

NetBeans™ IDE Field Guide: Developing Desktop, Web, Enterprise, and Mobile Applications, Second Edition

By Patrick Keegan, Ludovic Champenois,
Gregory Crawley, Charlie Hunt, Christopher Webster,
John Jullion-Ceccarelli, Jiri Prazak, Martin Ryzl,
Gregg Sporar, Geertjan Wielenga



Publisher: **Prentice Hall**

Pub Date: **May 09, 2006**

Print ISBN-10: **0-13-239552-5**

Print ISBN-13: **978-0-13-239552-6**

Pages: **424**

[Table of Contents](#) | [Index](#)

Overview

The Only Complete Guide and Reference for NetBeans™ IDE 5.0

The award-winning NetBeans™ IDE eases all aspects of Java application development, incorporating a wide range of powerful features into one well-designed package. NetBeans IDE is consistently first in supporting the latest Java technologies for developing desktop, web, enterprise, and mobile applications.

NetBeans™ IDE Field Guide

provides an introduction to the IDE and an extensive range of topics to help you with both everyday and advanced programming tasks, including

- Taking advantage of the Ant-based project system to create easily deployable projects
- Developing web applications with the built-in Apache Tomcat web server
- Constructing, assembling, and verifying large-scale Java EE applications
- Managing the Sun Java System Application Server through NetBeans IDE
- Developing mobile applications with the NetBeans Mobility Pack
- In this expanded second edition, you can also learn how to
- Build powerful and attractive desktop applications with the Matisse GUI Builder

- Profile your applications for performance issues
- Develop modules for NetBeans IDE and rich-client applications based on the NetBeans Platform
- Chat and share code with other developers using the NetBeans Collaboration Modules

NetBeans™ IDE Field Guide: Developing Desktop, Web, Enterprise, and Mobile Applications, Second Edition

By Patrick Keegan, Ludovic Champenois,
Gregory Crawley, Charlie Hunt, Christopher Webster,
John Jullion-Ceccarelli, Jiri Prazak, Martin Ryzl,
Gregg Sporar, Geertjan Wielenga



Publisher: **Prentice Hall**

Pub Date: **May 09, 2006**

Print ISBN-10: **0-13-239552-5**

Print ISBN-13: **978-0-13-239552-6**

Pages: **424**

[Table of Contents](#) | [Index](#)

[Copyright](#)

[Foreword to the First Edition](#)

[Foreword to the First Edition](#)

[Preface](#)

[About the Authors](#)

[Acknowledgments for the Second Edition](#)

[Acknowledgments for the First Edition](#)

[Chapter 1. Download, Installation, and First Project](#)

[Downloading the IDE](#)

[Installing the IDE](#)

[Setting a Proxy](#)

[First NetBeans IDE Project](#)

[Chapter 2. NetBeans IDE Fundamentals](#)

[Creating a Project](#)

[Configuring the Classpath](#)

[Creating a Subproject](#)

[Creating and Editing Files](#)

[Setting Up and Modifying Java Packages](#)

[Compiling and Building](#)

[Viewing Project Metadata and Build Results](#)

[Navigating to the Source of Compilation Errors](#)

[Running](#)

[Creating and Running Tests](#)

[Debugging the Application](#)

[Integrating Version Control Commands](#)

[Managing IDE Windows](#)

[Chapter 3. IDE Project Fundamentals](#)

[Introduction to IDE Projects](#)

[Choosing the Right Project Template](#)

[Creating a Project from Scratch](#)

[Importing a Project Developed in a Different Environment](#)

[Navigating Your Projects](#)

[Working with Files Not in the Project](#)

[Creating Packages and Files in the Project](#)

[Configuring the Project's Classpath](#)

[Changing the Version of the JDK Your Project Is Based On](#)

[Changing the Target JDK for a Standard Project](#)

[Referencing JDK Documentation \(Javadoc\) from the Project](#)

[Adding Folders and JAR Files to the Classpath](#)

[Making External Sources and Javadoc Available in the IDE](#)

[Structuring Your Projects](#)

[Displaying and Hiding Projects](#)

[Compiling a Project](#)

[Running a Project in the IDE](#)

[Deploying a Java Project Outside of the IDE](#)

[Building a Project from Outside of the IDE](#)

[Customizing the IDE-Generated Build Script](#)

[Running a Specific Ant Target from the IDE](#)

[Completing Ant Expressions](#)

[Making a Menu Item or Shortcut for a Specific Ant Target](#)

[Chapter 4. Versioning Your Projects](#)

[Setting up CVS in NetBeans IDE](#)

[Checking Out Sources from a CVS Repository](#)

[Putting a Project into CVS](#)

[Keeping Track of Changes](#)

[Updating Files](#)

[Committing Changes](#)

[Ignoring Files in CVS Operations](#)

[Adding and Removing Files from a Repository](#)

[Working with Branches](#)

[Working with Patches](#)

[Working with Versioning Histories](#)

[Working with Other Version Control Systems](#)

[Chapter 5. Editing and Refactoring Code](#)

[Opening the Source Editor](#)

[Managing Automatic Insertion of Closing Characters](#)

[Displaying Line Numbers](#)

[Generating Code Snippets without Leaving the Keyboard](#)

[Using Code Completion](#)

[Inserting Snippets from Code Templates](#)

[Using Editor Hints to Generate Missing Code](#)

[Matching Other Words in a File](#)

[Generating Methods to Implement and Override](#)

[Generating JavaBeans Component Code](#)

[Creating and Using Macros](#)

[Creating and Customizing File Templates](#)

[Handling Imports](#)

[Displaying Javadoc Documentation While Editing](#)

[Formatting Code](#)

[Text Selection Shortcuts](#)

[Navigating within the Current Java File](#)

[Navigating from the Source Editor](#)

[Searching and Replacing](#)

[Deleting Code Safely](#)

[Changing a Method's Signature](#)

[Encapsulating a Field](#)

[Moving a Class to a Different Package](#)

[Moving Class Members to Other Classes](#)

[Creating a Method from Existing Statements](#)

[Creating an Interface from Existing Methods](#)

[Extracting a Superclass to Consolidate Common Methods](#)

[Changing References to Use a Supertype](#)

[Unnesting Classes](#)

[Tracking Notes to Yourself in Your Code](#)

[Comparing Differences Between Two Files](#)

[Splitting the Source Editor](#)

[Maximizing Space for the Source Editor](#)

[Changing Source Editor Keyboard Shortcuts](#)

[Chapter 6. Building Java Graphical User Interfaces](#)

[Using Different Layout Managers](#)

[Placing and Aligning a Component in a Form](#)

[Setting Component Size and Resizability](#)

[Setting Component Alignment](#)

[Specifying Component Behavior and Appearance](#)

[Generating Event Listening and Handling Methods](#)

[Customizing Generated Code](#)

[Previewing a Form](#)

[Using Custom Beans in the Form Editor](#)

[Deploying GUI Applications Developed with Matisse](#)

[Chapter 7. Debugging Java Applications](#)

[Starting a Debugging Session](#)

[Attaching the Debugger to a Running Application](#)

[Starting the Debugger Outside of the Project's Main Class](#)

[Stepping through Code](#)

[Setting Breakpoints](#)

[Managing Breakpoints](#)

[Customizing Breakpoint Behavior](#)

[Monitoring Variables and Expressions](#)

[Backing up from a Method to Its Call](#)

[Monitoring and Controlling Execution of Threads](#)

[Fixing Code During a Debugging Session](#)

[Viewing Multiple Debugger Windows Simultaneously](#)

[Chapter 8. Developing Web Applications](#)

[Representation of Web Applications in the IDE](#)

[Adding Files and Libraries to Your Web Application](#)

[Editing and Refactoring Web Application Files](#)

[Deploying a Web Application](#)

[Testing and Debugging Your Web Application](#)

[Creating and Deploying Applets](#)

[Changing the IDE's Default Web Browser](#)

[Monitoring HTTP Transactions](#)

[Chapter 9. Creating Web Applications on the JSF and Struts Frameworks](#)

[JSF Overview](#)

[Struts Overview](#)

[Chapter 10. Introduction to Java EE Development in NetBeans IDE](#)

[Configuring the IDE for Java EE Development](#)

[Java EE Server Support](#)

[Getting the Most from the Java BluePrints Solutions Catalog](#)

[Chapter 11. Extending Web Applications with Business Logic: Introducing Enterprise Beans](#)

[EJB Project Template Wizards](#)

[Adding Enterprise Beans, Files, and Libraries to Your EJB Module](#)

[Adding Business Logic to an Enterprise Bean](#)

[Adding a Simple Business Method](#)

[Enterprise Bean Deployment Descriptors](#)

[Chapter 12. Extending Java EE Applications with Web Services](#)

[Consuming Existing Web Services](#)

[IDE and Server Proxy Settings](#)

[Creating a WSDL File](#)

[Implementing a Web Service in a Web Application](#)

[Implementing Web Services within an EJB Module](#)

[Testing Web Services](#)

[Adding Message Handlers to a Web Service](#)

[Chapter 13. Developing Full-Scale Java EE Applications](#)

[Creating Entity Beans with the Top-Down Approach](#)

[Creating Entity Beans with the Bottom-Up Approach](#)

[Assembling Enterprise Applications](#)

[Importing Existing Enterprise Applications](#)

[Consuming Java Enterprise Resources](#)

[Java EE Platform and Security Management](#)

[Understanding the Java EE Application Server Runtime Environment](#)

[Ensuring Java EE Compliance](#)

[Refactoring Enterprise Beans](#)

[Database Support and Derby Integration](#)

[Chapter 14. Developing Java ME Mobile Applications](#)

[Downloading and Installing the Mobility Pack](#)

[Mobility Primer](#)

[Configuration vs. Configuration](#)

[Setting up Mobility Projects](#)

[Creating a Project from Scratch](#)

[Importing a Project](#)

[Physical Structure of Mobile Projects](#)

[Using Mobility File Templates](#)

[Configuring the Project's Classpath](#)
[Debugging Your Project](#)
[Configuring Your Project for Different Devices](#)
[Setting the Active Configuration for Your Project](#)
[Reusing Project Settings and Configurations](#)
[Structuring Project Dependencies](#)
[Managing the Distribution JAR File Content](#)
[Handling Project Resources for Different Configurations](#)
[Writing Code Specific to a List of Configurations](#)
[Using the Preprocessor](#)
[Using Configuration Abilities](#)
[Creating and Associating an Ability with a Configuration](#)
[Localizing Applications](#)
[Using the MIDP Visual Designer](#)
[Understanding the Flow Designer](#)
[Understanding the Screen Designer](#)
[Deploying Your Application Automatically](#)
[Incrementing the Application's MIDlet-Version Automatically](#)
[Using Ant in Mobility Projects](#)
[Using Headless Builds](#)
[Using the Wireless Connection Tools](#)
[Finding More Information](#)

[Chapter 15. Profiling Java Applications](#)
[Supported Platforms](#)
[Downloading and Installing the NetBeans Profiler](#)
[Starting a Profiling Session](#)
[The Profiler Control Panel](#)
[Monitoring an Application](#)
[Analyzing Performance](#)
[Analyzing Code Fragment Performance](#)
[Analyzing Memory Usage](#)
[Attaching the Profiler to a JVM](#)

[Chapter 16. Integrating Existing Ant Scripts with the IDE](#)
[Creating a Free-Form Project](#)
[Mapping a Target to an IDE Command](#)
[Setting up the Debug Project Command for a General Java Application](#)
[Setting up the Debug Project Command for a Web Application](#)
[Setting up Commands for Selected Files](#)

[Setting up the Compile File Command](#)

[Setting up the Run File Command](#)

[Setting up the Debug File Command](#)

[Setting up the Debugger's Apply Code Changes Command](#)

[Setting up the Profile Project Command for a General Java Application](#)

[Changing the Target JDK for a Free-Form Project](#)

[Making a Custom Menu Item for a Target](#)

[Debugging Ant Scripts](#)

[Chapter 17. Developing NetBeans Plug-in Modules](#)

[Plug-in Modules](#)

[Rich-Client Applications](#)

[Extending NetBeans IDE with Plug-in Modules](#)

[Setting up a Plug-in Module](#)

[Using the NetBeans APIs](#)

[Registering the Plug-in Module](#)

[Adding a License to a Plug-in Module](#)

[Building and Trying Out a Plug-in Module](#)

[Packaging and Distributing a Plug-in Module](#)

[Packaging and Distributing a Rich-Client Application](#)

[Finding Additional Information](#)

[Chapter 18. Using NetBeans Developer Collaboration Tools](#)

[Getting the NetBeans Developer Collaboration Tools](#)

[Configuring NetBeans IDE for Developer Collaboration](#)

[Creating a Collaboration Account](#)

[Managing Collaboration Accounts](#)

[Logging into a Collaboration Server](#)

[Collaborating and Interacting with Developers](#)

[Appendix A. Importing an Eclipse Project into NetBeans IDE](#)

[Getting the Eclipse Project Importer](#)

[Choosing Between Importing with and Importing without Project Dependencies](#)

[Importing an Eclipse Project and Preserving Project Dependencies](#)

[Importing an Eclipse Project and Ignoring Project Dependencies](#)

[Handling Eclipse Project Discrepancies](#)

[Handling Eclipse Project Reference Problems](#)

[Appendix B. Importing a JBuilder Project into NetBeans IDE](#)

[Getting the JBuilder Project Importer](#)

[Importing a JBuilder 2005 Project](#)

[Project Import Warnings](#)

[Running the Imported Project](#)
[Index](#)

Copyright

Sun Microsystems, Inc. has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents, foreign patents, or pending applications.

Sun, Sun Microsystems, the Sun logo, J2ME, J2EE, Java Card, and all Sun and Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

This publication is provided "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. This publication could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein; these changes will be incorporated in new editions of the publication. Sun Microsystems, Inc. may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Library of Congress Cataloging-in-Publication Data

NetBeans IDE field guide : developing desktop, web, enterprise,
2nd ed.

p. cm.

Includes index.

ISBN 0-13-239552-5 (pbk. : alk. paper)

1. Java (Computer program language) 2. Computer programm

QA76.73.J38N463 2006

005.13'3dc22

2006010947

Copyright © 2006 Sun Microsystems, Inc.

4150 Network Circle, Santa Clara, California 95054 U.S.A.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458
Fax: (201) 236-3290

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing, May 2006

Foreword to the First Edition

James Gosling

The NetBeans IDE has really come a long way in the last few years. Since the first book was written, NetBeans has progressed from a tool with promise (from a small, young company in the Czech Republic) to one of the market leaders in the open source IDE tools space. It's been like watching a child grow and mature over the years and blossom in ways you could have never predicted.

It's a bit like watching Java grow. At first it was a language for the Internet and browsers. It was so versatile, people started using it in many ways. It developed into a great language for writing multi-tier applications. And then with J2EE, it created a whole new ecosystem of enterprise applications. Later, J2ME conquered the phone and mobility market.

NetBeans has been through a similar, multifaceted growth. With NetBeans IDE 4.1, one tool can manage the range of Java development. The IDE now adds J2EE EJB and web services development to the rich suite of development capabilities that service J2SE and J2ME.

I use NetBeans for all my Java development. The exciting new language features in J2SE 5.0 are fun to use and easy to develop with. One of the things that's nice for me personally is that with each release, once I download it, it is ready to go. I don't have to go on a treasure hunt and assemble a particular set of plug-ins for me to begin development as soon as I install NetBeans, I'm ready to start coding.

The PR people at Sun like to call me "the Father of Java." Given that, NetBeans must be my first techno-grandchild. Enjoy all that NetBeans IDE 4.1 provides and the worlds it opens for you.

Happy programming.

James Gosling

May 2005

Foreword to the First Edition

Bill Shannon

The NetBeans IDE was the first free and open source tool to provide support for building J2EE web tier applications five years ago. With the 4.1 release, the NetBeans IDE has advanced even further to include full support for building complete J2EE 1.4 applications, including Enterprise JavaBeans (EJB) components, as well as supporting the key new capability of J2EE 1.4 web services.

EJB components have been a core strength of the J2EE platform from the beginning. The NetBeans IDE now provides support for creating and using EJB components. Developers can create EJB session beans to contain their transactional business logic, and use message-driven beans to create event-driven applications. Developers can also use the powerful database support in the NetBeans IDE to map existing database tables to EJB entity beans, or to create new object models using EJB entity beans and map them to database tables.

The use of web services in the enterprise is expanding rapidly and forms the core of a Service-Oriented Architecture (SOA). A developer using the NetBeans IDE can easily create simple Java applications that are exposed as web services for others to use, without knowing all the details of web services protocols, Web Services Description Language (WSDL), etc. Similarly, given a WSDL description of a web service developed by someone else, a developer using the NetBeans IDE can easily make use of that web service in his or her J2EE application. Web services are incredibly important to enterprise applications, and the NetBeans IDE makes web services easy!

Deployment descriptors are key to enabling portable enterprise

applications, but they can get in the way when developing simple applications. The NetBeans IDE removes the burden of dealing with deployment descriptors by completely managing them for the developer. Developers never need to think about deployment descriptors when developing and deploying J2EE applications. The NetBeans IDE will automatically and transparently create and manage the deployment descriptors that are needed for the J2EE application being developed.

The NetBeans IDE, when used with the J2EE SDK, provides a complete environment for creating, packaging, deploying, and debugging J2EE applications. The NetBeans IDE uses the J2EE application server from the J2EE SDK, and completely manages the application server for the developer. With a single click the NetBeans IDE will start the application server, deploy the application, and run the application in a mode ready for debugging!

The complexity of managing deployment descriptors is one of the problems recognized by J2EE 5.0, whose major goal is to significantly simplify development of J2EE applications. Version 4.1 of the NetBeans IDE delivers much of that simplification today. Future versions of the NetBeans IDE will further simplify J2EE application development, taking advantage of the improvements in J2EE 5.0.

The success of the J2EE platform is greatly enhanced by tools such as the NetBeans IDE, and J2EE developers will find that they're even more successful with the NetBeans IDE. We look forward to working with the NetBeans IDE team to provide great tools support to deliver on the promise of J2EE 5.0!

Bill Shannon

May 2005

Preface

Welcome to the second edition of the *NetBeans™ IDE Field Guide!*

This book is designed both as an introduction to NetBeans IDE and as a task reference, something that you can pick up from time to time to get an answer to a question or to find new ways to take advantage of the IDE's possibilities. Whether NetBeans is your first IDE or your fifth, this book can help you make the most of the IDE.

This edition is updated and expanded to cover the following features, some of which are brand new in NetBeans IDE 5.0:

- Matisse GUI builder
- The IDE's new and improved CVS support
- New editing and refactoring features
- Support for developing on the JSF and Struts web frameworks
- Support for creating plug-in modules for NetBeans IDE and standalone applications based on the NetBeans platform
- Support for profiling applications for performance problems

What is NetBeans IDE?

NetBeans IDE is a free-of-charge integrated development environment (IDE) primarily focused on making it easier to

develop Java applications. It provides support for all types of Java applications, from rich desktop clients to multi-tier enterprise applications to applications for Java-enabled handheld devices.

NetBeans IDE has a modular architecture that allows for plug-ins. However, the range of features in the basic installation is so rich that you can probably start using the IDE for your work without worrying about plug-ins at all.

The IDE itself is written in Java, so you can run it on any operating system for which there is a Java 2 Standard Edition JDK (version 1.4.2, version 5.0, or later) available. Click-through installers are available for Microsoft Windows, Solaris, Linux, Mac OS, and other systems. You can also download the IDE as a ZIP or TAR file if you want to install it on an operating system other than the ones listed here.

The IDE's basic job is to make the edit-compile-debug cycle much smoother by integrating the tools for these activities. For example, the IDE:

- Identifies coding errors almost immediately and marks them in the Source Editor.
- Helps you code faster with code completion, code template, word matching, and fix import features.
- Provides visual navigation aids, such as the Navigator window and "code folding," as well as numerous keyboard navigation shortcuts designed especially for Java programmers.
- Can display documentation for a class as you are typing in the Source Editor.

- Hot-links compilation errors in the Output window, so you can jump straight to the source by double-clicking the line or pressing F12.
- Manages package names and references to other classes. When you rename or move classes around, the IDE identifies places in the code that are affected by these changes and enables you to have the IDE generate the appropriate changes to those files.
- Has many debugging features that provide a comprehensive view of the way your code is working as it runs. You can set breakpoints (which persist from session to session) and keep your code free of clutter (such as `println` statements).
- Helps you integrate other parts of your workflow, such as checking sources in to and out from a version control system.

You can also download the NetBeans Profiler to augment the traditional edit-compile-debug cycle with performance profiling.

What Makes NetBeans IDE Special

When you use NetBeans IDE, you get the benefits of a top-shelf IDE without the negatives that you might associate with moving your development to a single environment.

Like other integrated development environments, NetBeans IDE provides a graphical user interface for command-line tools that handle the compiling, debugging, and packaging of applications.

Unlike other IDEs, NetBeans IDE does not force a build structure on you with project metadata that you need to reverse engineer if you are to build the project outside of the

IDE. NetBeans IDE builds on existing open standards to help you automate your development process without locking you in.

NetBeans IDE (beginning with version 4.0) bases its whole project system on Ant, which is the de facto standard build tool for Java applications. The project metadata that NetBeans IDE produces is in the form of XML and properties files that can be used by Ant outside of the IDE. Thus, developing a project in NetBeans IDE does not lock you or co-developers into NetBeans IDE.

You can use NetBeans IDE to create large projects with sophisticated build parameters. Where you already have such projects in place, you can adapt NetBeans IDE to work with them without necessarily changing the project's structure. If you are generally more comfortable with command-line tools because of their transparency and the level of control they allow you over your projects, NetBeans IDE could become the first IDE that you love.

NetBeans IDE is also consistently ahead of the curve in providing support for new and evolving standards, such as the new language features that were introduced in the Java SE 5 JDK and new specifications in all areas of Java technology.

NetBeans IDE provides an astonishing array of features right out of the box. NetBeans has a fully featured Java EE platform development environment built in. All the editor, debugger, and project support that is available for Java application development is also available for Java EE development. In addition, NetBeans IDE provides access to the Java BluePrints Solutions Catalog and the ability to install them as NetBeans projects.

