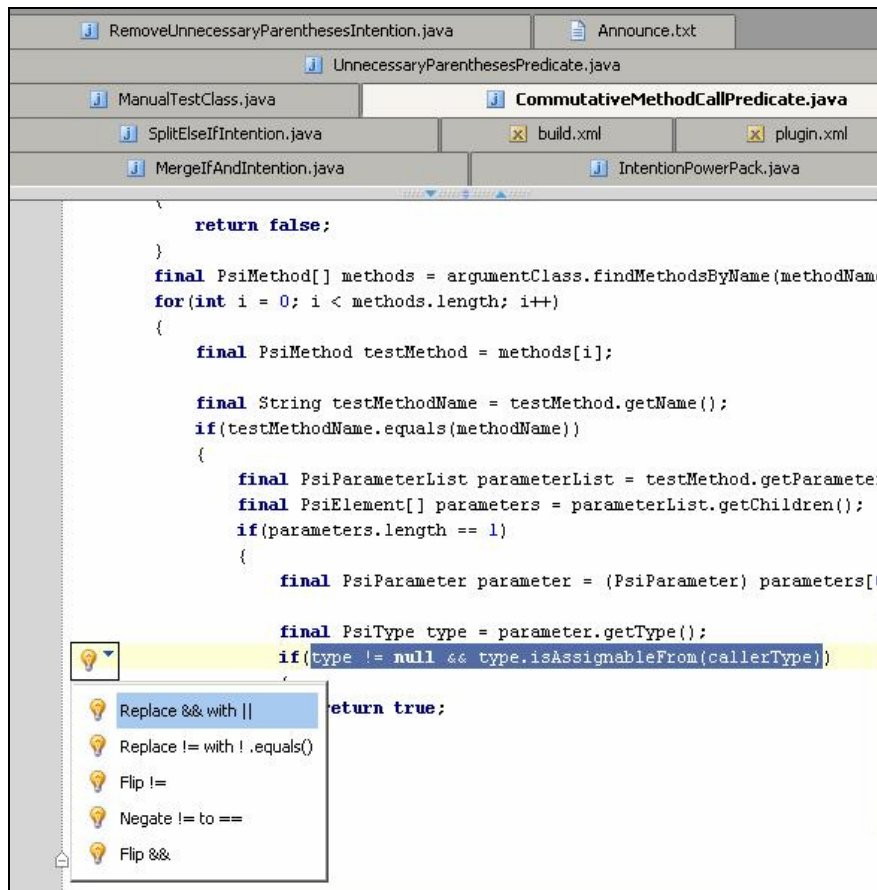
InspectionGadgets' easy to read and navigate inspection report

Intention Power-Pack

IDEA broke new ground in version 3.0 with “programming by intention”. With “programming by intention”, many simple programming actions or “intentions” were made available to the developer based on their current editing position. Clicking on the name of an abstract class, for instance, would bring up the “light bulb” icon, presenting the programmer with the option to create a new sub-class of it. The Intention Power-Pack plugin adds additional intentions to IntelliJ IDEA, automating a lot of common programming tasks. Intentions are provided by Intention Power-Pack to perform the following:

- Convert **&&** to **||**, and vice versa
- Reorder and simplify boolean expressions
- Convert “*equals()*” expressions to “**==**”, and vice versa
- Replace switch statements with if statements, and vice versa
- Convert ternary conditional expression (**?:**) to if-then-else statements, and vice versa
- Translate integer literals between decimal, octal, and hexadecimal
- Replace simple assignments with operator assignments
- “Flip” boolean operations, numeric comparisons, and commutative method calls.