The Mobility Pack, available as a free add-on installer, enables Java ME developers to design, develop, and debug MIDlets from within NetBeans IDE. Providing one of the most powerful sets of mobile development tools, the Mobility Pack includes a flow

designer to visually lay out the application logic, a screen designer to create the user interface, an integrated device fragmentation solution, and tools for building client server applications.

The NetBeans Profiler, also available as a free add-on installer, makes application performance profiling accessible to developers.

What Comes with NetBeans IDE

Besides providing support for coding, NetBeans IDE comes bundled with other tools and libraries that you might already use in your production environment. The IDE integrates these tools into the IDE workflow, but you can also use them at the command line.

Out of the box with NetBeans IDE 5.0, you get:

- Apache Ant 1.6.5
- Tomcat 5.5.9
- JUnit 3.8.1
- Java BluePrints Solutions Catalog

If you download the Mobility Modules pack, you also get the Wireless Toolkit.

You can also get NetBeans IDE in a bundle with the Java SE JDK or the Sun Java System Application Server Platform Edition.

If you download the NetBeans Profiler, you also get a full-featured, nonintrusive Java profiler that is based on profiling

technology developed at Sun Labs.

What This Book Will Do for You

This book was written with both new and existing NetBeans IDE users in mind.

If you are new to NetBeans IDE (or IDEs in general), this book will quickly guide you through the basics and advantages of using NetBeans IDE. Learn how to take advantage of the IDE's layout and feature integration to tighten up the basic edit-compile-debug cycle. Learn how to take advantage of the IDE's support for increasingly popular advanced technologies such as web services and Java EE technology to add new capabilities to your applications.

If you are already familiar with NetBeans IDE, this book will provide a new perspective on what you already know and possibly point you to useful features that you have not yet discovered. Learn how you can customize the IDE to work with complex build structures. If you are looking to move from client-server web applications to multi-tier transactional enterprise applications, this book will help you make that jump.

This book does not teach the Java programming language. Much of the material in this book is meaningful only if you have some experience with programming Java applications. However, this book could be a useful companion if you are expanding your Java technology palette into Java EE technology and other advanced areas.

How to Use This Book

NetBeans IDE is overflowing with features, so quite a bit can be written about it. *NetBeans™ IDE Field Guide* sorts out the

essentials so that you can get productive quickly and then adds a generous selection of tips and advanced information.

This book is primarily designed as a task-reference with short topics on accomplishing specific tasks. If you wish, you can read the book from cover to cover, but most likely you will want to keep it near your computer to get answers to pressing questions or simply to read up on ways to get more out of your work with the IDE. The topics are written in a way that allows you to skip all over the book to get answers to the specific questions you have without having to follow long end-to-end examples.

[Chapter 1](#) provides the information you need to install NetBeans IDE and to set up your first project.

[Chapter 2](#) provides an overview of the IDE environment and the basic tasks for developing general Java projects. If you have never used NetBeans IDE, you will probably want to read this chapter from beginning to end.

[Chapter 3](#) provides in-depth information on setting up and configuring projects. Although this chapter is mostly geared toward general Java applications, a working knowledge of the information in this chapter will be useful for developing Java EE and Java ME applications as well.

[Chapter 4](#) covers the IDE's unique and powerful client for the CVS version control system.

[Chapter 5](#) provides useful tips and tricks for making your day-to-day coding more productive, such as using editor hints, code completion, code templates, and refactoring features.

[Chapter 6](#) shows you how to use the IDE's powerful Project Matisse GUI builder to easily create visual desktop applications.

[Chapter 7](#) provides an overview of the IDE's rich set of

debugging features and displays practical techniques for finding problem areas in your code.

[Chapter 8](#) covers development of web applications, with a focus on developing with the Tomcat web server.

[Chapter 9](#) provides an overview of the IDE's support for the JavaServer Faces (JSF) and Struts web application frameworks.

There are several chapters devoted to Java EE topics. You should begin with [Chapter 10](#), Introduction to Java EE Development in NetBeans IDE, to get information on setting up your environment and learning how to leverage the Java BluePrints Solutions catalog in Java EE development.

If you are familiar with web application development and would like to learn how to extend it into using Java EE Enterprise JavaBeans components, you should read [Chapter 11](#), Extending Web Applications with Business Logic: Introducing EJB Components.

If you are interested in learning how to extend your Java EE applications to include web services, you should read [Chapter 12](#), Extending Java EE Applications with Web Services.

[Chapter 13](#), Developing Full-Scale Java EE Applications, contains in-depth information on developing entity beans, assembling applications, verifying Java EE compliance, and other topics.

[Chapter 14](#) covers special IDE features for using the NetBeans Mobility Pack to develop Java ME applications for handheld devices.

[Chapter 15](#) walks you through use of the NetBeans Profiler to detect memory leaks and find code that is slowing down your application.

[Chapter 16](#) provides information for taking advantage of

NetBeans IDE's unique Ant integration to use the IDE with existing intricate build environments.

[Chapter 17](#) shows you the process of creating a plug-in module for NetBeans IDE and also provides information on using the NetBeans Platform as a framework for creating rich client applications.

[Chapter 18](#) shows you how to use the IDE's unique Developer Collaboration modules to chat and share code with other developers in real time.

The appendices show you how to get and use project importers to simplify migration of projects from the Eclipse and JBuilder environments into NetBeans IDE.

NetBeans as Platform and Open-Source Project

Besides being an IDE, NetBeans is also a 100% pure Java open-source platform. You can develop plug-in modules for NetBeans IDE or create an entirely different application built on top of a small core of the modules that make up the IDE. Because NetBeans is 100% pure Java, any platform that supports a Java Virtual Machine will run NetBeans IDE. Hence, any plug-in module or application that extends NetBeans and that is 100% pure Java will also execute on any platform for which there is a Java Virtual Machine.

[Chapter 17](#) of this book addresses developing modules for the IDE and using the NetBeans platform as an application framework.

About the Authors

Patrick Keegan is one of the lead technical writers for NetBeans IDE. He has been writing about the IDE since May 1999, when NetBeans was a small Czech company yet to be acquired by Sun Microsystems. He lives in Prague, Czech Republic.

Ludovic Champenois is a senior architect at Sun Microsystems, and has been with Sun and Java for the last ten years. He is currently the tech lead and architect for NetBeans Java EE support, working with the Application Server group (J2EE 1.4, Java EE 5, and GlassFish open source community) and the Tools organization to make sure that NetBeans IDE is actively responding to changes in the Java Platform (Java ME, Java SE, and Java EE 5). He is a civil engineer from L'Ecole des Mines, Saint-Etienne, France.

Gregory Crawley conceptualized and implemented the Mobility device fragmentation solution for NetBeans IDE 4.0. He continues to be an avid NetBeans IDE user and developer of J2ME games in association with Cotopia Wireless.

Charlie Hunt is a Java Performance Engineer at Sun Microsystems. He has been working with Java since 1997 and has held many positions at Sun Microsystems, including Java Architect, NetBeans Technology Evangelist, and Java Performance Engineer.

Christopher Webster, a member of the NetBeans Enterprise Pack development team, focuses on service-oriented architecture (SOA) development tools. Before joining Sun, Chris was a computer scientist at the Lawrence Livermore National Laboratory. Chris holds a B.S. in computer science from the University of Hawaii and an M.S. in computer science from Baylor University.

John Jullion-Ceccarelli has been writing about NetBeans IDE since he joined Sun Microsystems in 2001. John is the original author of much of the documentation that appears on www.netbeans.org. Currently, he is one of the lead technical writers on the NetBeans project.

Jiri Prazak is an engineer on the NetBeans Mobility Pack team. Jiri has been working for Sun in various capacities for five years. If he's not writing code or contributing to books, there's a good chance you will find him scaling enormous rocks in various mountain ranges around the world.

Martin Ryzl is the engineering manager for NetBeans Mobility Pack. He joined NetBeans in March 1999 when it still a small Czech startup company. Besides other projects, he has been involved in Java ME support since 1999, helped to integrate the first version with the award-wining J2ME Wireless Toolkit, and was later on responsible for Sun Java Studio Mobility and NetBeans Mobility Pack.

Gregg Sporar has been a software developer for over twenty years, working on projects ranging from control software for a burglar alarm to 3D graphical user interfaces. He has been using Java since 1998, and his interests include user interfaces, development tools, and performance profiling. He works for Sun Microsystems as a Technology Evangelist on the NetBeans project.

Geertjan Wielenga is the technical writer responsible for NetBeans documentation relating to plug-in modules and rich-client applications. He has been a technical writer since 1996, focusing mainly on software documentation, with a special emphasis on IDEs. Geertjan holds an LI.B degree from the University of Natal in Pietermaritzburg, South Africa. He is a very active blogger (<http://blogs.sun.com/geertjan>).

Acknowledgments for the Second Edition

The second edition of this book would not have been possible without the persistence of Charlie Hunt, Judith Lilienfeld, and Greg Doench in getting the second edition off the ground, and the rapid response and coordination of all the authors in pulling together the material. In addition, Brian Leonard and Ken Ganfield deserve awards for the very helpful review that they provided on very short notice. A big thanks to all.

Patrick Keegan

Acknowledgments for the First Edition

Patrick Keegan

First of all, I'd like to thank everybody who made my participation in this book possible, especially David Lindt, my manager, and John Jullion-Ceccarelli, who assumed many of my lead duties and still managed to make enormous content contributions to the NetBeans IDE 4.1 release. The book itself would not have been possible without the support of Tim Cramer and efforts of Larry Baron, who pulled together the resources to make it happen. Thank you to everybody who so enthusiastically supported the book with a combination of small contributions, reviews, suggestions, and moral support. In particular, I would like to cite Vincent Brabant and David Coldrick.

Other people whose insight and sharp eyes greatly added to the quality of the book: Gregg Sporar, Geertjan Wielenga, Marian Petras, Maros Sandor, Karel Zikmund, Lubomir Cincura, David Konecny, Roman Strobl, Milan Kubec, Jiri Prazak, and Adam Sotona. On the editorial and production side, thanks to Greg Doench, Tyrrell Albaugh, and Kathy Simpson for their guidance and flexibility. Last and possibly most, I'd like to thank Tim Boudreau, who goaded me into taking on this project.

Ludovic Champenois

First and foremost, my special thanks and love to my family Vannina, my wife, and Elio, Flora, Lucas, Bianca, my four children for supporting me during the last year. Bianca still believes Java is an island like Corsica, and she is right! Vannina reviewed parts of the book, and while having no knowledge at all of the domain, she gave me precious feedback: Keep it

simple stupid! I love you. Love to Gaspard and Stan: *Je pense à vous si souvent.*

This book is about NetBeans, and NetBeans is a fantastic communitythe people in Prague, the people in the U.S., many people from Sun Microsystems... Jesse Glick, who trained me on the internals of NetBeans, back in 1999, and Jeet Kaul and Jeff Jackson, who helped me tremendously in my quest to move J2EE support from Sun Studio products to the open source world. My thanks to the engineers from StudioNam Nnguyen, Rico Cruz, and Chris Websterwho changed their focus and invested all their time on the success of NetBeans IDE 4.1. Along with the engineers from the Sun Application Server organizationVince Kraemer, Peter Williams, Nitya Doraisamy, Rajeshwar Patil, and Anil Gaurthey did most of the foundation work. Thanks, this was incredible cross-organizational teamwork! The foundation is finally there; let's have some rest before J2EE 5.0.

Gregory Crawley

Thanks to Martin Ryzl and the entire mobility team for their support throughout this entire process.

Charlie Hunt

I thank Tim Cramer, director of NetBeans, for suggesting I contribute to this book. I have been blessed to work with a great team of coauthors who made the book writing an enjoyable experience: Patrick Keegan, Ludovic Champenois, Chris Webster, and Greg Crawley. I wish to extend a special thanks to Patrick Keegan for leading the writing effort. I also want to thank Bill Shannon for writing a foreword. Finally, I want to thank my wife, Barb Hunt, for her encouragement to contribute to the book and for putting up with me while I wrote.

Christopher Webster

I would like to thank Ludo for his leadership skills. I would also like to acknowledge the hard work from the Java Studio Enterprise team, the Application Server team, and the NetBeans J2EE team. Finally, I would also like to thank the early adopters of NetBeans IDE 4.1, whose feedback has been invaluable.

Chapter 1. Download, Installation, and First Project

- [Downloading the IDE](#)
- [Installing the IDE](#)
- [Setting a Proxy](#)
- [First NetBeans IDE Project](#)

THIS CHAPTER PROVIDES THE BASIC INFORMATION that you need to get NetBeans IDE running on your system and then runs you through creation of a very simple project to get you started. Additional basic information follows in [Chapter 2](#), NetBeans IDE Fundamentals.

Downloading the IDE

You can download NetBeans IDE from the netbeans.org web site or the java.sun.com site. Visit <http://www.netbeans.org/downloads/index.html> for a list of and links to all downloads.

NetBeans IDE 5.0 is available in the following distributions:

- The basic IDE distribution. This distribution includes support for developing general Java libraries, visual desktop applications, web applications, and enterprise tier applications. In addition, it includes a rich set of features for developing plug-in modules for the IDE and for developing standalone rich client applications based on the NetBeans Platform.

You can get this distribution as an installer (for Microsoft Windows, Solaris, Linux, and Mac OS systems) or as an archive distribution (.zip file or .tar file).

Use this distribution if you already have the JDK installed on your system (must be version 1.4.2, 5.0, or compatible) and you do not need to download the Sun Java System Application Server (either because you already have it or because you do not need it for the applications you are developing).

- The basic IDE distribution bundled with the Sun Java System Application Server Platform Edition.

This is the most convenient download if you want to start developing and deploying applications to an application server (and you do not yet have the Sun Java System Application Server).

- The basic IDE distribution bundled with Java SE JDK 5.0.

This is a convenient download if you do not already have the JDK installed on your system. Both the JDK and the IDE are installed at the same time.



Having just the Java Runtime Environment (JRE) installed on your system is not sufficient for running NetBeans IDE. You need to have the JDK, which includes a copy of the JRE. The IDE relies on development tools provided by the JDK, such as the `javac` compiler, and takes advantage of other parts of that download, such as the JDK sources that it includes. You can go to <http://java.sun.com/j2se/index.jsp> to find and download the latest version of the JDK.

In addition, there are the following optional add-on installers for NetBeans IDE:

- NetBeans Mobility Pack. This installer adds support for developing applications based on Java ME technology for mobile devices.
- NetBeans Profiler. This pack adds the ability to profile your application's performance using dynamic bytecode instrumentation.

As the word "add-on" implies, these installers work only if you have a compatible version of NetBeans IDE already installed on your system.

Installing the IDE

Installing the IDE is simple and is basically composed of these steps:

- 1.** Make sure that you have a suitable JDK version installed on your system. For NetBeans IDE 5.0, the JDK version must be 1.4.2 or higher. If you do not have a suitable JDK version on your system, install that first or download a JDK/NetBeans IDE bundle.
- 2.** If you are running on the Solaris or Linux operating system, change the permissions on the installer file to make it executable (if necessary).
- 3.** Double-click the installer (or, on Solaris or Linux, launch the installer from the command line) and then step through the installer wizard.

That's pretty much all you have to do. The installer guides you through selecting a JDK on your system to run the IDE on and (optionally) creates desktop icons and Start menu items for the IDE.

See [Chapter 10](#), Introduction to Java EE Development in NetBeans IDE, for information on setting up the Sun Java System Application Server.

Setting a Proxy

It is useful for the IDE to have a web connection. The IDE periodically checks the web to see if new or updated modules are available. Also, some IDE functions such as Validate XML might rely on resources on the web. In addition, some Help menu items are links to documentation on the web.

If you work behind a firewall, you might need to configure the IDE to use a proxy for HTTP connections to the web. The IDE attempts to use your system's proxy. If that does not work, you can set the proxy manually.

To manually set a proxy in the IDE:

- 1.** Choose Tools | Options.
- 2.** On the left side of the Options dialog box, click General.
- 3.** In the Proxy section of the dialog box, select the HTTP Proxy radio button, and fill in values for the proxy host and port.

First NetBeans IDE Project

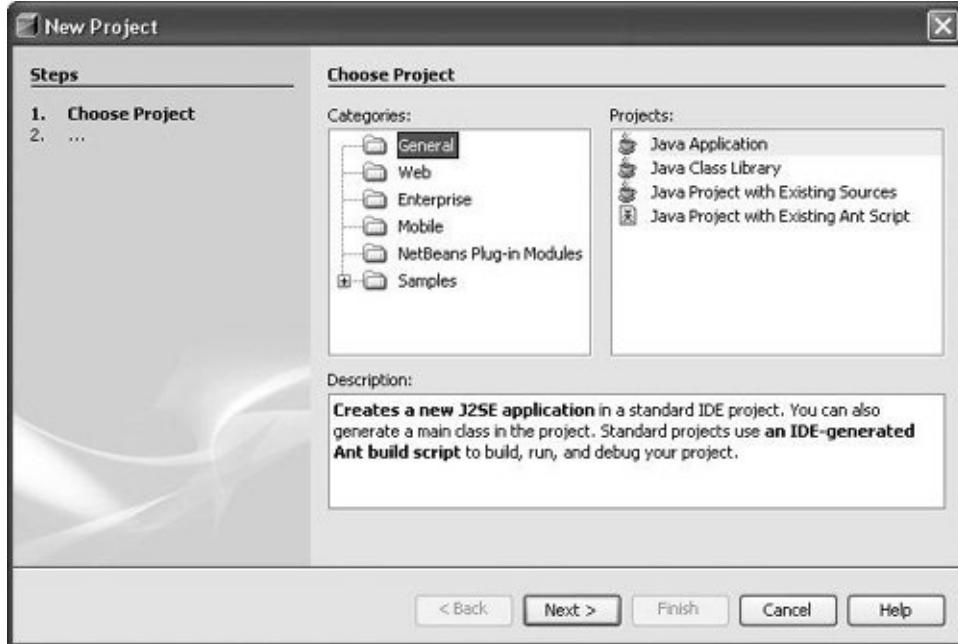
Once you have started the IDE, you are presented with a "welcome" window and some other empty windows. To help get you started, this section provides a quick run-through of setting up, compiling, and running a "Hello World" project.

To set up the project:

1. Choose File | New Project (or press Ctrl-Shift-N).
2. In the New Project wizard, expand the General node, select Java Application, and click Next (as shown in [Figure 1-1](#)).

Figure 1-1. New Project wizard, Choose Project page

[[View full size image](#)]

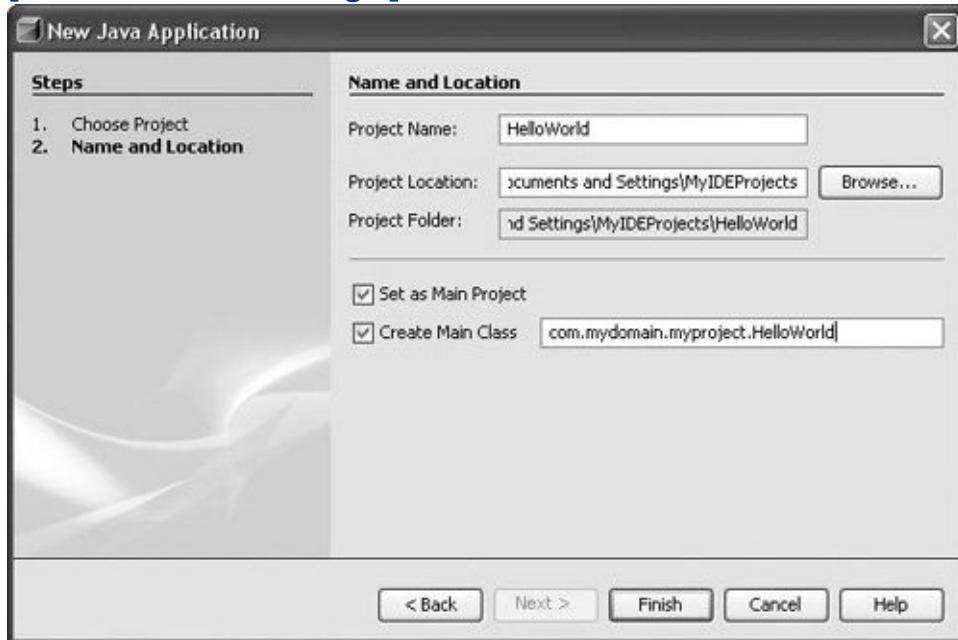


The Java Application template sets up a basic project and includes a main class.

3. In the Name and Location page of the wizard (as shown in [Figure 1-2](#)), type `HelloWorld` as the project name.

Figure 1-2. New Project wizard, Name and Location page

[[View full size image](#)]



4. In the Create Main Class field, change `helloworld.Main` to `com.mydomain.myproject.HelloWorld`. (When you enter a fully qualified class name in this field, the IDE generates directories for each level of the package structure.)
5. Click Finish.

Once you have finished the wizard, the IDE will run a scan of the classpath that has been set for the project to enable features such as code completion to work.

The following windows are then populated:

- The Projects window, which provides access to your sources, any tests you might have, and your classpath (represented through the Libraries and Test Libraries nodes). See [Figure 1-3](#).

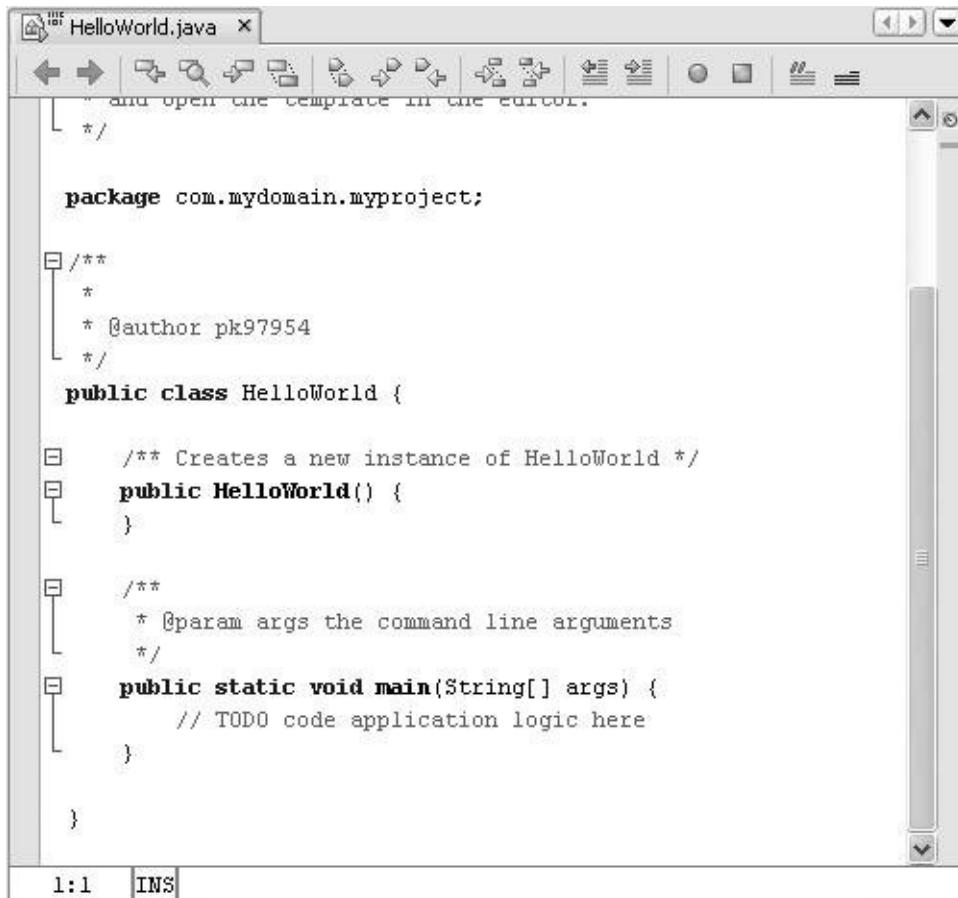
Figure 1-3. Projects window with nodes for the HelloWorld project



- The Navigator window, which provides an easy way for you to view and access members of the currently selected class. It also makes it easy to browse the inheritance tree of a class.
- The Source Editor, where a tab for the `HelloWorld` source file opens. See [Figure 1-4](#).

Figure 1-4. Source Editor with HelloWorld.java

open



The screenshot shows a Java code editor window titled "HelloWorld.java". The code is as follows:

```
/* and open the template in the editor.
 */
package com.mydomain.myproject;

/**
 * @author pk97954
 */
public class HelloWorld {

    /** Creates a new instance of HelloWorld */
    public HelloWorld() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}

1:1 INS
```

To modify, build, and run the project:

1. In the Source Editor, click within the `main` method at the end of the line that reads `// TODO code application logic here`.
2. Press the Enter key and then type the following line:

```
System.out.println("Hello World!");
```

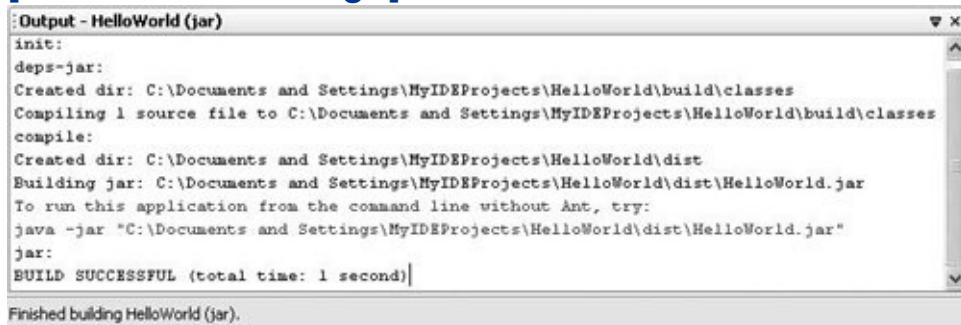
3. Press Ctrl-S to save the application.

4. Press F11 (or choose Build | Build Main Project) to compile and package the application. This command triggers an Ant script that the IDE has generated and will maintain for the project.

The Output window opens and displays the output from the Ant script as it runs through its targets. See [Figure 1-5](#).

Figure 1-5. Output window showing successful building of the HelloWorld project

[\[View full size image\]](#)



```
:Output - HelloWorld (jar)
init:
deps-jar:
Created dir: C:\Documents and Settings\MyIDEProjects\HelloWorld\build\classes
Compiling 1 source file to C:\Documents and Settings\MyIDEProjects\HelloWorld\build\classes
compile:
Created dir: C:\Documents and Settings\MyIDEProjects\HelloWorld\dist
Building jar: C:\Documents and Settings\MyIDEProjects\HelloWorld\dist\HelloWorld.jar
To run this application from the command line without Ant, try:
java -jar "C:\Documents and Settings\MyIDEProjects\HelloWorld\dist\HelloWorld.jar"
jar:
BUILD SUCCESSFUL (total time: 1 second)

Finished building HelloWorld (jar).
```

5. Press F6 (or choose Run | Run Main Project) to run the project.

The Output window should display a combination of Ant output and the "Hello World!" message from your application. See [Figure 1-6](#).

Figure 1-6. Output window showing the successful running of the HelloWorld project

```
:Output - HelloWorld (run)
init:
deps-jar:
compile:
run:
Hello World!
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
Finished building HelloWorld (run).
```

With that, you have created and run an application in the IDE. You can now move on to the next chapter to get a broader overview of the IDE, or you can skip ahead to [Chapter 3](#) to cut straight to the details of creating and customizing projects.

Trying Out NetBeans IDE with Sample Code

If you want to check out the features of NetBeans IDE on working code without touching your existing projects, or if you just want to see what a working project looks like in the IDE, you can open one of the sample projects that come with the IDE.

When you create a sample project, the sample code is copied into a directory of your choosing, and all necessary project metadata is generated.

To create a sample project:

1. Choose File | New Project.
2. In the New Project wizard, expand the Samples folder; choose a template from one of the categories; and click Next.

The General category contains visual desktop applications.

The Web category contains several examples designed to run on the Tomcat server.

The NetBeans Plug-in Modules category contains an example of how you can create a project to add features to the IDE itself.

The BluePrints Solutions category contains examples of useful design patterns for Java EE applications. These examples accompany the BluePrints Solutions Catalog, which is available from the Help menu and which provides documentation for these design patterns in a problem/solution format.

If you have the Mobility Pack installed, a Mobility category also appears and includes samples from the Java ME Wireless Toolkit.

3. On the Name and Location page of the wizard, check the generated values for the name and location of the project and change them, if you wish. Then click Finish.

Once you have created a new sample project, you can view its source code and project structure and then build and run that application within the IDE.

Chapter 2. NetBeans IDE Fundamentals

- [Creating a Project](#)
- [Configuring the Classpath](#)
- [Creating a Subproject](#)
- [Creating and Editing Files](#)
- [Setting Up and Modifying Java Packages](#)
- [Compiling and Building](#)
- [Viewing Project Metadata and Build Results](#)
- [Navigating to the Source of Compilation Errors](#)
- [Running](#)
- [Creating and Running Tests](#)
- [Debugging the Application](#)
- [Integrating Version Control Commands](#)
- [Managing IDE Windows](#)

THIS CHAPTER PROVIDES A GENERAL OVERVIEW of both the workflow in the IDE and the key parts of the IDE. Once you finish this chapter, you should have a solid understanding of the IDE's principles and be able to take advantage of the IDE's

central features.

If you are already familiar with NetBeans IDE (4.0 or higher), you can probably skim this chapter or skip it altogether. Subsequent chapters will revisit most of this material in greater depth to answer more involved questions and provide additional details that you can use to squeeze more productivity out of your work with the IDE.

Creating a Project

Before you can do any serious work in the IDE, you need to set up a project. The project essentially sets up a context for you to write, compile, test, and debug your application. This context includes the classpath; folders for your sources and tests; and a build script with targets for compiling the application, running tests, and building JAR files (or other types of distributable archive files).

You can choose among a variety of project template categories, which are grouped according to the technology you are basing your application on (for example, general Java, Java EE web tier, Java EE enterprise tier, and Java ME).

Within the template categories, there are templates for new applications and for setting up an IDE project for existing applications you are working on. The New Project wizard provides a description for each template.

The With Existing Sources templates in each category enable you to set up standard IDE projects around applications that you have been developing in a different environment.

The With Existing Ant Script templates in each category, unlike the With Existing Sources templates and other standard project templates, enable you to set up a project based entirely on any existing Ant script. This approach requires some manual configuration to get some IDE features (such as debugging) to work with the Ant script, but the payoff is that you can get the IDE to work with any project structure, even if it does not adhere to the conventions of a standard IDE project. See [Chapter 16](#) for information on creating a project with a With Existing Ant Script template.

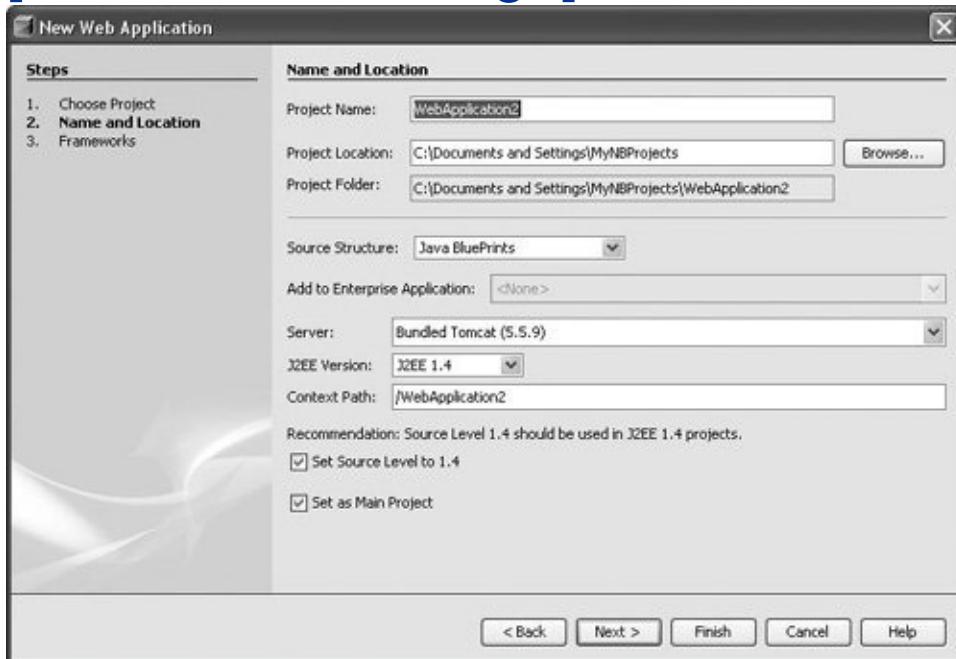
To set up a project:

1. Choose File | New Project.
2. In the wizard, select a template for your project, and complete the wizard.

The fields that you are asked to fill in depend on the template. Typically, you need to specify a location for the project (or, in the case of projects that use existing sources, where the sources are located). Web, Enterprise, and Mobility projects also include fields relevant for those specific types of applications. [Figure 2-1](#) shows the Name and Location page of the wizard for a new web application project.

Figure 2-1. New Project wizard, Web Application template, Name and Location page

[\[View full size image\]](#)



When you create a project, typically, the IDE does the following things for you:

- Creates a source tree with a skeleton class inside.
- Creates a folder for unit tests.
- Creates an Ant build script (`build.xml`), which contains the instructions that the IDE uses when you perform commands on your project, such as compiling source files, running the application, running tests, debugging, compiling Javadoc documentation, and building JAR files.

You can find more information on setting up standard projects in [Chapter 3](#). You can find more information on setting up free-form projects (those using the With Existing Ant Script template) in [Chapter 16](#).

Projects Window

The Projects window is essentially the command center for your work. It is organized as a tree view of nodes that represent parts of your project. It provides an entry point for your files and configuration options for the application you are developing.

In addition to displaying nodes for the files in the application that you are developing, the Projects window provides a representation of your classpath. The Libraries node for each project shows the version of the JDK you are developing against, as well as any other libraries you are basing your project on.

The Projects window presents your project in "logical" form that is, it represents the units of your application conceptually (rather than literally). For example, Java sources are grouped into packages without nodes for each level of file hierarchy. Files that you do not normally need to view, such as compiled Java classes and project metadata files, are hidden. This makes it easier to access the files you most regularly work with.

If you want to browse the physical structure of the project, including the project metadata, compiled classes, JAR files, and other files created in builds, open the Files window.

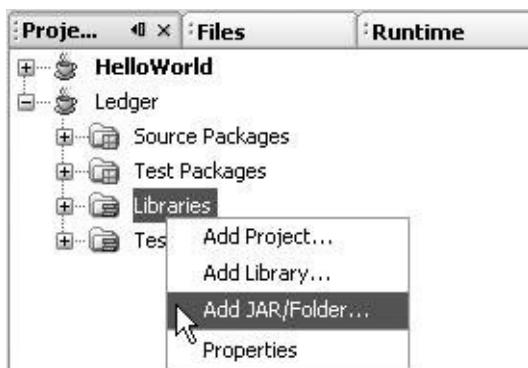
Configuring the Classpath

When you create a project, the IDE sets up a default classpath for you based on the project template you are using. If you have other things to add to the classpath, you can do so through the Libraries node of the project.

In fact, the IDE distinguishes among several types of classpaths, depending on project type, such as compilation classpath, test compilation classpath, running classpath, and test running classpath. The compilation classpath typically serves as a base for the other classpaths (for example, other classpaths inherit what is in the compilation classpath).

To add an item to the compilation classpath (and, thus, the other classpaths as well), right-click the project's Libraries node and choose Add JAR/Folder (see [Figure 2-2](#)).

Figure 2-2. Projects window, adding a JAR file to the classpath



When you right-click the Libraries node, you also can choose Add Project or Add Library. When you add a

project, you add the project's output (such as a JAR file) to the classpath.

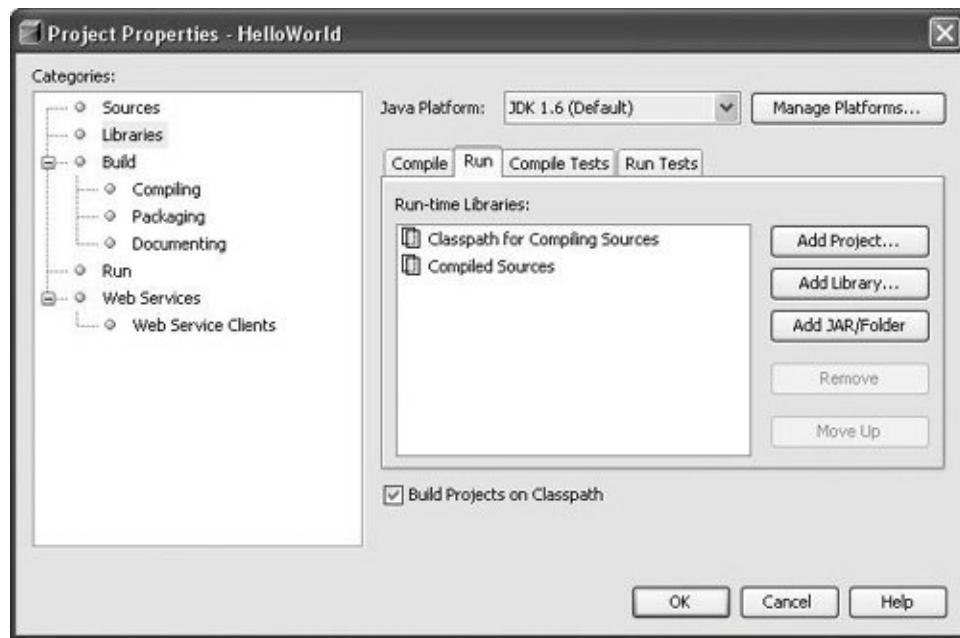
If you choose Add Library, you can add one of the "libraries" recognized by the IDE's Library Manager. In the Library Manager, libraries are essentially just a convenient grouping of one or more JAR files, sources, and/or Javadoc documentation. Designating libraries in the Library Manager is useful for several IDE features. For example, designating a JAR file and its sources as a library ensures that you can step through that JAR file's code when debugging.

You can manage existing libraries and designate new ones in the Library Manager, which you can open by choosing Tools | Library Manager.

You can edit other classpaths in the Properties dialog box for a project. To open the Project Properties dialog box, right-click the project's node in the Projects window and choose Properties. In the dialog box, click the Libraries node and use the customizer in the right panel to specify the different classpaths (see [Figure 2-3](#)).

Figure 2-3. Project Properties dialog box, Libraries page

[\[View full size image\]](#)



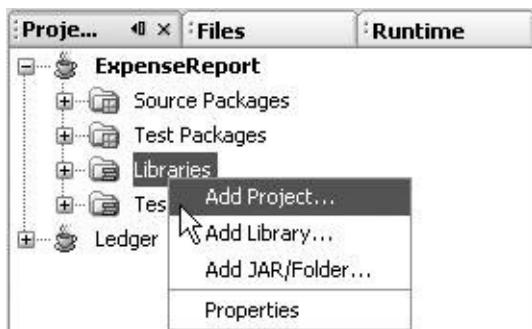
Creating a Subproject

Although there is no explicit distinction in the IDE between a project and a subproject, you can create a hierarchy of projects by specifying dependencies between projects. For example, you might create an umbrella Web Application project that relies on one or more Java Class Library projects. For larger applications, you might have several layers of project dependencies.

To set dependencies between projects:

1. Right-click the project's Libraries node and choose Add Project (as shown in [Figure 2-4](#)).

Figure 2-4. Projects window, making one project depend on another



2. In the file chooser that appears, navigate to the folder for the project you want to depend on and click Add Project. Project folders are designated with the icon.

Once you have established this dependency, the distributed outputs (such as JAR files) of the "added" project become part of the other project's classpath.



There is no visual project/subproject distinction in the

IDE, but there is a concept of "main" project. The main project in the IDE is simply the one that the IDE treats as the entry point for the primary commands such as Build Main Project and Run Main Project. The current main project is indicated with bold font in the Projects window.

There can be only one main project set at a time, although it is possible to have multiple projects open at the same time (including umbrella projects that serve as entry points for other applications you are developing).

You can make a project the main project by right-clicking its node in the Projects window and choosing Set Main Project.

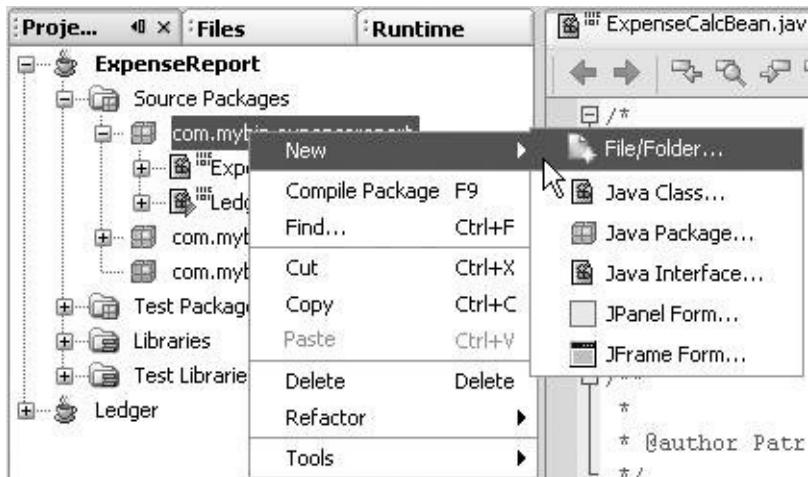
Creating and Editing Files

Once you have a project set up, you can add files to your project and start editing. You can add files to a project by creating them with the New File wizard.

To open the New File wizard, do one of the following:

- In the Projects window, right-click the Source Packages node (or one of the package nodes underneath it) and choose one of the templates from the New submenu. If none of the templates there suits you, choose File/Folder (as shown in [Figure 2-5](#)) to open a wizard with a complete selection of available templates.

Figure 2-5. Projects window, creating a new file



- Choose File | New File to open the New File wizard.

In the New File wizard, you can name the file and specify a folder. For Java classes, you can designate a period-delimited

package name (as opposed to a slash-delimited folder name).

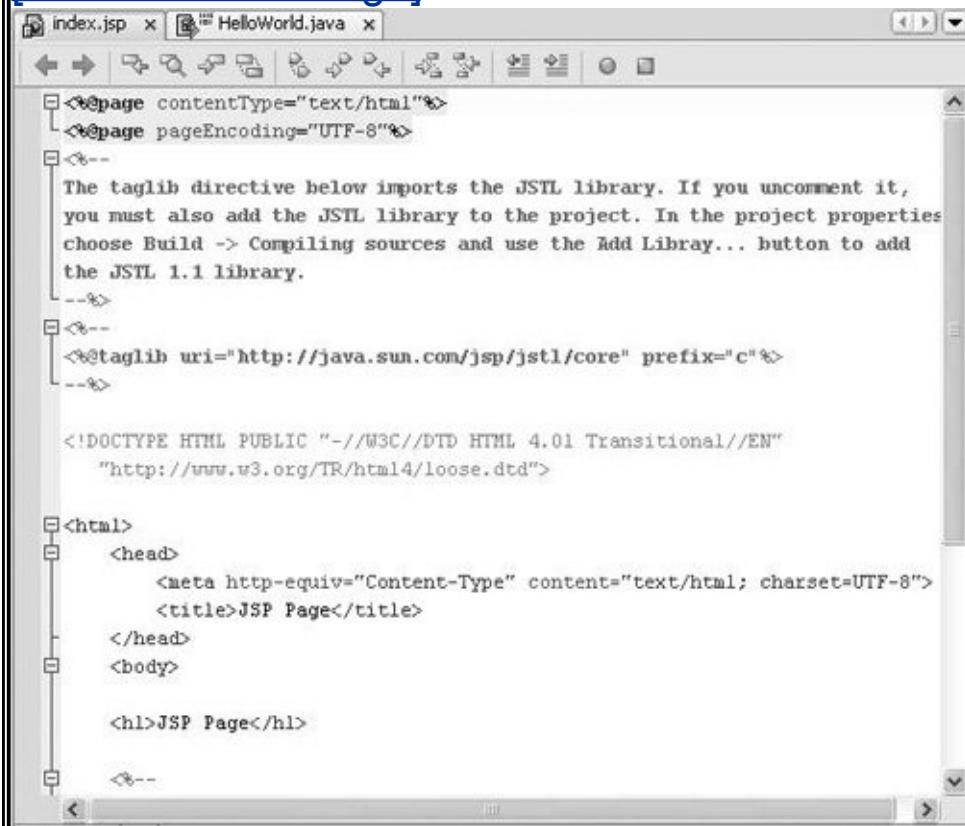
Once you complete the wizard, the file opens in a tab in the area of the IDE to the right of the Projects window. For most templates, a Source Editor tab opens.