The following two figures show the replacement of `&&` to `||` with Intention Power-Pack:



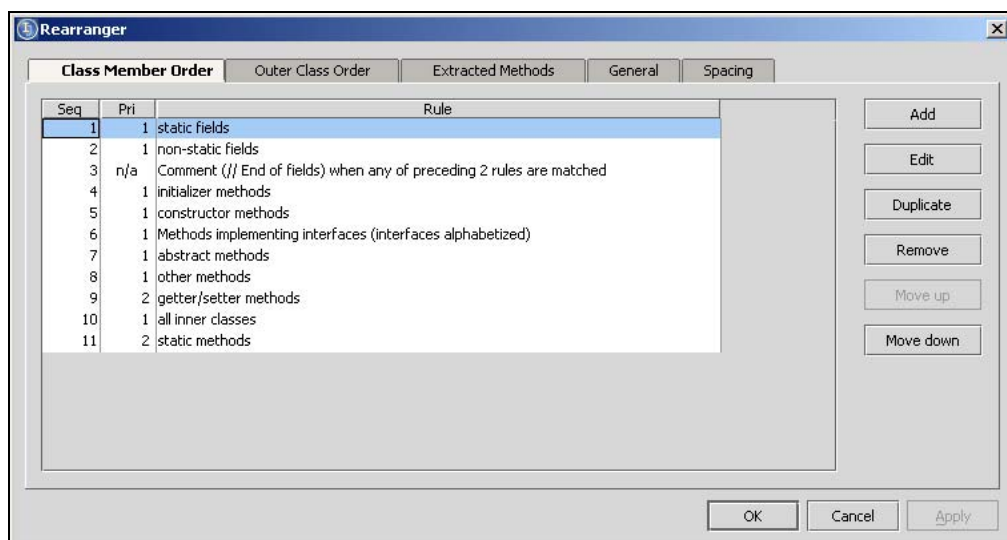
Rearranger

The Rearranger plugin rearranges (reorders) class and class member declarations according to a user-specified order and/or method call hierarchy. For example, take the following sample class in figure *Rearranger 1.1*:

```
1  +/.../
11 package com.wrq.rearranger;
12
13 /**
14  * Illustrates rearrangement by the plugin.
15  */
16 abstract public class RearrangeMe
17 {
18     abstract boolean isGood(); // matches rule 7, "abstract methods"
19     /**
20      * field1 is documented here. This documentation moves with the field.
21      */
22     int field1; // matches rule 2, "non-static fields"
23     private void method2() {} // called by "method()" -- should appear just below it
24     protected RearrangeMe(int field1) // matches rule 5, "constructor methods"
25     {
26         this.field1 = field1;
27     }
28     public void method() // matches rule 8, "other methods"
29     {
30         method2(); // calls another method
31     }
32     public static final int CONSTANT = 1; // matches rule 1, "static fields"
33 }
```

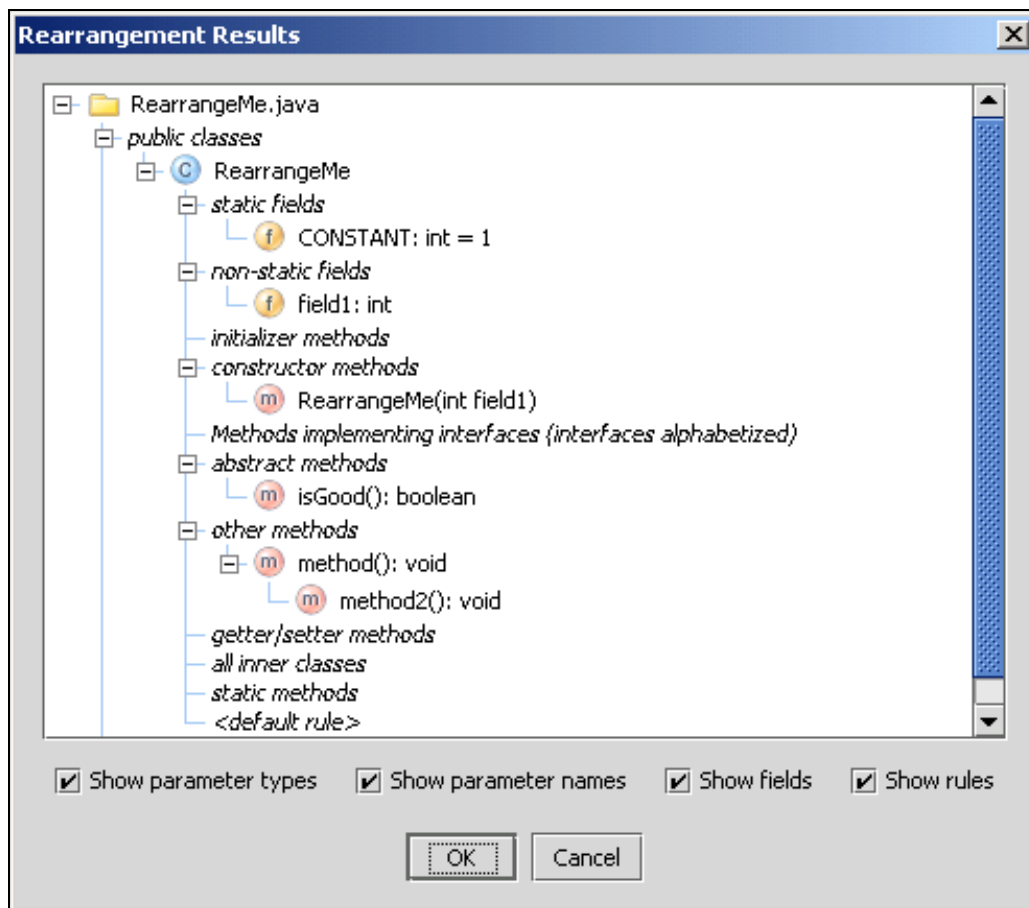
Rearranger 1.1

To use Rearranger, just invoke its control panel and set specific rearrangement rules as shown in figure *Rearranger 1.2*



Rearranger 1.2

The plugin can optionally display the proposed rearrangement in a popup similar to File Structure (Ctrl-F12), but showing the items matching each rule, and showing the method hierarchy as shown in figure *Rearranger 1.3*



Rearranger 1.3

In this example case, the Rearranger was configured to move called methods below their callers; this is why *method2()* appears just below *method1()*.

Once the rearrangement process has been initiated, the resulting source will appear as shown in figure *Rearranger 1.4*.

```

12
13 /**
14  * Illustrates rearrangement by the plugin.
15  */
16 abstract public class RearrangeMe
17 {
18     public static final int CONSTANT = 1; // matches rule 1, "static fields"
19     /**
20      * field1 is documented here. This documentation moves with the field.
21      */
22     int field1; // matches rule 2, "non-static fields"
23     // End of fields
24     protected RearrangeMe(int field1) // matches rule 5, "constructor methods"
25     {
26         this.field1 = field1;
27     }
28
29     abstract boolean isGood(); // matches rule 7, "abstract methods"
30
31     public void method() // matches rule 8, "other methods"
32     {
33         method2(); // calls another method
34     }
35
36     private void method2() {} // called by "method()" -- should appear just below it
37 }

```

Rearranger 1.4

The Rearranger is also able to generate comments conditionally (see **rule 3** and **line 23**), and will remove old comments before rearranging so that generated comments aren't duplicated.

To get more information about the above three community plugins, please see www.intellij.org for more, in-depth descriptions of each plugin and their related features and functions.

Conclusion

After reading through this Overview, you should have a clear conception of the main features and functions IDEA 4.0 comes equipped with. What you cannot garner from reading this Overview, however, is a better understanding of how it actually *feels* to use IDEA; what cannot be written on paper or in electronic form will certainly be assuaged by actually using IDEA. Many other IDEA features which we did not cover in this Overview were not covered because there is no way to manually invoke them, they are autonomic, and like the beat of a heart, they just work as you code. It is therefore imperative, that once one has read the Overview to get an idea, they follow up by trying IDEA to also get the feeling. To have this complete experience, there is only one thing left to do, and that is to download a free evaluation of IDEA and to try it. After all, what do you have to lose besides your old IDE?

Download: <http://www.jetbrains.com/idea/download/index.html>

- *Finis* -