About the Source Editor

The Source Editor is the central area of the IDE where you write and generate code. The Source Editor is actually a collection of different types of editors with different purposes. There are text editors for different types of files, such as Java, JSP (as shown in [Figure 2-6](#)), XML, HTML, and plain-text files. These editors all share a base of features (such as a set of common keyboard shortcuts). The individual editors have features unique to that file type, such as syntax highlighting, additional keyboard shortcuts, code completion, special navigation shortcuts, and so on. See [Chapter 5](#) for a survey of Source Editor features.

Figure 2-6. Source Editor window with a JSP file open

[[View full size image](#)]

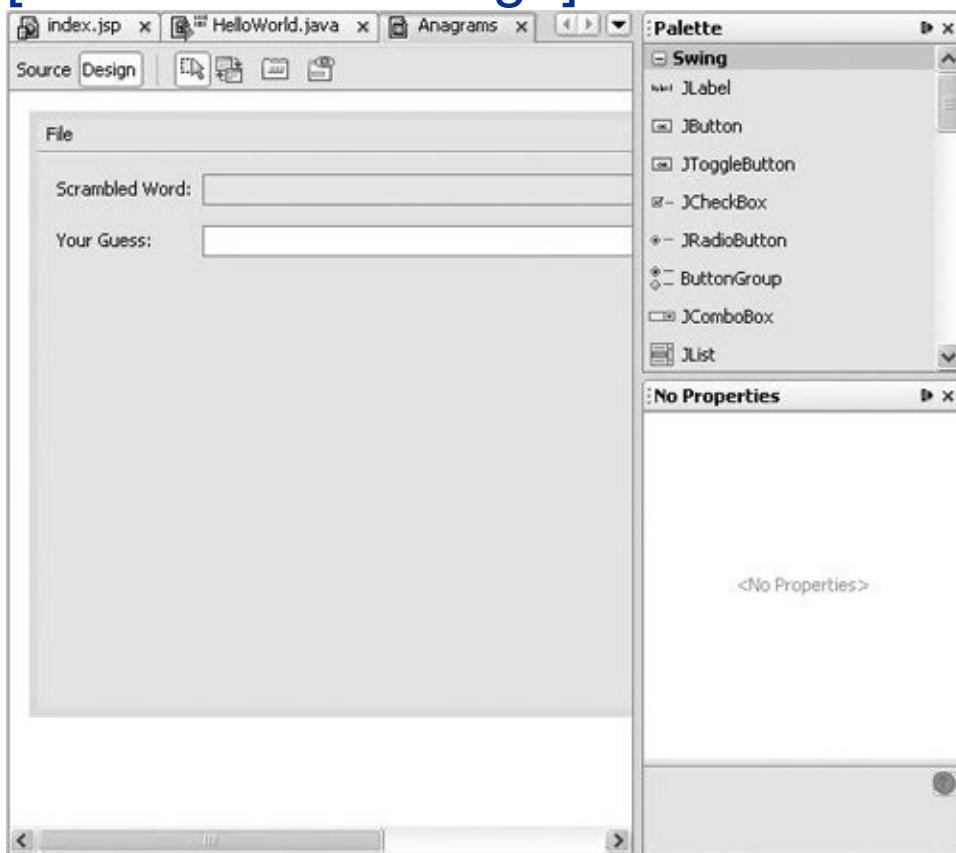


There are also visual editors for AWT and Swing forms, deployment descriptors, and other types of files, although it is possible to edit the source of these types of files directly.

For example, GUI templates such as JPanel Form and JFrame Form open in a visual design area (as shown in [Figure 2-7](#)) along with Palette, Inspector, and Properties windows. You can click the Source button in the design area's toolbar to access the file's source.

Figure 2-7. Form Editor Design view in the Source Editor window

[[View full size image](#)]



Setting Up and Modifying Java Packages

You can set up a Java package in the New Project and New File wizards.

To create a new package, right-click the Source Packages node within your project and choose New | Java Package. In the wizard, fill in a period-delimited package name (for example, `com.mybiz.myapp`).

You can then move classes into this package by cutting and pasting or by dragging their nodes.



When you move classes, the Move Class dialog box opens and offers to update the rest of the code in the project to reflect the changed location of the class. Click Next to see a preview of the changes in the Refactoring window. Then click Do Refactoring to make the changes.

Compiling and Building

When you set up a project, the IDE provides a default classpath and compilation settings, so the project should be ready to compile as soon as you have added some classes to the project.

You can compile an individual file or package by right-clicking its node and choosing Compile. But more typically, you will "build" the entire project. Building, depending on project type, typically consists of compiling projects and subprojects and creating outputs such as JAR files for each of those projects.

To build your project, right-click the project's node in the Projects window and choose Build Project. If that project is currently designated as the main project (the project name is in bold in the Projects window), you can choose Build | Build Main Project or press F11. If you want to delete the products of previous builds before building again, choose Build | Clean and Build Main Project or press Shift-F11.

When you initiate a build, the IDE tracks the progress of the build in the Output window in the form of Ant output.



You can specify compiler options in the Project Properties dialog box. Right-click the project's node in the Projects window and choose Properties. Then click the Build | Compiling node to enter the options.

See [Chapter 3](#) for more detailed information on building your project.

Viewing Project Metadata and Build Results

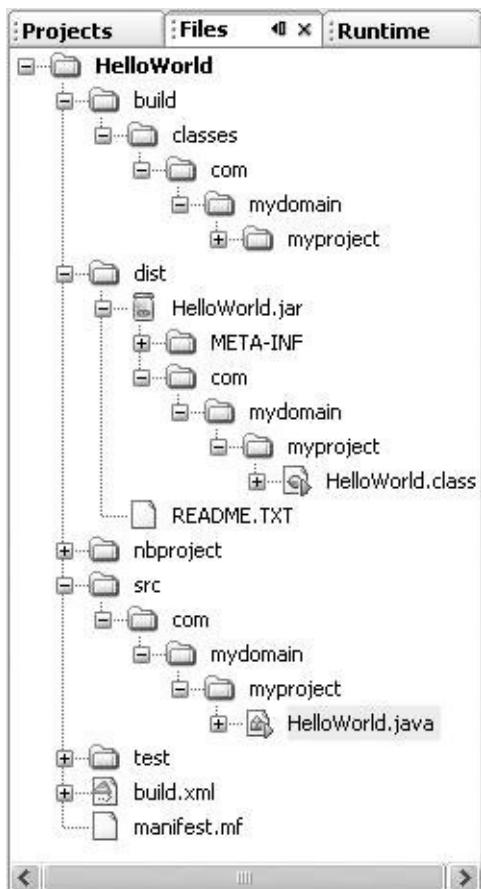
In the Files window, you can view the physical structure of your project, including compiled class files, output JAR files, your build script, and other project metadata.

Project-related commands (such as Build Project) are not available from nodes in the Files window, but other "explorer"-type commands like Open, Cut, and Paste are available.

The Files window is useful if you want to customize the build script for your project or you want to browse your project's outputs. You can also examine the contents of JAR files created by your project.

[Figure 2-8](#) shows the structure of the HelloWorld application you created in [Chapter 1](#).

Figure 2-8. Files-window "physical" view of the HelloWorld project



Navigating to the Source of Compilation Errors

If any compilation errors are reported when you compile or build, you can navigate straight to the source of the error by clicking the hyperlinked error in the Output window (as shown in [Figure 2-9](#)) or by pressing F12.

Figure 2-9. Output window with compiler error showing

[[View full size image](#)]



The screenshot shows a Windows-style window titled "Output - anagrams (jar)". The window contains the following text:

```
init:  
deps-jar:  
Compiling 1 source file to C:\Documents and Settings\Patrick Keegan\AnagramGame\build\classes  
C:\Documents and Settings\Patrick Keegan\AnagramGame\src\com\tutorialanagrams\ui\Anagrams.java:21: cannot find symbol  
symbol : variable guessedMor  
location: class com.tutorialanagrams.ui.Anagrams  
        guessedMor.requestFocusInWindow();  
1 error|  
BUILD FAILED (total time: 0 seconds)
```

If you have multiple errors, you can use F12 (Next Error) and Shift-F12 (Previous Error) to navigate between the locations of the errors.

Running

You can run the application you are developing from within the IDE by right-clicking the project's node and choosing Run Project or by pressing F6.

You can run an individual file by right-clicking the file in the Source Editor or the file's node in the Projects window and choosing Run File or pressing Shift-F6.

You can stop a running application by choosing Build | Stop Build/Run.

If you need to specify a main class for the project or you want to run the project with some arguments, you can specify these in the Project Properties dialog box. Right-click the project's node in the Projects window, choose Properties, and select the Run node in the Project Properties dialog box. You can then use the Main Class, Arguments, and VM Options fields.

Creating and Running Tests

IDE project templates are set up with unit testing in mind. Most project types set up a folder for unit tests next to the folder containing your sources. You can have the IDE generate skeleton code for a class's unit test and place it within the test folder with a package structure corresponding to that of the class to be tested.

To generate unit test code for a class:

1. In the Projects window, right-click the class you want to create a test for and choose Tools | Create JUnit Tests (Ctrl-Shift-U).
2. In the Create Tests dialog box, set a class name and location, and specify the code generation options for the test.

By default, the class name is filled in for you and corresponds to the name of the class being tested with `Test` appended to the name. The test class is placed in a test folder that has the same package structure as your sources.

If you later add some new methods to your class, you can choose the Create Tests command again. Test methods for the new methods will be added to the existing test class.

To run the selected project's tests, press Alt-F6 or choose Run | Test `"ProjectName"`.

To run a test for a specific file, select the file in the Source Editor or Projects window and press Ctrl-F6 or choose Run | Run File | Test `"Filename"`.

Debugging the Application

The IDE's debugger enables you to pause execution of your program at strategic points (*breakpoints*) and check the values of variables, the status of threads, and so on. Once you have paused execution at a breakpoint, you can step through code line by line.

To start debugging a program:

1. Make sure that the program you want to debug is currently set as the IDE's main project.

The name of the main project is shown in bold font in the Projects window. You can make a project the main project by right-clicking its node and choosing Set Main Project.

2. Determine the point in your code where you want to start debugging, and set a breakpoint at that line by clicking in the left margin of that line.

The  icon appears in the left margin to mark the breakpoint. In addition, the whole line is highlighted in pink.

3. Start the debugger by choosing Run | Debug Main Project or pressing F5.

The IDE builds (or rebuilds) the application and then opens the Debugger Console in the bottom-left portion of the IDE and the Watches, Call Stack, and Local Variables windows in the lower-right portion.

4. Click the Local Variables window (as shown in [Figure 2-10](#)) to view the values of any of the variables of the program that are currently in scope.

Figure 2-10. Debugger windows, with the Local Variables window in focus

Watches		Local Variables	Call Stack
Name	Type	Value	
+---◆ this	Anagrams	#[1231]	[...]
+---◆ evt	ActionEvent	#[1232]	[...]

See [Chapter 7](#) for a more in-depth look at the IDE's debugging features.

Integrating Version Control Commands

If you already use a version control system for your sources, you can integrate that system's commands into the IDE workflow.

In NetBeans IDE 5.0, support for working with CVS is built in. If you already have sources checked out from a CVS repository, that IDE recognizes this fact and provides you with a menu of CVS commands that you can use on the sources.

In addition, this CVS support is integrated with the IDE's project system. You can version IDE project metadata and easily check out IDE projects that other people have added to the repository.

See [Chapter 4](#) for information on getting the most out of the IDE's version control support.



NetBeans IDE 5.0 comes with built-in support only for CVS. "Generic" support for other version control systems is available from the IDE's Update Center, although this support is not as tightly coupled with the IDE's project system as the built-in CVS support. Built-in support for other version control systems is planned for future releases.

Managing IDE Windows

The IDE's window system is designed to provide a coherent and unobtrusive layout of the various windows you need while enabling you to adjust the layout effortlessly as you work. These are some of the things you can do as you work:

- Resize windows by clicking on a window border and dragging it to the width or height you prefer.
- Maximize a window within the IDE by double-clicking on its tab. (You can revert to the previous window layout by again double-clicking on the tab.) You might find this feature particularly useful in the Source Editor.
- Move a window to a different part of the IDE by clicking on its tab and dragging it to a different part of the IDE.
- Use drag and drop on a tab in the window to split the window.
- Make a window "sliding" by clicking its  button. When you click this button, the window is minimized, with a button representing that window placed on one of the edges of the IDE. You can mouse over the button to display the window temporarily, or you can click the button to open the window.

Chapter 3. IDE Project Fundamentals

- [Introduction to IDE Projects](#)
- [Choosing the Right Project Template](#)
- [Creating a Project from Scratch](#)
- [Importing a Project Developed in a Different Environment](#)
- [Navigating Your Projects](#)
- [Working with Files Not in the Project](#)
- [Creating Packages and Files in the Project](#)
- [Configuring the Project's Classpath](#)
- [Changing the Version of the JDK Your Project Is Based On](#)
- [Changing the Target JDK for a Standard Project](#)
- [Referencing JDK Documentation \(Javadoc\) from the Project](#)
- [Adding Folders and JAR Files to the Classpath](#)
- [Making External Sources and Javadoc Available in the IDE](#)
- [Structuring Your Projects](#)
- [Displaying and Hiding Projects](#)

- [Compiling a Project](#)
- [Running a Project in the IDE](#)
- [Deploying a Java Project Outside of the IDE](#)
- [Building a Project from Outside of the IDE](#)
- [Customizing the IDE-Generated Build Script](#)
- [Running a Specific Ant Target from the IDE](#)
- [Completing Ant Expressions](#)
- [Making a Menu Item or Shortcut for a Specific Ant Target](#)

NETBEANS IDE HAS A COMPREHENSIVE PROJECT SYSTEM that provides a structure for your sources, tests, and outputs and that simplifies your workflow. This project system is based on the Ant build tool, which provides added flexibility in the way you configure your projects.

This chapter provides an overview of the project system and how you can leverage it for your projects. In [Chapter 16](#), you can learn some more advanced techniques, which are particularly useful if you are developing applications with specific requirements that are not addressed in standard IDE projects.

This chapter focuses on general issues for general Java projects (with some information specific for web projects added in). However, most of the information is relevant for all project categories. If you need project-related information that is specific to Enterprise or Mobility projects, see the corresponding chapters in this book.

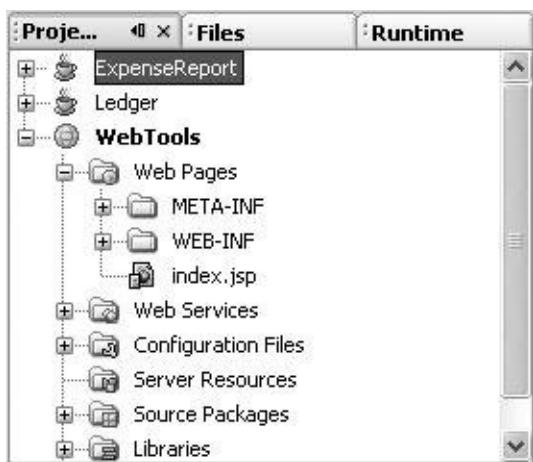
Introduction to IDE Projects

The starting point for most work in the IDE is through the creation of a project. When you create a project, the IDE typically does the following things for you (depending on project type):

- Creates a source tree with a skeleton class inside.
- Creates a folder for unit tests.
- Sets classpaths for compilation, running, and (depending on type of project) testing. (The compilation classpath also determines the classes that the Source Editor is aware of, for example, when you use code completion features.)
- Sets the Java platform on which the project will run. By default, it is the same platform that the IDE runs on.
- Creates an Ant build script (`build.xml`), which contains the instructions that the IDE uses when you perform commands on your project, such as compiling source files, running the application, running tests, debugging, compiling Javadoc documentation, and building JAR files. In addition, you can use this build script to run Ant targets on your project from outside of the IDE.

You can have multiple projects open at the same time, and the projects can be linked through dependencies. Project-specific commands in the Build and Run menus act on the currently designated main project. The main project is marked in bold, as shown in [Figure 3-1](#).

Figure 3-1. Projects window



What Is Ant, and Do I Need to Know Anything About It?

Ant is the tool that NetBeans IDE uses for running project-related commands. If you have no interest in Ant as such, you can completely ignore it, much as you would never bother decoding project metadata in another IDE. However, if Ant is already the lifeblood of your build process, you can set up NetBeans IDE to accommodate your existing build process, either by overriding specific Ant targets that the IDE generates or by providing your own Ant script.

Ant was developed by the Apache Software Foundation to automate routine developer tasks, such as compiling, testing, and packaging your application. Ant is similar to Make but has the advantage of being written in Java, so it works across multiple platforms. You can also use Ant to invoke other processes, such as checking out sources from version control, obfuscating classes, and so on. In addition, you can write Java classes to extend Ant's functionality. On big development efforts, Ant is often used as a production tool to compile and package the whole application for distribution.

Ant scripts themselves are written in XML. They are divided into high-level *targets*, which are collections of tasks that are run for specific purposes, such as cleaning the build directory, compiling classes, and creating packaged outputs.

Other IDEs provide integration with Ant to support writing and running of build scripts. NetBeans IDE takes this a step further by making Ant the backbone for all project-related commands in the IDE. When you create an IDE project, the IDE generates an Ant script for you with targets for, among other things, compiling, running, debugging, and packaging your application. When you run project commands in the IDE (such as Build Main Project or Run Main Project), the IDE itself is calling an Ant script.

The fact that the IDE's project system is based on Ant provides another advantage: Other developers do not have to use NetBeans IDE to build the project. It is possible to run an IDE-generated Ant script from another IDE or the command line, which could be particularly useful for doing production builds of your team's application.



Unlike in 3.x versions of NetBeans IDE, explicit creation of an IDE project is a primary step in your workflow in versions of the IDE beginning with 4.0. In NetBeans IDE 3.x versions, "projects" were a peripheral paradigm with a limited feature scope.

Also, as opposed to the filesystem concept in NetBeans

IDE 3.x, the folders included in the Projects and Files windows do not necessarily represent the classpath. In fact, it is OK to have multiple projects open, even if they have no relationship to the main project you are working with. Likewise, a project you are working on can depend on projects that you currently do not have open.

Choosing the Right Project Template

When you open the New Project wizard (File | New Project), you are presented with several templates, the use of which might not be immediately apparent.

Depending on the distribution of the IDE that you have, you might have several categories of templates. See [Table 3-1](#) for the list.

Table 3-1. Project Template Categories

Template Category	Description
General	For desktop applications or Java libraries based on Java 2 Standard Edition.
Web	For web applications based on the Java Enterprise Edition platform. These templates also include support for using the JavaServer Faces (JSF) and Struts web frameworks.
Enterprise	For enterprise tier applications, such as those that include Enterprise JavaBeans components (EJBs) and web services, based on Java 2 Enterprise Edition.
Mobile	For applications targeted toward handheld devices, based on Java 2 Micro Edition.
NetBeans Plug-in Modules	For creating modules that you can use to extend the IDE or to create a rich-client application based on the NetBeans Platform.
Samples	Sample applications that are ready to build and run from the IDE.

For each category, the IDE provides various templates based on the structure of the project and whether you already have sources and/or a fixed Ant script in place.

Standard project templates (all of the templates with the exception of With Existing Ant Script templates) provide maximum integration with the IDE's user interface. However, the use of those templates assumes that your project:

- Is designed to produce one distributable output (for example, a JAR or WAR file)
- Will use the Ant script that the IDE has generated for you (although you can customize this Ant script)

If either of those things is not true of your project, you can do one or more of the following:

- Create individual projects for each output and declare dependencies between the projects.
- Modify the generated Ant script to add or override targets.
- Use a With Existing Ant Script (or free-form) template (marked with a  icon to create your project).

The free-form templates offer you more flexibility in structuring your project. However, when you use a free-form template, you have to do some extra configuration and write build targets to get some IDE functionality (like debugging) to work. See [Chapter 16](#) for more information on free-form projects.

This chapter mainly covers general Java projects created from standard templates. There is also some information on web projects, though web projects are covered in more detail in

[Chapter 8](#). See [Chapter 11](#) for information on setting up EJB projects, [Chapter 12](#) for web service projects, [Chapter 13](#) for Java EE application projects, and [Chapter 14](#) for Java ME projects.

Creating a Project from Scratch

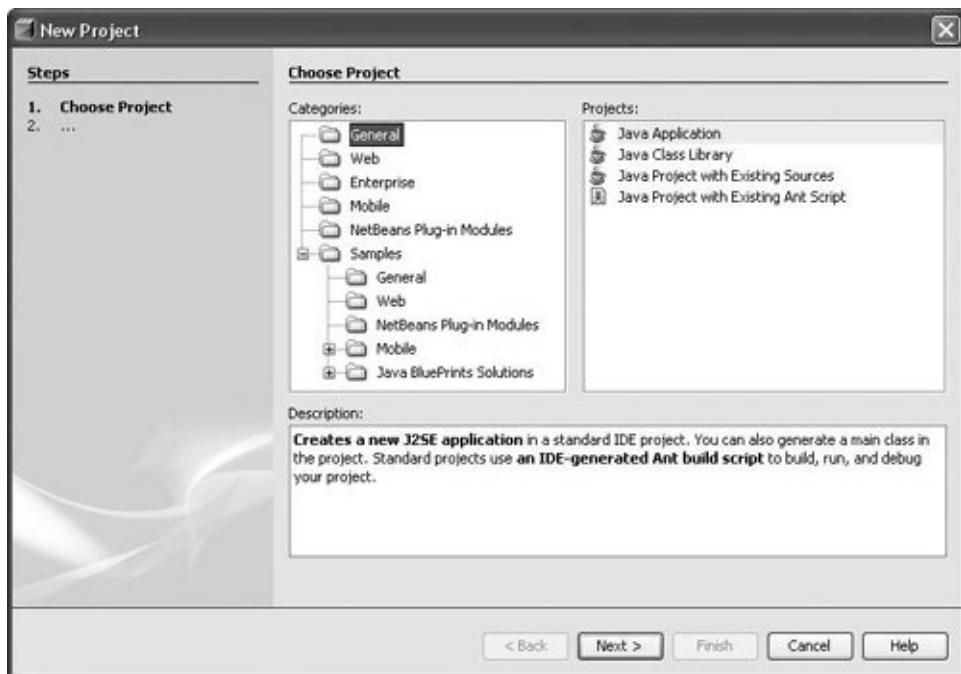
If you want to start developing an application from scratch, start with the base template for the type of application you are developing. Following are procedures for creating a general Java application and a web application.

For a Java desktop application or a standard Java library to be used by other applications, do the following:

- 1.** Choose File | New Project (Ctrl-Shift-N).
- 2.** In the Categories tree, select the General folder.
- 3.** Select Java Application (as shown in [Figure 3-2](#)) or Java Class Library.

Figure 3-2. New Project wizard, Choose Project page

[\[View full size image\]](#)



The Java Application template includes skeleton code for a main class. Use this template if you want this project to contain the entry point for your application.

The Java Library template contains no main class. Use this template if you want to create a library that is used by other applications.

4. Click Next.
5. Optionally, fill in the following fields on the Name and Location page of the wizard:

Project Name. The name by which the project is referred to in the IDE's user interface.

Project Location. The location of the project on your system.

The resulting Project Folder field shows the location of the

folder that contains folders for your sources, test classes, compiled classes, packaged library or application, and so on.

6. Optionally, deselect the Set As Main Project checkbox if you have another project open that you want the IDE's main project commands (such as Build Main Project) to apply to.
7. Set the main class, using the fully qualified class name but without the `.java` extension (for example, `com.mycompany.myapp.MyAppMain`).
8. Click Finish.

The generated project is viewable in both the Projects window and the Files window.

The Projects window provides a "logical" view of your sources and other files you are likely to edit often (such as web pages in web application projects). Project metadata (including the project's Ant build script) and project outputs (such as compiled classes and JAR files) are not displayed here.

Java source files are grouped by package instead of by folder hierarchy.

The Files window represents all the files that are in your project in folder hierarchies as they appear on your system.

For web applications, do the following:

1. Choose File | New Project (Ctrl-Shift-N).
2. In the Categories tree, select the Web folder.
3. Select Web Application and click Next.

- 4.** Fill in the following fields (or verify the generated values) in the Name and Location panel of the wizard (as shown in [Figure 3-3](#)):

Project Name. The name by which the project is referred to in the IDE's user interface.

Project Location. The location of the project on your system.

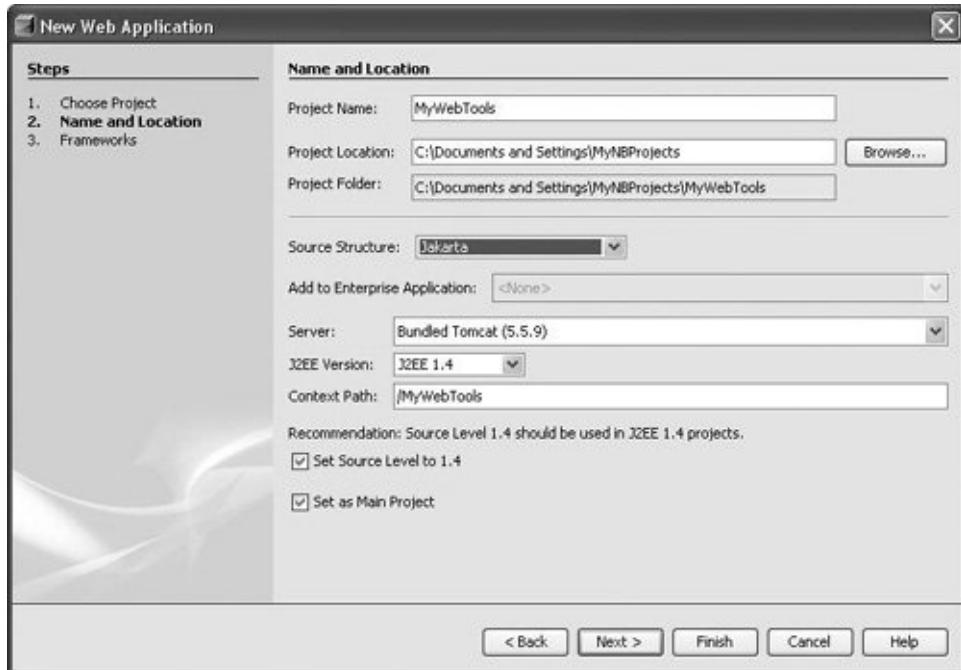
The resulting Project Folder field shows the location of the folder that contains folders for your sources, test classes, compiled classes, packaged library or application, and so on.

Source Structure. Sets some conventions that will be used for structuring your application. If you will be deploying to Tomcat, you should choose Jakarta. If you will be deploying to Sun Java System Application Server, choose Java BluePrints.

Server. The server you plan to deploy the application to. Only servers that are registered in the IDE's Server Manager are available in the combo box. You can add additional servers (or server instances) there by choosing Tools | Server Manager in the main menu.

Figure 3-3. New Project wizard, Name and Location page for Web Application project

[\[View full size image\]](#)



5. Optionally, deselect the Set As Main Project checkbox if you have another project open that you want the IDE's main project commands (such as Build Main Project) to apply to.

6. Optionally, adjust the following:

J2EE Version. The version of the Java EE platform your application will run against. In NetBeans IDE 5.0, versions 1.3 and 1.4 are available. Version 1.4 is preferable if you are starting from scratch, as it supports several constructs that are not recognized in version 1.3, such as tag files. However, you might need to use version 1.3 if you are setting up a project with existing sources already developed against that level.

In post-5.0 versions of the IDE, version 1.5 will be supported, which will provide features that will greatly simplify development of enterprise applications.

Context Path. The URL namespace the web application

uses. For example, if the context property value is `/MyWebApp`, the web application is accessed from a file within `http://HostName:PortNumber/MyWebApp/`.

- 7.** If you would like to base your application on the JavaServer Faces (JSF) or Struts web frameworks, click Next and select the framework to use.
- 8.** Click Finish.

For more information specific to developing web applications, see [Chapter 8](#).

If the application you are developing requires multiple outputs, you can create multiple IDE projects to accommodate them. You can connect projects by declaring dependencies in a project's Project Properties dialog box. See [Creating Subprojects](#) later in this chapter.

Importing a Project Developed in a Different Environment

If you have a project you have been working on in a different development environment, you can "import" the project into NetBeans IDE using a "With Existing Sources" project template.

When you import a project, you point the IDE to the folders for your sources and your tests (they are not copied), and create a folder that holds metadata for the project. Following are procedures for importing a general Java application and importing a web application. If you want to import a project developed in Eclipse or JBuilder, you can use special importing tools for those projects. See the appendices at the end of the book.

To import a Java application that was created in a different environment:

1. Choose File | New Project (Ctrl-Shift-N).
2. In the Categories tree, select the General folder.
3. Select Java Project with Existing Sources and click Next.
4. On the Name and Location page of the wizard, fill in the following fields (or verify the generated values):

Project Name. The name by which the project is referred to in the IDE's user interface.

Project Folder. The location of the project's metadata. By default, the IDE places this folder with your project sources, but you can designate a different location.

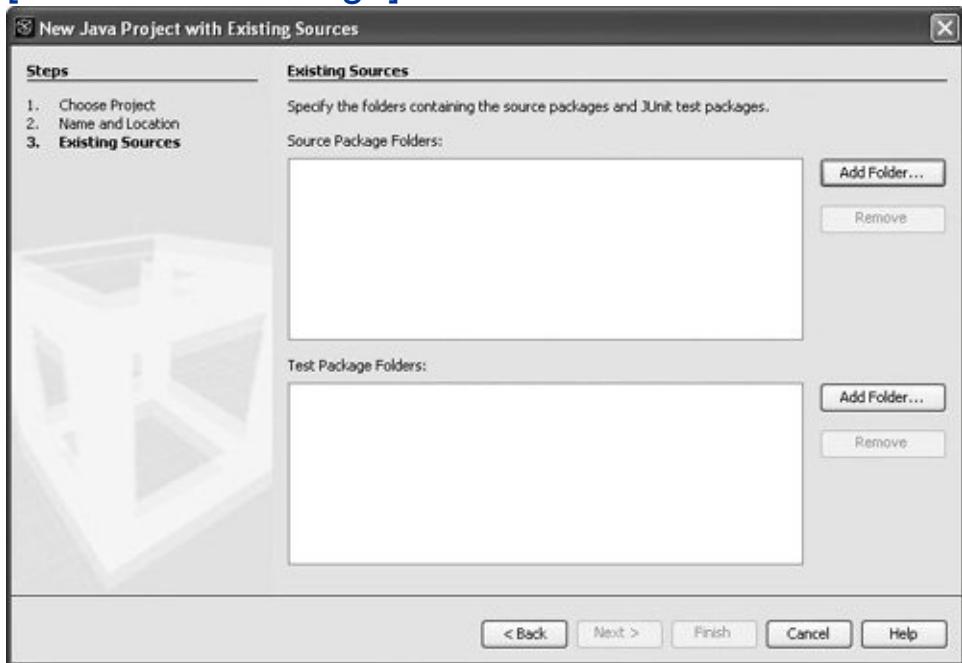
5. Optionally, deselect the Set As Main Project checkbox if you

have another project open that you want the IDE's main project commands (such as Build Main Project) to apply to.

6. On the Existing Sources page of the wizard (as shown in [Figure 3-4](#)), enter the location of your sources by clicking the Add Folder button next to the Source Package Folders field and navigating to the root of your sources. For example, the root folder might be called `src`.

Figure 3-4. New Project wizard, Existing Sources page for Java project with existing sources

[[View full size image](#)]



If you have multiple source root folders, repeat this step for each source root. Note, however, that sources added in this manner can be added to only one project (though this limitation might disappear after NetBeans IDE 5.0).

7. In the Test Packages Folder field, enter the folder that contains the default package of your unit tests (for example, the folder might be called `tests`).

If you have multiple test root folders, repeat this step for each source root. You can leave this field blank if you have no unit tests.

8. Click Finish.

To import a web application that you have begun developing in a different environment:

- 1.** Choose File | New Project (Ctrl-Shift-N).
- 2.** In the Categories tree, select the Web folder.
- 3.** Select Web Application with Existing Sources and click Next.
- 4.** On the Name and Location page of the wizard, fill in the following fields (or verify the generated values):

Location. The folder that contains your web pages and sources. If you have multiple source roots, you can fill those in later.

Project Name. The name by which the project is referred to in the IDE's user interface.

Project Folder. The location of the project's metadata.

Server. The server on which you plan to deploy the application. Only servers that are registered in the IDE's Server Manager are available in the combo box. You can add additional servers (or server instances) there by choosing Tools | Server Manager in the main menu.

- 5.** Optionally, deselect the Set As Main Project checkbox if you have another project open that you want the IDE's main project commands (such as Build Main Project) to apply to.
- 6.** Optionally, adjust the following:

J2EE Version. The version of the Java EE platform your application will run against. Version 1.4 is preferable if you are starting from scratch, as it supports several constructs that are not recognized in version 1.3, such as tag files. However, you might need to use version 1.3 if you are setting up a project with existing sources already developed against that level.

Context Path. The URL namespace the web application uses. For example, if the context property value is `/MyWebApp`, the web application is accessed from a file within `http://HostName:PortNumber/MyWebApp/`.

7. Click Next.
8. In the Existing Sources and Libraries page (as shown in [Figure 3-5](#)), fill in or verify the contents of the following fields:

Web Pages Folder. The folder that contains your web pages. This field should be filled in automatically, based on what you specified in the Location field of the wizard's Name and Location page.

Libraries Folder. The (optional) folder that contains any libraries specifically for this project, such as tag libraries.

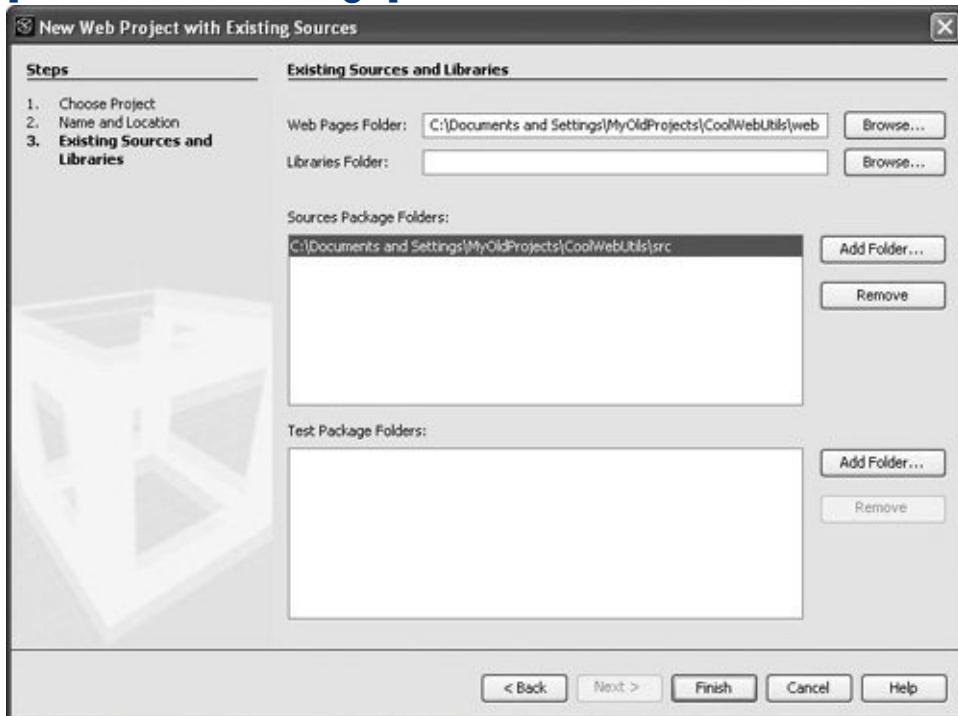
Source Package Folders. The folder (or folders) that contains your sources. This field should be filled in automatically, based on what you specified in the Location field of the wizard's Name and Location page and should contain the default Java package for the project (for example, the folder might be called `src`). You can click Add Folder to add additional source roots. However, you can add each source root to only one NetBeans IDE 5.0 project.

Test Package Folders. The folder (or folders) that contains any unit tests you have written for your sources.

You can click Add Folder to add additional test roots.

Figure 3-5. New Project wizard, Existing Sources and Libraries page for Web Application with Existing Sources template

[View full size image]



9. Click Finish.



For more complex projects that require a lot of classes, it might work best to use the web application project as the main project and configure it to depend on Java Library projects that handle most of the processing in your application. For applets and tag libraries, it is necessary to use Java Library projects to get them to work within your web application. See [Chapter 8](#).

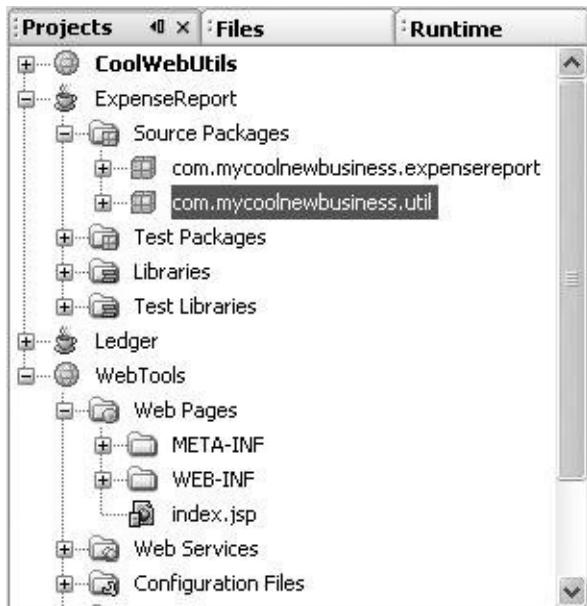
Navigating Your Projects

Once the project is set up, your files are accessible from both the Projects window and the Files window.

Projects Window

The Projects window (shown in [Figure 3-6](#)) is designed to be the center of operations for file creation, project configuration, and project building. The Projects window displays only the files that are likely to be regularly edited in a project, such as Java source files, web pages, and tests. Build outputs and project metadata are ignored.

Figure 3-6. Projects window



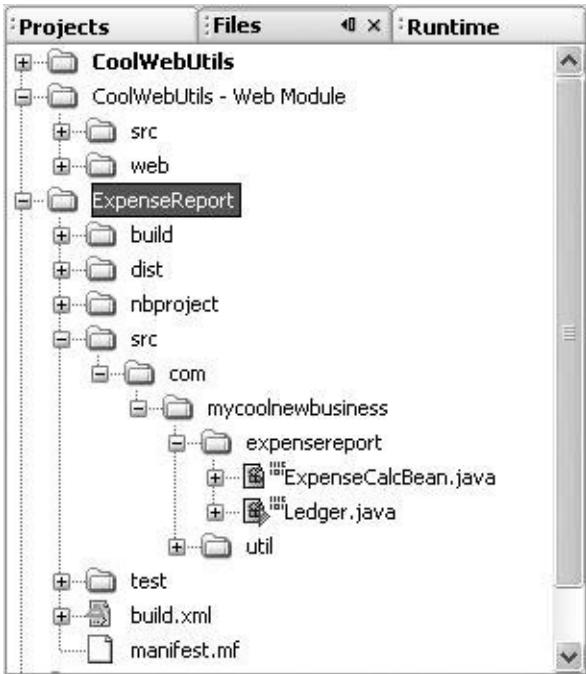
Java sources are displayed according to package structure, which generally makes it easier to navigate to files because you do not have to navigate through nested folder hierarchies.

The main node for each project has commands on its contextual (right-click) menu for compiling, running, debugging, creating Javadoc, and performing other project-related actions.

Files Window

The Files window displays your files organized according to file hierarchy, much as they appear in a file manager on your system. (One exception to this is that the `.form` files that are used to generate the design view of GUI classes that you create in the IDE's Form Editor are not displayed; just the `.java` file is displayed.) In addition to your sources, tests, and web files, project metadata and files produced when you build the project appear. See [Figure 3-7](#).

Figure 3-7. Files window



The Files window is useful if you need to browse the physical structure of your project, such as the contents of the JAR file (or other archive file) produced by the project. The Files window also provides direct access to project metadata files if you need to customize them. Otherwise, it is probably best to work in the Projects window, because project-related commands are not available in the Files window.



If you prefer a physical view of your files but do not want to see the top levels of the file hierarchy that are necessitated by the Java package structure, you can use the Favorites window. See [Working with Files Not in the Project](#) later in this chapter.

You can also set the Projects window to display files according to file hierarchy instead of by package. Choose Tools | Options and click General in the left side of the window. Then select the View Java Packages as Tree radio button.

Fast Navigation Between the Projects and Files Windows

If you are working with a node in one window but want to switch to that node in a different window, there are several keyboard shortcuts available to speed navigation. For example, if you have been working in the Projects window and now want to inspect the build script (which, in standard projects, is viewable through the Files window), you can jump straight to the project's node in the Files window without having to scroll to the node. See [Table 3-2](#) for some of the shortcuts that are particularly useful for navigating between the "explorer"-style windows.

Table 3-2. Shortcuts to Select the Current File's Node in a Different Window

Keyboard Shortcut	Action
Ctrl-Shift-1	Open the Projects window and display the node that corresponds to the currently selected file, package, or folder.
Ctrl-Shift-2	Open the Files window and display the node that corresponds to the currently selected file, package, or folder.
Ctrl-Shift-3	Open the Favorites window and display the node that corresponds to the currently selected file, package, or folder.

Ctrl-Shift-1 Open the Projects window and display the node that corresponds to the currently selected file, package, or folder.

Ctrl-Shift-2 Open the Files window and display the node that corresponds to the currently selected file, package, or folder.

Ctrl-Shift-3 Open the Favorites window and display the node that corresponds to the currently selected file, package, or folder.

You can also use these shortcuts in the Source Editor when you want to jump to the node for that file.

Physical Structure of IDE Projects

The Projects window provides a "logical" view of projects that is oriented toward coding activities and does not include all files in the project. For example, output files (such as compiled classes and JAR files), project metadata, and version control metadata files (if you are using version control) are hidden.

If you want to see the actual files created in the project, open the Files window (by clicking the Files tab above the Projects window).

When you create a general project (using a standard template), the IDE creates the following folders and files:

- `nbproject` folder, which includes files that store data about your project and are used to create the build script for your project.
- `src` folder, which holds your source files (assuming the IDE project was not created with existing sources). This folder corresponds with the Source Packages node that appears in the Projects window.
- `test` folder, which holds any unit tests you create for your project. This folder corresponds with the Test Packages node that appears in the Projects window.
- `build.xml` file, which is the Ant build script that is used when compiling, cleaning, running, and debugging your project. This file actually imports the `build-impl.xml` file, which is generated in the `nbproject` folder. See [Inside the Generated Build Scripts](#) later in this chapter for more information on the build script and files inside the `nbproject` folder.

When you build your project, the following folders are added:

- `build` folder, which holds the compiled class files.
- `dist` folder, which holds the packaged output for the project (a JAR file). If there are any JAR files on the project's classpath, the IDE creates the `lib` subfolder and places those JAR files there in order to simplify deployment of the finished application.

For web applications created from standard templates, the IDE creates the following folders and files:

- `nbproject` folder, which includes files that store data about your project and are used to create the build script for your project.
- `src` folder, which holds Java source files.
- `web` folder, which typically holds the `WEB-INF` and `META-INF` folders, as well as any HTML pages, JSP pages, custom tag libraries, and other files for the web application.
- `build.xml` file, which is the Ant build script that is used when compiling, cleaning, running, and debugging your project.

When you build your project, the following folders are added:

- `build` folder, which holds the compiled class files.
- `dist` folder, which holds the compiled application in distributable form (a WAR file).

Working with Files Not in the Project

When you set up a project in the IDE, only the files most likely to need hand editing (such as your source files) are exposed in the Projects window. You can access all files in your project using the Files window, but sometimes there are other files you might want to access. If there are other files on your system that you want to regularly have access to, you can display them in the Favorites window.

To use the Favorites window:

1. Choose Window | Favorites (Ctrl-3) to open the Favorites window as a tab in the area occupied by the Projects window.
2. In the Favorites window, right-click and choose Add Favorites.
3. In the file chooser, select the root folder that you want to add to Favorites.

You can add multiple folders to the Favorites window, and you can add folders at any level of a file hierarchy. This provides a lot of flexibility in what files you make visible. For example, if you have a deep package hierarchy, you can save a lot of time on folder expansion by adding folders that directly contain your files.

You can run most IDE commands on nodes in the Favorites window. However, project-specific commands (such as Build Main Project) are not available in Favorites, so you must choose those commands in the Projects window, through the main menu, or through keyboard shortcuts.

If you merely want to open a file in the IDE one time, you



can use the File | Open command.

Creating Packages and Files in the Project

Once the project is set up, you can create packages and files from the Projects window or the Files window. Right-click the node for the package or folder where you would like to add a class or package and choose New | File/Folder (Ctrl-N) to open the New File wizard. Or you can directly choose one of the templates below the File/Folder menu item.

To create a Java package:

1. Right-click the Source Packages node in the Projects window and choose New | Java Package.

If Java Package is not one of the choices in the New submenu, choose New | File/Folder instead. In the New File wizard, select the Java Classes node, select Java Package, and click Next.

2. In the Package Name field of the wizard, type the name of the package, delimiting levels of the package with periods (for example, `com.mydomain.myproject`), and then click Finish.



When creating subpackages, you can save yourself a few keystrokes by choosing New | Java Package from a package node. The base package name is already filled in for you, so you just need to add the last part of the new package name.

You can also enter a new package when using the New File wizard to create a new Java class.

To create a file:

1. Right-click the Source Packages node in the Projects window and choose New | File/Folder (Ctrl-N).
2. In the New File wizard, browse the templates available, select the one you want, and click Finish.



You can also select a template straight from the New submenu, where a short list of templates commonly used for the selected project type is displayed. The list of files that are available there is updated to reflect the templates you commonly use.

See [Chapter 5](#) for more information on editing Java files.

File Templates

File creation in the IDE begins with templates. The templates that are available depend on the features you have installed in the IDE. Following are some of the available categories.

Java Classes

Several templates that provide skeleton code for basic types of classes, such as main classes, interfaces, and exceptions.

Java GUI Forms

Swing and AWT templates for developing visual desktop applications. When you create a file from one of these

templates, the Form Editor opens, which enables you to build forms visually (dragging and dropping components from a palette to the Form Designer, changing properties in the Component Inspector, and so on).

JavaBeans Objects

Various templates for classes that adhere to the JavaBeans component architecture. Included are templates for a bean with a skeleton getter and setter, BeanInfo classes, a property editor, and a customizer class.

JUnit

Templates that provide skeleton code for unit tests of Java classes.

Web Services

Templates for web service clients and WSDL files.

XML

XML-related templates for XML documents, XML schemata, DTDs, XSL stylesheets, and cascading stylesheets.

Ant Build Scripts

Provides a simple template for a simple skeleton for an Ant script and a template for a custom Ant task with detailed comments. These scripts could be useful if you want to extend

the default behavior of the IDE's project system but are not necessary if the IDE's project system already provides all the features you need.

Web

Provides templates that are useful for web applications, including JSP files, HTML files, tag library files, and tag library descriptors (TLD files). Also provides Java class templates for servlets, filters, tag handlers, and web application listeners. See [Chapter 8](#) for more information on working with these types of files.

Other

Provides templates for HTML, properties, and empty files.

Starting with a Blank File

If you want to start with a completely blank file without a predetermined file extension, you can use the Other | Empty File template. If you give the file an extension that the IDE recognizes, the IDE will treat it as that type of file in the editor (complete with syntax highlighting and other Source Editor features).

Configuring the Project's Classpath

You can manage the classpath for your application through the Libraries node of the project's Project Properties dialog box. To get there, right-click the project's node in the Projects window, choose Properties, and then select the Libraries node in the dialog box that appears.

The IDE enables you to have different classpaths for compiling, testing, and running your application. When you set the compilation classpath, those items are automatically added to the other classpaths. Then you can add items specific to compiling tests, running the application, and running tests on the application.

The compilation classpath also affects some editor features, such as which classes are available in the code completion feature.

See the following topics for instructions on specific tasks.

Changing the Version of the JDK Your Project Is Based On

By default, IDE projects use the same JDK that the IDE runs on. To set up a project to run on a different JDK, you need to:

1. Add the JDK in the Java Platform Manager.
2. Specify the JDK for the project to use in the project's Project Properties dialog box.

By default, the IDE includes only the version of the JDK that the IDE is running on in the Java Platform Manager.

To make another JDK version available for projects in the IDE:

1. Choose Tools | Java Platform Manager.
2. Click Add Platform.
3. In the file chooser, navigate to and select the JDK version you want to add to the Java Platform Manager.

The JDK folder should be marked with the  icon in the file chooser.

4. Close the Java Platform Manager.

To switch the JDK that a project uses:

1. Right-click the project's main node in the Projects window and choose Properties.
2. Select the Libraries node.
3. Select the JDK that you want to use in the Java Platform combo box.

4. If the version of the JDK that you want does not appear in the list, click Manage Platforms to open the Java Platform Manager and click Add Platform to add a JDK to the list of those recognized by the IDE.



If you want to change the JDK that the IDE itself runs on, you can do so in the `netbeans.conf` file in the IDE's installation directory. In a file manager on your system, navigate to the IDE's installation directory, expand `NetBeansHome/etc`, and open `netbeans.conf` in a text editor. Below the `#netbeans_jdkhome="/path/to/jdk"` comment line, type the `netbeans_jdkhome` option with the path to the JDK.

For example:

```
netbeans_jdkhome="C:/j2sdk1.4.2_12"
```

The Default choice, which appears anywhere that you can specify the Java platform, then is changed to refer to the JDK you have specified.

Changing the Target JDK for a Standard Project

If you want to have the sources for your project compiled for a lower version of the JDK than the project is using, you can set a lower source level in the Project Properties dialog box.

To change the target JDK for your project:

- 1.** Right-click the project's main node in the Projects window and choose Properties.
- 2.** Select the Sources node.
- 3.** In the Source Level combo box, select the source level that you want to compile to.

Referencing JDK Documentation (Javadoc) from the Project

When you are coding in the IDE, it is often useful to have Javadoc documentation handy for the classes you are using. When you have Javadoc documentation on your system, you can freely browse it from within the IDE by jumping to a browser from the class you have currently selected in your code (Alt-F1).

The JDK documentation is not included with the standard JDK download. If you do not have the JDK documentation on your system, you can get it from <http://java.sun.com> (for JDK 5.0, go to <http://java.sun.com/j2se/1.5.0/download.jsp>).

To make JDK documentation viewable in the IDE:

- 1.** Choose Tools | Java Platform Manager.
- 2.** Select the Javadoc tab.
- 3.** Click the Add ZIP/Folder button. Then navigate to and select the JDK documentation .zip file or folder on your system.
- 4.** Close the Java Platform Manager.



You do not need to have the JDK's Javadoc on your system to see Javadoc when you are using code completion. The code completion box gets Javadoc comments straight from the JDK source code.

Adding Folders and JAR Files to the Classpath

If your project relies on prebuilt binaries or classes that are not part of a NetBeans IDE project, you can add these to your classpath as well.

To add a JAR file or a set of classes to your project classpath:

1. Expand the project's node in the Projects window.
2. Right-click the project's Libraries node and choose Add JAR/Folder. In the file chooser, navigate to and select the folder or JAR file and click Open.



If the JAR file you want to add to the classpath is built from another project in the IDE, you can link the project so that JAR is rebuilt each time you build your current project. See [Structuring Your Projects](#) later in this chapter.

Making External Sources and Javadoc Available in the IDE

You might want to associate documentation and source with classes your project depends on. This is useful if you want to do any of the following:

- As you are debugging, see the source when stepping into a class your project depends on.
- Jump to the source of a referenced class from the Source Editor (Alt-O).
- Jump to the Javadoc documentation of a source file from within the IDE (Alt-F1).

The IDE has a Library Manager feature that enables you to declare these associations, which you can then take advantage of for all your projects.

To create a library:

1. Choose Tools | Library Manager and then click the New Library button.
2. In the New Library dialog box, type a display name for the library and click OK.
3. In the Class Libraries list, select the new library.
4. Select the Classpath tab and click Add JAR/Folder. Then select the JAR file or the folder that contains the classes and click Add JAR/Folder.
5. If you want to associate sources with the library, select the

Sources tab and click Add JAR/Folder. Then select the JAR file or the folder that contains the sources and click Add JAR/Folder.

6. If you want to associate Javadoc documentation with the library, select the Javadoc tab and click Add ZIP/Folder. Then select the JAR file or the folder that contains the documentation and click Add ZIP/Folder.
7. Click OK to close the Library Manager.

To add a library to your project:

1. Expand the project's node in the Projects window.
2. Right-click the project's Libraries node and choose Add Library.



When a library with a JAR file, sources, and documentation is designated in the Library Manager, the IDE recognizes the association of those elements even if you only add the JAR file to your project.

If the library you have designated consists of more than one JAR file, the other JARs are not duplicated on the classpath if you have added them to the classpath individually.

Structuring Your Projects

Standard IDE projects are essentially modular. If your application needs to be built into just one JAR file, you can build it with a single project. If your application exceeds that scope, you can build your application from multiple IDE projects that are linked together with one of those projects declared as the main project.

Setting the Main Project

The main project is the one that project-specific commands (such as Build Main Project) in the main menu always act on. When an application is composed of many related projects, the main project serves as the entry point for the application for purposes of compiling, running, testing, and debugging. You can have only one main project set at a time.

To make a project the main project, right-click that project's node and choose Set Main Project.



If you find that your main project inadvertently gets changed from time to time, it might be because some project templates contain an option to make the new project the main project. If you create a new project that you do not want to be a main project, make sure to deselect the Set As Main Project checkbox in the New Project wizard.

Creating Subprojects

For more complex applications, you might need to create multiple IDE projects, where one project is the application entry point that depends on other projects. (Any project that has another project depending on it functions as a subproject, though it is not specifically labeled as such in the IDE.)

Each IDE project can create one distributable output (such as a JAR file), which in turn might be used by other projects. There is no particular limit to how long a chain of dependencies can be, though the chain must not be circular. (For example, if project A depends on classes in project B, project B cannot depend on classes in project A.)



Although it might be a hassle at first, reorganizing your code to eliminate circular dependencies will probably pay off in the long run by making your code easier to maintain and extend.

To make one project dependent on another project:

1. Expand the project's node in the Projects window.
2. Right-click the project's Libraries node and choose Add Project. In the file chooser that appears, navigate to the IDE project folder. All project folders are marked with the icon in the file chooser.



When you add a project to another project's classpath, all sources and Javadoc documentation are recognized by the other project as well.

Displaying and Hiding Projects

You can have multiple IDE projects open at the same time, whether or not they are connected to one another in any way. However, the only projects you *need* to have open at any given time are the main project (which serves as an entry point for building, running, and debugging) and the projects you are currently editing.

Even if your main project depends on other projects (subprojects), these projects do not need to be open if you are not actively working on them. Any dependencies that you have set up in the Project Properties dialog box are honored whether the subprojects are open or closed.

To hide a project, right-click the project's node in the Projects window and choose Close Project.

To display a project, choose File | Open Project (Ctrl-Shift-O).

To open all the projects that a project depends on, right-click the project's node in the Projects window and choose Open Required Projects.

Compiling a Project

Once you have set up your project, created your source files, and set up any necessary dependencies, you can compile your project by choosing Build | Build Main Project (F11).

By default, when you build a standard Java project in the IDE, the following occurs:

- All modified files in the project are saved.
- Any uncompiled files (or files that have been modified since they were last compiled) under the Source Packages node are compiled.
- A JAR file (or other archive, depending on the project category) is created with your classes and other resources.
- Any projects the main project depends on are built. (In fact, these projects are built first.)

When you build your project, output on the progress of the Ant script is printed to the Output window.



If you merely want to compile your project (without building JAR files or running any other postcompile steps), you can create a shortcut to the `compile` target in the project's build script. Open the Files window and expand the `build.xml` node. Rightclick the `compile` target and choose Create Shortcut. In the wizard, you can designate a menu item, toolbar button, and a keyboard shortcut for the target. The shortcut works only for that particular project.

Setting Compiler Options

You can set compiler options in the Project Properties dialog box for a project.

1. Right-click the project's node and choose Properties.
2. In the Project Properties dialog box, select the Build | Compile node.
3. Select the checkbox for any options you want to have included.
4. For options that are not covered by a checkbox, type the option in the Additional Compiler Options field as you would when compiling from the command line.

Compiling Selected Files or Packages

If you have a larger project, you might want to be able to compile a few files without rebuilding the entire project.

To compile a single file, right-click the file's node and choose Compile File (F9).

To compile a package, right-click the package's node in the Projects window and choose Compile Package (F9).

Doing a Fresh Build

The Build Project command produces its outputs incrementally.

When you rebuild a project, any files that have changed or have been added are recompiled, and the JAR file (or other output file) is repackaged with these changed classes. As your project evolves, sometimes you need to explicitly erase your compiled classes and distributables, particularly if you have removed or renamed some source files. Otherwise, classes that are no longer part of your sources might linger in your compiled outputs and cause problems with running your program.

To clean your project, right-click the node for your project in the Projects window and choose Clean Project. This command deletes the `build` and `dist` folders in your project.

To do a fresh build, choose Build | Clean and Build Main Project (Shift-F11).

Stopping a Build

If you start a build and want to halt it before it completes, choose Build | Stop Build/ Run. The build will terminate, but any artifacts created by the build will remain.

Changing the Location of Compiled Classes and JAR Files

By default, the compiled class files and the packaged distributable (for example, a JAR or WAR file) are placed in folders (named `build` and `dist`, respectively) parallel to the `src` folder in your project. If you would like the outputs to appear elsewhere or simply would like to change the names of the folders, you can do so in the `project.properties` file.



If you want to have your class files placed in the same directory as their source files, you need to override the

project's `clean` target so that only class files are deleted from the directory when the Clean command is run. You also need to adjust the IDE's Ignored Files property. See [Compiling Classes into the Same Directories As Your Sources](#) later in this chapter.

To access the `project.properties` file:

1. Open the Files window by clicking the Files tab (or pressing Ctrl-2).
2. Expand the project's `nbproject` folder and open the `project.properties` file.

[Table 3-3](#) lists properties that determine where your outputs are created for general Java and web projects.

Table 3-3. Ant Properties for Build Outputs in Standard Projects

Property	Specifies the Directory...
<code>build.classes.dir</code>	Where compiled class files are created. You can change the name of the output folder here.
<code>build.test.classes.dir</code>	Where unit test files are created (for general Java and web projects).
<code>build.test.results.dir</code>	Where unit test results are stored (for general Java and web projects).
<code>build.generated.dir</code>	Where the IDE places various temporary files, such as servlet source and class files generated by the IDE when you run the Compile JSP command. These files are not included in the WAR file.

`build.web.dir` Where the `WEB-INF` folder and other key folders are placed in the built web application (for web projects).

`build.dir` Where the above directories for the previously listed properties are placed. You can also change the value of this property. If you remove this property from the values of other properties, you must also override any targets that use this property (such as the `clean` target).

`dist.jar` Where the project's JAR file is created. Also specifies the name of the JAR file (for general Java projects).

`dist.war` Where the project's WAR file is created. Also specifies the name of the WAR file (for web projects).

`dist.javadoc.dir` Where Javadoc for the project is generated when you run the Generate Javadoc for Project command.

`dist.dir` Where the generated Javadoc documentation and the project's JAR file or WAR file are placed.



If you move either the `build` folder or the `dist` folder outside of the project's folder, it will no longer be visible in the Files window. You can remedy this by setting up the Favorites window to display the moved folder. Choose Window | Favorites (Ctrl-3). Then right-click in the Favorites window and choose Add to Favorites. Navigate to the folder you want to display and click Add.

Compiling Classes into the Same Directories As Your Sources

If your build environment requires that you compile your classes into the same directories as your sources, you must:

- Modify the properties that represent any build outputs that you want to have moved.
- Override the IDE's `do-clean` target so that it deletes only the class files in your source directory and not your sources as well.
- Optionally, update the IDE's Ignore Files property so that compiled class files are not displayed in the Projects window.

Here is the step-by-step procedure for having your classes compiled into the same directories as your sources for a general Java project:

1. Open the Files window by clicking the Files tab (or pressing **C**).
2. Expand the project's `nbproject` folder and open the `project.properties` file.
3. Change the `build.classes.dir` property to point to your source directory.
4. Open the `build-impl.xml` file and copy the `do-clean` target.
5. Paste the `do-clean` target into your `build.xml` file.
6. In the `build.xml` file, modify the `do-clean` target so that only clas

are deleted. For example, you might replace

```
<delete dir="${build.dir}"/>
```

with

```
<delete includeEmptyDirs="true">
    <fileset dir=".">
        <include name="${build.classes.dir}/**/*.*.cla
        </fileset>
    </delete>
```

7. If you do not want compiled class files to display in the Project and Files windows, choose Tools | Options, click Advanced Options, expand IDE Configuration | System, and select the System Settings node. Select the Ignored Files property and add to the regular expression to designate the files to ignore. For example, you can add `|class$` to the end of the expression.

Investigating Compilation Errors

If you get an error when compiling a class, you can jump to the source of the error by clicking the hyperlinked error line in the Output window or by pressing F12. If there are multiple errors, you can cycle through them by pressing F12 (next error) or Shift-F12 (previous error).

Saving Build Output

You can save build output to a file by right-clicking in the

Output window and choosing Save As.

By default, every command that you run in the IDE that uses Ant re-uses the same tab in the Output window; thus, the output from the previous command is cleared every time you run a new build command. If you would like to preserve the Ant output in the UI, you can configure the IDE to open a new tab every time you run a build command.

To preserve the output from all build commands in the IDE:

- 1.** Choose Tools | Options and click Miscellaneous in the left pane and expand the Ant node.
- 2.** Deselect the Reuse Output Tabs From Finished Processes checkbox.

Running a Project in the IDE

Once you have set up your project, created your source files, and set up any necessary dependencies, you can run your project in the IDE by choosing Run | Run Main Project (F6).

By default, when you run a general Java project in the IDE, the following occurs:

- All modified files in the project are saved.
- Any uncompiled files (or files that have been modified since they were last compiled) under the Source Packages node are compiled.
- A JAR file with your classes and other resources is created (or updated).
- Any projects that the main project depends on are built. (In fact, these projects are built first.)
- The project is run inside of the IDE. For general Java projects, the IDE uses the designated main class as the entry point.

When you build your project, output on the progress of the Ant script is printed to the Output window.

Setting or Changing the Project Main Class

To run a general Java project, you have to have an executable class designated as the entry point. You can designate the main

class:

- In the New Project wizard when creating your project from the Java Application project template.
- In the project's Project Properties dialog box. (In the Projects window, right-click the project's node and choose Properties. Then select the Run node and fill in the Main Class field with the fully qualified name of the main class, without the `.java` extension.)



If you try to run a project that has no main class set, you will be prompted to choose one from a list of executable classes found within the project's source packages. If the class that you want to use as the main class is not among the project's sources (for example, if it is in a subproject or another JAR on your classpath), you can set that class in the Project Properties dialog box as detailed in the procedure above.

Setting Runtime Arguments

If you need to pass arguments to the main class when you are running your project, you can specify those arguments through the Project Properties dialog box:

1. Right-click the project's node in the Projects window and choose Properties.
2. Select the Run node and add the arguments as a space-separated list in the Arguments field.

Setting Java Virtual Machine Arguments

If you need to pass arguments to the Java virtual machine that is spawned to run the project in, you can do so through the Project Properties dialog box:

1. Right-click the project's node in the Projects window and choose Properties.
2. Select the Run node and add the arguments as a space-separated list in the VM Options field.

Setting the Runtime Classpath

By default, the classpath for running the project inherits the compilation classpath. If you need to alter the classpath just for runtime, you can make adjustments in the Running Project section of the project's Project Properties dialog box:

1. Right-click the project's node in the Projects window and choose Properties.
2. Select the Libraries node.
3. In the tabbed panel on the right side of the Project Properties dialog box, click the Run tab.
4. Use the Add Project (for other IDE projects), Add Library (for collections of JARs, sources, and Javadoc that you have designated in the Library Manager), or Add JAR/Folder buttons to make additions to your classpath.

Deploying a Java Project Outside of the IDE

When you have finished a general Java project and are ready to distribute it to others as an application, you can do so fairly easily. The IDE helps you assemble the parts that will go into the final application.

When you build a general Java application where you have specified a main class, the IDE does the following:

- Creates a JAR file containing the project's compiled classes and other resources and places the JAR file in the project's `dist` folder.
- Copies the JAR files that are on the project's classpath and pastes them into the `dist/lib` folder. Note, however, that any folders on the classpath are not similarly copied.
- In the `Class-Path` element of the `manifest.mf` file that is included in the application's JAR file, lists each of the JAR files that were copied to the `dist/lib` folder.

The `dist` folder (including its `lib` subfolder) thus should contain everything the application needs to run on its own. You can distribute the application by packaging the entire contents of the `dist/lib` folder into a zip file.

You can instruct users to run the application by doing the following:

1. If necessary, unzip the application.
2. At the command line, navigate to the `dist` folder and type the following:

```
java -jar JarFileName.jar
```

The application is run on the user's system, with the class specified in the `Main-Class` as the application's entry point.

The work the IDE does to make your application easier to distribute is enough for most cases, but there are some cases where you might need to do some further manual configuration, such as when:

- There are folders on the project's classpath (these folders are not automatically copied to the `dist/lib` folder like JAR files are).
- Two or more JAR files on the project's classpath have the same name. In this case, the IDE only copies the first JAR file it finds with that name to the `dist/lib` folder. You would have to create a place for the other JAR files and move them manually (as well as update the manifest `Class-Path` element).
- One of the JAR files included in the `dist/lib` folder also uses the `Class-Path` element. If that is the case, you will have to add the items that are listed there to the `Class-Path` element of your application's main JAR file.

See the subtopics below for further information on things you can do to prepare your applications for distribution.

Writing Your Own Manifest for Your JAR File

When you build a general Java project, a JAR file is created with a simple manifest with entries for `Manifest-Version`, `Ant-Version`, and `Created-By`. If the project has a main class designated, that main class is also designated in the JAR manifest.

If you have other entries that you would like to add to the manifest of a project created from the Java Application template or Java Project with Existing Sources template, you can add them directly in the `manifest.mf` file that sits next to the `build.xml` file. Go the Files window and double-click the `manifest.mf` file's node to edit in the Source Editor.

You can also specify a different manifest file for the project to use. To specify a custom manifest:

1. Open the Files window by clicking the Files tab (or by pressing Ctrl-2).
2. Expand the project's `nbproject` folder and open the `project.properties` file.
3. In the `manifest.file` property, type the manifest's name. If the manifest is not in the same folder as the `build.xml` file, include the relative path from the `build.xml` file.



You can write the manifest in the IDE's Source Editor by using the Empty File template. Open the Files window. Then right-click the project's main folder and choose New | Empty File.

For projects created from the Java Library template, a basic manifest is generated, but no editable copy of that manifest appears in the Files window. If you would like to specify a different manifest for a project created from the Java Library

template, simply add the `manifest.file` property to the project's `project.properties` file and point it to the manifest you have created.

Filtering Contents Packaged into Outputs

If you have any files that appear within your source packages but that you do not want packaged in the project's distributable, you can have those files filtered out of the project's distributable.

By default, `.form` files and `.java` files are filtered from the output JAR file of general Java projects. In web projects, `.form`, `.java`, and `.nbattrs` files are filtered out by default.

To change the filter for the JAR file contents of a general Java project:

1. Right-click the project's node in the Projects window and choose Properties.
2. Select the Build | Packaging node and modify the regular expression in the Exclude from JAR File field. (The name of this field depends on the type of project. For web projects, the field is called Exclude from WAR File.)



`.form` files are created for classes you create with the IDE's Form Editor. The IDE uses these files to regenerate the design view of those classes; however, they are not necessary for the packaged application. If you delete a `.form` file, the corresponding `.java` file remains, and you can edit the code, but you can no longer use the IDE's Form Editor to change the form.

`.nbattrs` files are files that NetBeans IDE creates to hold information about given directories. Projects created in NetBeans IDE 4.0 and later are unlikely to have these files, but legacy projects might.

Building a Project from Outside of the IDE

Because the IDE's project commands are based on Ant scripts and properties files, you can run these targets from outside of the IDE.

Assuming that you have Ant installed on your system, you simply can call the `build.xml` file or one of its targets from the command line by changing directories to the directory holding `build.xml` and typing `ant`.

If your project uses optional Ant tasks that are defined within the IDE, you might need to do some manual configuration. For example, if you have a target that depends on JUnit, you need to place the JUnit binary in your Ant classpath.

Setting up a Headless Build Environment

If you have a large nested project structure that you want to run from outside of the IDE, such as when you are doing production builds of your application, you will probably need to make some adjustments to set up headless builds.

Following are the tasks you need to perform:

- Set up Ant (version 1.6 or higher) on your system. You can either download Ant or use the Ant JAR file that is included with the IDE (`NBHome/ide6/ant/lib/ant.jar` in the IDE's installation directory). Visit <http://ant.apache.org/> to download or get more information on Ant. NetBeans IDE 5.0 is bundled with Ant version 1.6.5, so any standard projects you set up will use that version. Also, make sure the command-line version of Ant you are using is running from the same version of the Java platform that your

project is using.

- Make JUnit available to Ant. You can do this by adding the JUnit JAR file to `AntHome/lib`. You can use the IDE's copy of JUnit, which is located in `NBHome/ide6/modules/ext` and is named according to version number. (In NetBeans IDE 5.0, for example, it is `junit3.8.1.jar`.)
- Make sure any libraries the IDE uses when building your project are accessible from the build script that the build machine uses. These libraries are specified in the `build.properties` file located in your IDE user directory. You can find the IDE's user directory by choosing Help | About, clicking the Detail tab, and looking at the User Dir value.

Customizing the IDE-Generated Build Script

The build scripts the IDE generates for you in standard projects are based on common scenarios that work for many development situations. But if the script does not do what you want it to do, you can add to, override parts of, or change the script entirely.

When you create a standard project, the IDE generates two build scripts: `build-impl.xml` and `build.xml`.

The `build-impl.xml` file is generated based on the type of project template you started with and is regenerated based on any changes that occur in the project's associated `project.xml` file. Do not edit `build-impl.xml` directly, because any changes you make there will be lost any time the file is regenerated.

The `build.xml` file serves as the master build script. By default, it has no targets of its own. It imports `build-impl.xml`. You can freely edit `build.xml`.



To help make sense of the Ant script, you can use the Ant Debugger to step through execution of the script so you can quickly see the order in which the various targets are called. See Debugging Ant Scripts in [Chapter 16](#) for more information.

Adding a Target

To add a target to the build script of a standard project:

1. In the Files window, expand the project's main folder.
2. Double-click the `build.xml` file to open it in the Source Editor.
3. Below the `import` element, type any targets you would like to add to the build script.

Adding a Subtarget

To make customization of build scripts easier, the generated build scripts include several empty targets that are called from main targets.

For example, the `compile` target depends on `pre-compile` and `post-compile` targets, which have nothing in them. If you need to add any steps to the build process just before or after compilation of your files, you can customize these targets without having to add to the `depends` attribute of the `compile` target.

To add to a target using one of the existing empty subtargets:

1. In the Files window, expand the project's `nbproject` folder and open the `build-impl.xml` file.
2. Copy the empty target you want to use, paste it into the project's `build.xml` file, and then make your modifications to it there.

For example, you could call Ant's `rmic` task in the `post-compile` target to run the `rmic` compiler on all classes with names beginning with `Remote` (as shown in the snippet below):

```
<target name="-post-compile">
    <rmic base="${build.classes.dir}" includes="**/Remote"
</target>
```

For further convenience, some of the main targets also include `"-pre-pre"` targets that handle some basic steps before the empty targets are called. For example, the `-pre-pre-compile` target, which creates the directory to hold the class files to be compiled, is called before the `-pre-compile` target.

Overriding an Existing Target

If adding subtargets to a main target is not sufficient for what you need to accomplish, you can completely override part of a build script.

To override a target in a standard project's build script:

1. In the Files window, expand the projects `nbproject` folder and open the `build-impl.xml` file.
2. Copy the target you want to override, paste it into the project's `build.xml` file, and then make your modifications to it there.

When a target appears in both the `build-impl.xml` and `build.xml` files, the version in the `build.xml` file takes precedence.

If you are merely modifying the `depends` attribute of the target, you do not have to copy the whole body of the target. You can copy just the `target` element without its subelements. The subelements will be imported from the `build-impl.xml` file.

Inside the Generated Build Scripts

Here is a look at all of the pieces of the project metadata and how they work together.

When you create a standard project, the IDE creates the files

listed in [Table 3-4](#).

Table 3-4. Metadata Files for a Standard Project

File	Description
<code>build.xml</code>	This script is the master build script for the project. When you call a project-related command from the IDE, the IDE calls a target in this file. You can freely edit this file if you want to make customizations to your build process. This file is generated when you create the project but is not regenerated afterward. Any configuration you do in the IDE that is relevant to the build script is reflected in the <code>build-impl.xml</code> file, which is imported by <code>build.xml</code> . If a target with the same name appears in both <code>build.xml</code> and <code>build-impl.xml</code> , the target in <code>build.xml</code> takes precedence.
<code>nbproject/build-impl.xml</code>	Included in standard projects (but not free-form projects), this file contains the meat of the build script and is imported by <code>build.xml</code> . It is generated based on the type of project and the contents of that project's <code>project.xml</code> file. Do not edit this file.
<code>nbproject/project.properties</code>	Included in standard projects (but not free-form projects), this file contains values that the build script uses when building your project. These values include things such as the name and location of the directory for your compiled files and references to properties set elsewhere in the project. Changes you make in your Project Properties dialog box are propagated here. You can also modify this file directly in the Source Editor.
<code>nbproject/project.xml</code>	Provides the basic metadata that determines how the project works in the IDE. For standard projects, this file determines how <code>build-impl.xml</code> and <code>project.properties</code> are generated. For free-form projects, this file serves as the glue between your build script and the IDE's user interface. This file is generally editable, but you are likely to need to edit it only for freeform projects.

`nbproject/genfiles.xml`

Used by the IDE to help keep track of the state of the build script (such as whether the `build-impl.xml` file needs to be regenerated). Do not edit this file.

`nbproject/private/private.properties`

Holds properties that are specific to your installation of the IDE. These properties are not to be versioned, but they can be used by headless builds run on your machine.



There is also a `build.properties` file that is created in your IDE's user directory. This file holds properties for the location of libraries that are packaged with the IDE, any libraries you specify with the IDE's Library Manager, and any versions of the Java platform you register with the IDE's Java Platform Manager. The `private.properties` file references the `build.properties` file with its `user.properties.file` property.

Running a Specific Ant Target from the IDE

You can run any Ant target within the IDE by expanding the build script's node in the Files window, right-clicking the target, and choosing Run Target.

You can stop the target by choosing Build | Stop Build/Run.

Completing Ant Expressions

The IDE has a "completion" feature to reduce the number of keystrokes needed when editing an Ant script. When you use the completion feature, the Source Editor gives you a choice of how to complete the current word with a popup dialog box.

Activate the completion feature by typing the first few characters of an element or attribute and then pressing Ctrl-spacebar.

If there is only one possible completion, the missing characters from the word are filled in. (For attributes, `="` is generated as well.)

If there are multiple possible matches, you can choose a completion by scrolling through the list and then pressing the Enter key once the correct word is selected. Keep typing to narrow the number of selections in the list.

The order of the selection list is generally *smart*, meaning that elements or attributes that are commonly used in the given context are put at the top.

For example, you can enter the following target

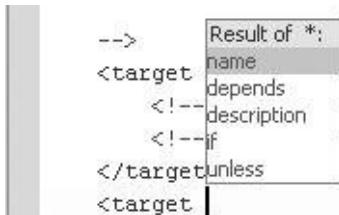
```
<target name="default" depends="dist,javadoc"  
       description="Build whole project."/>
```

by doing the following:

- 1.** Typing `<t`
- 2.** Pressing Ctrl-spacebar and then Enter to select `target`

- 3.** Pressing the spacebar
- 4.** Pressing Ctrl-spacebar and then Enter to select `name`, as shown in [Figure 3-8](#) (`name="` is inserted into the script)

Figure 3-8. Code Completion dialog box for an Ant script



- 5.** Typing `default"`
- 6.** Pressing Ctrl-spacebar and then Enter to select `depends` (`depends="` is inserted into the script)
- 7.** Typing `dist,javadoc"`
- 8.** Pressing Ctrl-spacebar and then Enter to select `description` (`description="` is inserted into the script)
- 9.** Typing `Build whole project."/>`

Making a Menu Item or Shortcut for a Specific Ant Target

If you have a target in your build script that you use often but that is not represented by a menu item or keyboard shortcut, you can create such a menu item and a shortcut:

1. In the Files window, expand the projects `nbproject` folder and expand the `build-impl.xml` node.
2. Right-click the target you want to map and choose Create Shortcut.

A wizard opens that enables you to set a menu item, toolbar item, and a keyboard shortcut for the target. Because the IDE records these custom menu items and shortcuts as Ant files, you can customize the way the shortcuts are called by selecting the Customize Generated Ant Code checkbox in the wizard.

Shortcuts set in this way are not global. They apply only to the build script on which they are set.



The wizard prevents you from overriding existing custom shortcuts, but it does not prevent you from overriding standard IDE shortcuts. If you inadvertently use a key combination for your new shortcut that is used elsewhere in the IDE, your new shortcut takes precedence.

Removing a Custom Menu Item or Shortcut

If you have added a menu item or shortcut for an Ant target and would like to remove it, you can do so manually. The IDE stores these customizations in your user directory in small XML files that work as mini-Ant scripts.

To remove a shortcut to an Ant target:

1. In your IDE's user directory, expand the `config` folder. (If you are not sure where your IDE user directory is, choose Help | About and click the Details tab to find out.)
2. Look in the following subfolders (and possibly their subfolders) for an XML file with the name of the Ant target or the keyboard shortcut:
 - Menu (if you have created menu items)
 - Shortcuts (if you have created keyboard shortcuts)
 - Toolbar (if you have added a toolbar item)
3. Manually delete the XML file that represents the shortcut.

Changing a Custom Menu Item or Shortcut

You can change a custom menu item for an Ant target by doing one of the following:

- Deleting the mini Ant script for the target and creating a new shortcut.
- Editing the mini Ant script for the target. For example, you can change the build script that a shortcut applies to by changing the `antfile` attribute in the shortcut's script.

To manually edit the file for the custom menu item or shortcut:

1. In your IDE's user directory, expand the `config` folder. (If you are not sure where your user IDE directory is, choose Help | About and click the Details tab to find out.)
2. Look in the following subfolders (and possibly their subfolders) for an XML file with the name of the Ant target or the keyboard shortcut:

Menu (if you have created menu items)

Shortcuts (if you have created keyboard shortcuts)

Toolbar (if you have added a toolbar item)

3. Double-click the node for the menu item or shortcut to open it in the Source Editor.

Chapter 4. Versioning Your Projects

- [Setting up CVS in NetBeans IDE](#)
- [Checking Out Sources from a CVS Repository](#)
- [Putting a Project into CVS](#)
- [Keeping Track of Changes](#)
- [Updating Files](#)
- [Committing Changes](#)
- [Ignoring Files in CVS Operations](#)
- [Adding and Removing Files from a Repository](#)
- [Working with Branches](#)
- [Working with Patches](#)
- [Working with Versioning Histories](#)
- [Working with Other Version Control Systems](#)

USING A VERSION CONTROL SYSTEM (VCS) to share your projects and integrate contributions from multiple developers is one of the most important parts of working in a group development environment. In NetBeans IDE, version control functionality is integrated right into your daily work flow, so you do not have to keep switching to the command line or an

external tool to perform updates and commit local changes.

You can choose to share just your sources or to share the NetBeans projects with which you are working. When you check the NetBeans project metadata into the repository, other developers who are working on the project in NetBeans IDE do not have to go through the trouble of setting up the projects themselves they can just check out the projects and start working immediately.

Version control support in the IDE provides:

- Display of VCS status in the IDE
- Ability to run VCS commands on files from within the IDE
- Tools to help you check in changes to the repository and merge between branches
- Advanced tools for searching file version history and viewing differences between file revisions

The IDE provides its most comprehensive support for Concurrent Versioning System (CVS), with versioning actions integrated into the IDE's project system.

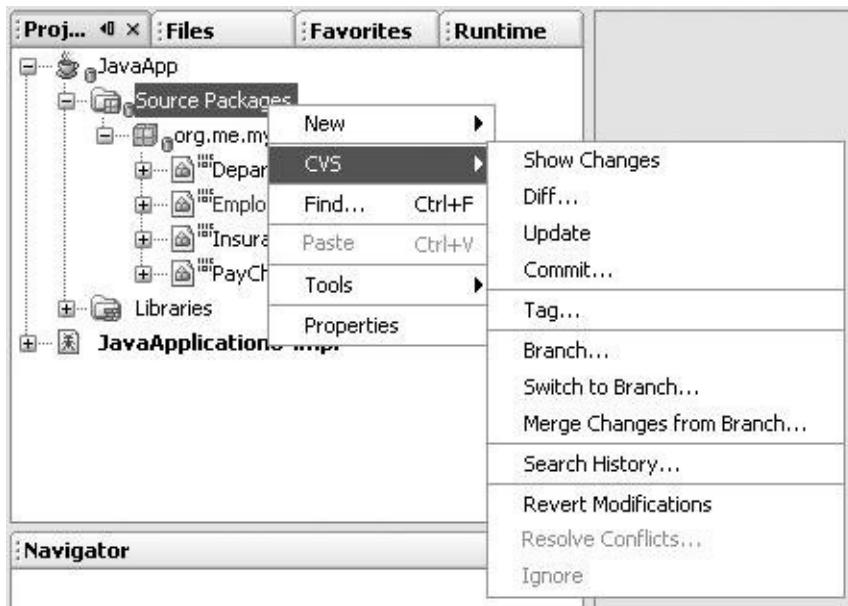
In addition to the "projects-aware" CVS support, you can download support for a wide range of version control systems, from Subversion to Microsoft Visual SourceSafe, via the Update Center. See [Working with Other Version Control Systems](#) later in this chapter for more information.

Setting up CVS in NetBeans IDE

For the most part, there is no setup necessary to work with files in CVS. Whenever you access files that are in a CVS working directory, the IDE offers the full range of CVS commands and status display for the files.

You can tell that a directory or project is in a CVS working directory by right-clicking the project and choosing CVS. If the CVS menu looks like [Figure 4-1](#), the IDE is recognizing the project as a CVS working directory.

Figure 4-1. CVS menu for a project in a CVS working directory



Checking Out Sources from a CVS Repository

To check out sources from a CVS repository:

1. Choose CVS | Checkout from the main menu.
2. On the first page of the CVS Checkout wizard, specify the location of the CVS repository by choosing it from the CVS Root drop-down menu. The IDE automatically offers all of the existing CVS roots from your `.cvspass` file, which is usually located in your home directory. If you are unfamiliar with the syntax, you can click the Edit button and enter the required information using the dialog.
3. Depending on which connection method you chose, you may have to enter additional information. The available connection methods are as follows:

`:pserver`. Connects to a remote server using password authentication. You have to specify your password.

`:ext`. Connects to a remove server using SSH. You have to specify your password or an external SSH command to run. See the [Connecting with SSH](#) section below for details.

`:local`. Connects to a local repository on your computer. Although you do not have to specify any additional information, you do have to correctly configure the CVS repository on your machine. See the [Connecting to a Local CVS Repository](#) section below.

`:fork`. A different implementation of the `:local` connection method that allows you to connect to a local CVS repository.

4. On the Module to Checkout page, specify the modules you want to check out. You can use the Browse button to browse all of the available modules in the repository. To

check out the entire repository, leave the Modules field empty.

5. If you want to check out a branch or a specific revision of the repository, specify the specific branch name, revision number, or tag. You can click the Browse button to choose from a list of all branches in the repository. If you just want to check out the latest version of the trunk, leave this field empty.
6. Specify the local working directory into which you want to check out the selected modules or branches.
7. Click Finish to check out the files.

When you check out sources from a CVS repository, NetBeans IDE immediately scans the sources for existing NetBeans projects. If the sources are already in a NetBeans project, the IDE lets you open the project immediately.

If the sources are not in a project, the IDE lets you immediately create a project from them. You should use one of the templates for creating projects from existing source code or with an existing Ant script. See [Chapter 3](#) for more information.

Connecting with SSH

The `pserver` connection method is inherently insecure, since it transmits your user name and password over the Internet in decrypted form. To guarantee your information is secure when you connect to a remote repository, you should set up a Secure Shell (SSH) tunnel to the repository through which you can communicate with the repository.

The IDE has an internal SSH client that you can use without having an external SSH client installed on your computer. If you

are using public key authentication, you have to use an external SSH client.

To connect to a repository using the internal SSH client:

1. Choose CVS | Checkout from the main menu.
2. Type `:ext:` in the CVS Root field. The wizard automatically updates to display fields for SSH information.
3. Enter the CVS root in the following format:

`:ext:username@hostname:/repository_path`

For example, to connect to the NetBeans test project on SourceForge, you would enter the following URL:

`:ext:myDeveloperName@cvs.sourceforge.net:/cvsroot/`

4. Select Use Internal SSH and enter your password. It is recommended that you select the option to remember your password. Otherwise, the IDE asks you for a password whenever you perform a CVS command.
5. Set your proxy information if necessary.
6. Click Next to continue checking out modules.

If you need to use an external SSH client, you can select the Use External Shell option in the Checkout wizard. You enter the SSH command exactly as you would run it from the command line. The IDE automatically appends the user name, server name, and CVS command that is executed to the SSH command.

A few notes about using an external SSH client:

- You must run the client in non-interactive mode, meaning

the commands can execute without any input from the user. If the command you enter requires that the user input his or her password or any other information, the command does not work with NetBeans IDE. You should therefore use public key authentication to verify your identity.

- Your SSH command must handle firewall tunnelling if you are behind a firewall.
- You either have to make the SSH executable available on your computer's path or specify the absolute path to the executable in the command string.

Connecting to a Local CVS Repository

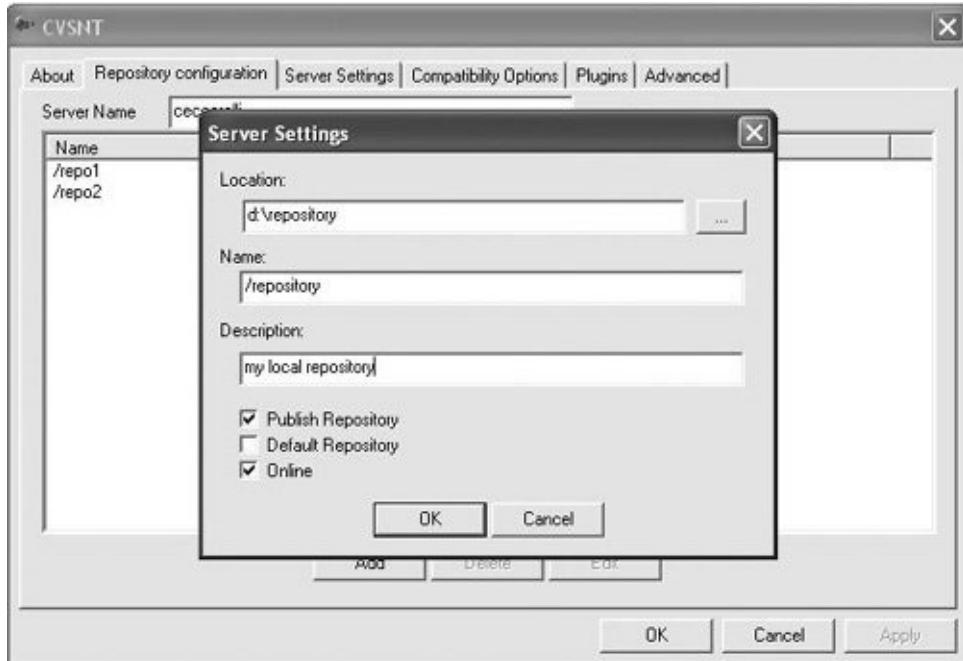
In addition to connecting to repositories on remote servers, you can also set up a local CVS repository on your computer and connect to it with the `:local` connection method. If you are working on a UNIX system, there is no additional configuration needed to use the `:local` connection method. On a Windows machine, you have to install and configure special software.

To set up a local repository on a Windows machine:

1. Download and install CVSNT. You can download CVSNT for free at <http://www.march-hare.com/cvspro>.
2. Use the CVSNT Control Panel to create a new repository, as shown in [Figure 4-2](#).

Figure 4-2. Using CVSNT Control Panel to set up a local CVS repository

[\[View full size image\]](#)



3. Set up an environment variable called `cvs_EXE` that points to the location of the `cvs.exe` executable in your CVSNT installation.
4. When doing a CVS checkout as described above, use the following syntax for your CVS root: `:local:/repository-name`

Putting a Project into CVS

When you create a NetBeans project for your source files, the IDE creates a build script, properties files, and other project metadata for the project in your project folder. See [Inside the Generated Build Scripts](#) in [Chapter 3](#) for a full description of the project metadata files. In addition to sharing your source files, you can also check the project metadata into CVS. Other developers who use NetBeans IDE can then check out the project and start working with it immediately without having to configure their own project. This is especially important for free-form projects, for which you have to write your own Ant targets to do things like running the project in the debugger.

One of the most important parts of adding projects to CVS is making sure that references to resources, such as JAR files on the classpath, source folders, and other projects, are correctly defined. If these references are not correctly defined, the project will build fine on your machine but not build or run correctly when other users check the project out of the repository.

Normally, NetBeans IDE stores resource references using absolute paths in the `nbproject/private/private.properties` file. You should never add the `/private` folder to the CVS repository, as it defines local properties that are different on each machine. When a project that has resource references defined in this way is put into CVS, each user who checks the project out must specify the location of the references on his or her machine.

If, however, you create a project in an existing CVS working directory, the IDE automatically uses relative links to all resources in the same CVS working directory. These relative links are defined in `project.properties`, which is checked into CVS and is shared by all users. As long as the projects and resources have the same relative location in the repository, no further

configuration is needed.



The IDE automatically excludes the `nbproject/private` folder, as well as the `build` and `dist` folders, from all CVS commands. If your project already has references defined with absolute paths defined in `private.properties` and you want to check the project into CVS, you can manually delete the references in `private.properties` and define them with relative paths in `project.properties`.

The recommended method of creating projects that are going to be checked into CVS is therefore as follows:

- 1.** Create a repository if it does not already exist.
- 2.** Check out the root of the repository by leaving the Modules to Check Out field empty in the CVS Checkout wizard.
- 3.** Create the projects in the working directory to which you checked out the repository root. The IDE automatically schedules all the new files for addition to the repository.
- 4.** Right-click each of the projects and choose CVS | Commit.

If you are only checking in one project or do not want to check out the entire root of your repository (which can sometimes contain too many files to manage effectively), you can right-click an individual project and choose CVS | Import into CVS. The IDE imports the project and then checks it out again, turning its location into a CVS working directory.

Resolving Merge Conflicts in Project Metadata Files

If your project's properties are modified by two or more developers simultaneously, merge conflicts can occur when you update your project from your version control system.

If merge conflicts occur in `project.xml` or `project.properties`, you should be able to resolve them manually in the Source Editor.

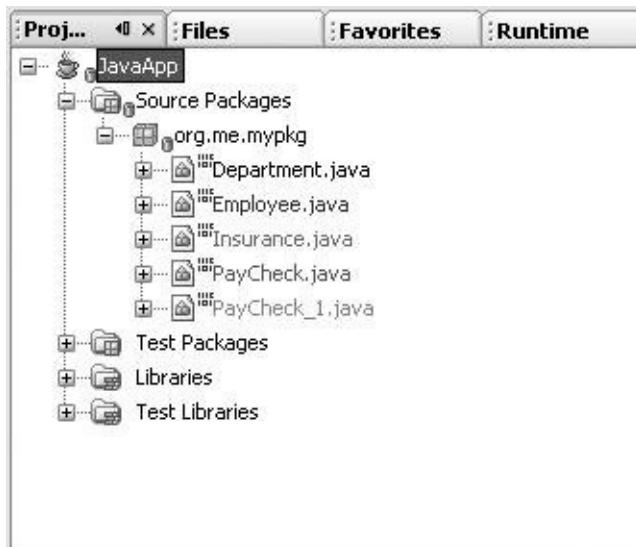
Should a merge conflict occur in your `build-impl.xml` file (which itself would probably be a result of a merge conflict in the `project.xml` file), do the following:

1. Resolve the merge conflict in the `project.xml` file and commit the change.
2. Delete the `build-impl.xml` file.
3. In the Projects window, right-click the project's main node and choose Close Project.
4. Reopen the project by choosing New | Open Project. The `build-impl.xml` file will be regenerated when you reopen the project.

Keeping Track of Changes

NetBeans IDE shows you the status of your files right in the UI using color coding and badges. A file displays its CVS status wherever it appears in the NetBeans UI, whether it is in the Projects window, as shown in [Figure 4-3](#), or in the Source Editor.

Figure 4-3. Files displaying their CVS status in the Projects window



The following colors are used to display file status:

- Green. Indicates that the file is a new local file that does not yet exist in the repository.
- Blue. Indicates that the file has been modified locally.

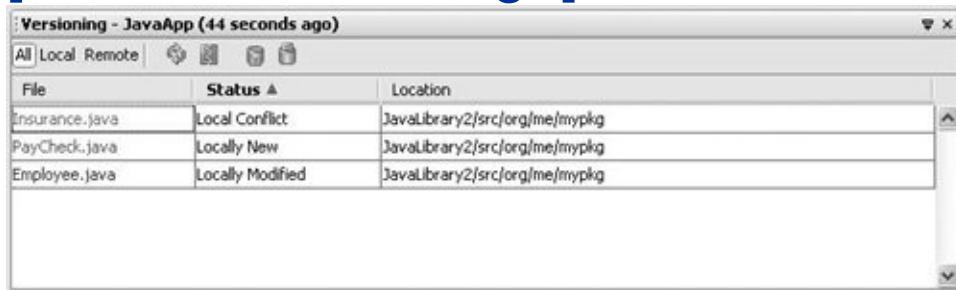
- Red. Indicates that the file contains conflicts.
- Gray. Indicates that the file is ignored by CVS and will not be included when calling versioning commands.

The IDE marks folder, package, and project nodes with a badge when these nodes contain files that are not up to date. A blue CVS badge means the node contains locally modified files. A red CVS badge means the node contains files with merge conflicts.

To see all the changes in your working directory, right-click any project, folder, or package node in any of the NetBeans navigation windows and choose CVS | Show Changes. The Versioning window, shown in [Figure 4-4](#), appears.

Figure 4-4. Versioning window

[\[View full size image\]](#)



The Versioning window automatically updates to reflect local changes in the location it is monitoring. From the Versioning window, you can:

- Diff files. Double-click any file to generate a diff for the file, or click the Diff All button to view all differences.

- Exclude any file from a commit. Right-click the file and choose Exclude from Commit. The file is displayed with its name in strikethrough.
- Update files. Click the Update All button .
- Commit changes to the repository. Click the Commit All  button.
- Run CVS commands on individual files. You can right-click any file to access the same CVS commands that are available in the Projects, Files, and Favorites windows.



When you use the Commit All and Update Add buttons, the IDE runs the commands on the project or folder that is currently being monitored in the Versioning window. The window's title bar displays the location currently being monitored.

Viewing Diffs

A diff compares the version of a file in your working directory to the version in the repository and displays any differences. NetBeans IDE provides a graphical diff viewer that makes reviewing changes much easier than working with traditional command-line diffs.



By default the IDE ignores whitespace in your diffs. You can turn this setting off by choosing Tools | Options, clicking Advanced Options, and selecting IDE Configuration | Server and External Tool Settings | Diff

and Merge Types | Built-in Diff Engine.

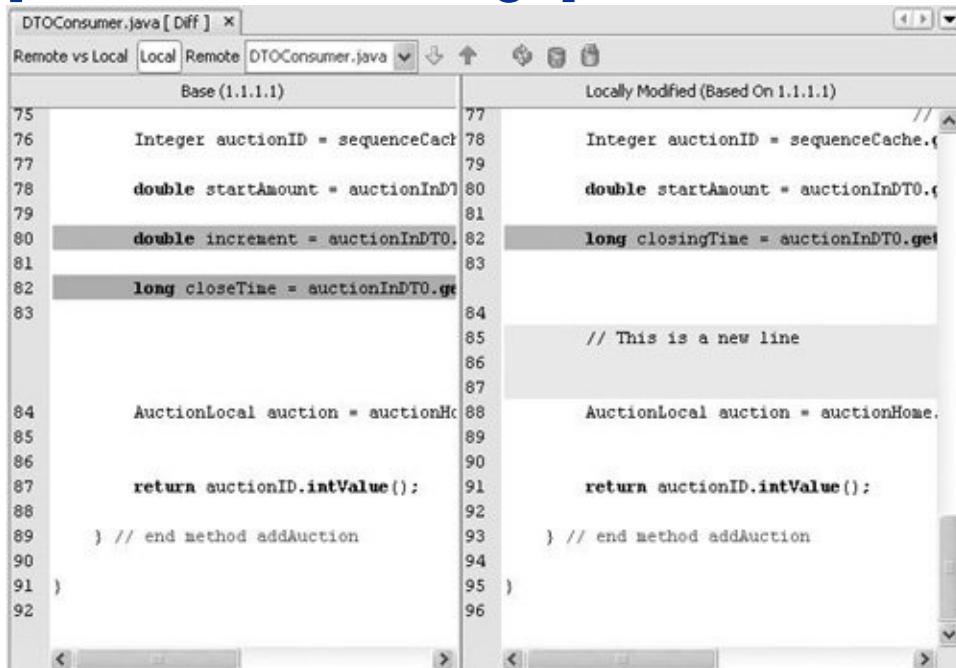
To view the differences between your local copy of the file and the base revision in the repository, do one of the following:

- Right-click any file, folder, package, or project node and choose CVS | Diff.
- In the Versioning window, double-click any individual file or click the Diff All button .

The Diff Viewer, shown in [Figure 4-5](#), displays one file at a time.

Figure 4-5. Diff Viewer

[\[View full size image\]](#)



The screenshot shows a Java code editor interface titled "DTOConsumer.java [Diff]". The tabs at the top are "Remote vs Local", "Local", "Remote", and "DTOConsumer.java". The main area displays two columns of code side-by-side, representing the "Base (1.1.1.1)" and "Locally Modified (Based On 1.1.1.1)" versions. The code compares the addition of a closing time variable and its assignment from a DTO. A tooltip "This is a new line" appears over the new line added in the modified version. Line numbers are visible on the left.

```
75      Integer auctionID = sequenceCache.  
76      double startAmount = auctionInDB.  
77      double increment = auctionInDTO.  
78      long closeTime = auctionInDTO.get  
79      AuctionLocal auction = auctionHome.  
80      return auctionID.intValue();  
81  } // end method addAuction  
82 }  
83  
84      Integer auctionID = sequenceCache.  
85      double startAmount = auctionInDB.  
86      double increment = auctionInDTO.  
87      long closingTime = auctionInDTO.get  
88      AuctionLocal auction = auctionHome.  
89      return auctionID.intValue();  
90  } // end method addAuction  
91 }  
92 }
```

You can use the Next Difference and Previous Difference buttons to navigate through the differences. When you reach the last difference in a file, the Next Difference button takes you to the next file. You can also switch through files using the drop-down list in the Diff Viewer toolbar.

The Diff Viewer uses the following colors to display differences:

- Blue areas mark existing lines that have been changed.
- Green areas mark new lines that have been added.
- Red areas mark lines that have been removed.

In addition to viewing the differences between your local version and the latest revision in the repository, you often have to diff files against earlier revisions, tags, or difference branches. To do so:

1. Right-click a file or folder and choose CVS | Search History.
2. In the From field, specify the beginning of the diff. If you just want to diff your current working directory against an earlier revision, leave this field empty. If you want to diff two revisions in the repository, enter a tag, branch name, or date in this field.
3. In the To field, enter the tag, branch, or date you want to diff the files against. You can also specify other search criteria. See [Working with Versioning History](#) for more information.
4. Click Search. Each of the files containing differences is listed in the window.

5. Click the Diff button and select any file to view its diff in the bottom panel of the Search History window.

Updating Files

Updating files merges your local version of the files with the latest version in the repository. To update files, do any of the following:

- Choose CVS | Update All to update all files in all open projects. Note that files in the Favorites window are not updated.
- Right-click any node in the Projects, Favorites, or Files window and choose CVS | Update. The node and all its subnodes are updated.
- Click the Update All button in the Versioning window to update all the files in the location currently being monitored by the Versioning window. The location is displayed in the title bar of the Versioning window.
- Select a project in the Projects window and choose CVS | Update Project with Dependencies. The project and any projects on its classpath are updated.

Committing Changes

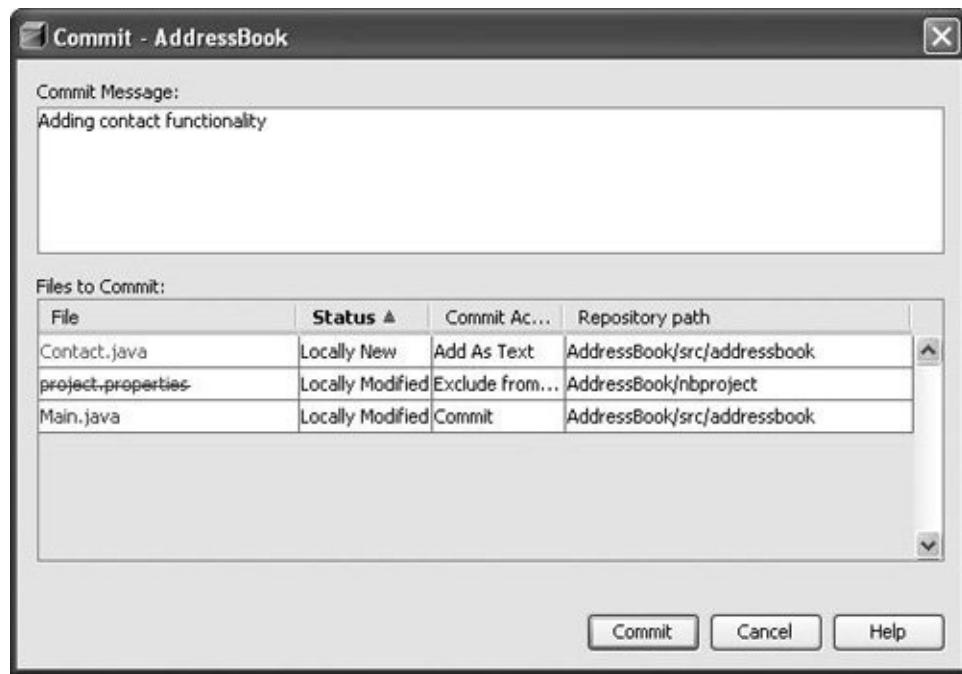
When you are ready to put your changed files back into the repository, you use the Commit command. To commit files, do one of the following:

- Choose CVS | Commit All from the main menu. The IDE commits all changes in all open projects. Note that files in the Favorites window are not committed.
- Click the Commit All button in the Versioning window to commit all the files in the location currently being monitored by the Versioning window. The location is displayed in the title bar of the Versioning window.
- Right-click any file, folder, or project node in the IDE and choose CVS | Commit.

In the Commit dialog box, shown in [Figure 4-6](#), specify a message for the commit and review that each file has the correct action. You can exclude any file from the commit in this dialog box.

Figure 4-6. Commit dialog box

[\[View full size image\]](#)



Ignoring Files in CVS Operations

Not every file that you have in a working directory needs to go into the CVS repository. You often have place-holder files, copies of files, and files that are not finished yet in your working directory. In NetBeans IDE, you can easily specify which files CVS should ignore. Since the IDE automatically schedules any new files for addition to the repository, it is important that you manage this function carefully.

You have two choices:

- Specify that a file should be excluded from CVS commits. This option is best for files that will eventually end up in the repository or are already in the repository, but you do not want to commit yet. To exclude a file from commits, right-click it in the Versioning window and choose Exclude from Commit. You can also exclude a file in the Commit dialog box.
- Specify that a file should be ignored altogether in CVS commands. This option is best for files that should never be added to the repository. To ignore a file, right-click its node in any IDE window and choose CVS | Ignore.



When you choose to ignore a file, the IDE adds the file name to the containing folder's `.cvsignore` file. This results in a locally modified or new `.cvsignore` file in the folder, which is in turn scheduled to be committed to the repository. You should only commit the `.cvsignore` file if it is likely that other developers will also have the same file in the same location.

Adding and Removing Files from a Repository

The IDE automatically schedules any new files in a CVS working directory for addition to the repository. In addition, the IDE scans the file and suggests whether to add it as text or as a binary file. Likewise, any file you delete in the IDE is automatically scheduled for removal from the repository.

This differs from the way CVS is commonly used on the command line and can potentially get you into trouble. For example, if you want to get a clean copy of your file from the repository, a common practice on the command line is to simply delete the file and update it. In the IDE, however, this deletes the file and schedules it for removal from the repository. In this case, even updating the containing folder does not get you a clean copy of the file.

If you want to get a clean copy of a file from the repository, right-click the file and choose CVS | Revert Modifications. If you accidentally deleted a file and would like to get a clean copy of the file from the repository, open the file's folder in the Versioning window, right-click the file in the Versioning window, and choose Revert Modifications.

Likewise, if you added a new file and do not want to commit it to the repository yet, choose to ignore the file completely or just to exclude it from commits as described above.

Working with Branches

Two useful functions when working with CVS files are the ability to tag and branch the repository. Tagging a repository provides a snapshot of the repository that you can use to later check out or diff against. For example, you could tag the repository before committing a large and potentially dangerous change. If you need to roll back the change, you can just use the tag with the update command to get back to the state before the commit.

Branching a repository makes a separate line of development that you can work on without merging it into the main trunk of the repository. For example, you may need to create a bug-fix branch for the current release while also continuing work on the next release in the trunk.

Creating a Tag

To create a tag in the repository:

- 1.** Right-click the project or folder you want to tag and choose CVS | Tag.
- 2.** Specify a name for the branch.
- 3.** If you want to check that you have committed all local modifications before you create the tag, select the Avoid Tagging Locally Modified Files checkbox.
- 4.** If the tag already exists, but is set on an earlier revision of the files you are tagging, select the Move Existing Tag checkbox. The IDE removes the tag from the earlier revision and assigns it to the revision you are tagging.
- 5.** Click Tag.

Once you have tagged the repository, you can update or check out a repository from the point where the tag was made by entering the tag name in the Check Out command dialog box.

Creating a Branch

A branch is created using "sticky tags." When you switch to a branch, your local files are tagged with the branch's sticky tags. Any commands that you run on the files, like update and commit, only work with the branch. For example, if you update the files, you only get changes that were made on the branch, not in the trunk.

To create a new branch in the repository:

1. Right-click the project, folder, or file you want to branch and choose CVS | Create Branch.
2. Enter the branch name. Make sure the branch name does not already exist in the repository. You can use the Browse button to review the existing branch names.



Do not use the Branch command to switch to an existing branch. Instead, use the Switch to Branch command.

3. Specify whether to tag the files before creating the branch and the name of the tag. Tagging the files before branching makes it easy to get a snapshot of the files before the branch was made.
4. Specify whether you want your files to be switched to the

branch or whether you just want to create the branch but continue working in the trunk or current branch.

Viewing a Working Directory's Branch

By default, the IDE only displays which branch a working directory is on in the title bar of the Versioning window. You can turn on the display of branch information in the IDE, so that each file displays its branch name in the IDE windows and Source Editor tabs. To display branch information, choose View | Show CVS Status Labels.

If no branch name is displayed, the working directory files are from the repository trunk.

Switching Between Branches

You should always commit your local changes to your working directory before switching to a branch, or your local changes could be lost.

To switch to a branch:

1. Right-click the project or folder that you want to switch to a branch and choose CVS | Switch to Branch.
2. Choose whether to switch to the trunk or to a branch. You can click Browse to see all of the available branches. Do not enter a tag name in the Switch to Branch dialog box.
3. Click Switch. The IDE updates the repository to the branch. Any files that exist on the branch to which you are switching are created. Any files that do not exist in the branch you are switching to are deleted.

Merging from a Branch

Merging from a branch can be tricky. You can often get multiple merge conflicts in files that have been modified in both the trunk and the branch. Here are some hints that can help:

- Always commit all of your changes before doing the merge. Do not have any locally modified files in your working directory.
- Always tag your branch after a merge, and then use that tag as the starting point for your next merge. For example, if you create a branch and one week later do a merge to synch the branch with the trunk, you usually have to resolve a few merge conflicts. If you then merge from the same point a week later, you have to resolve the same merge conflicts all over again. You should therefore only merge changes that were made after the point of your last merge.
- If you just need to integrate one fix and do not want to merge an entire branch back, you can use the Search History command to find the commit that included the fix. You can then export that commit as a patch and apply the patch to the trunk.

To merge the trunk from a branch:

1. If you do not have the trunk checked out, check out the trunk and open the working directory in the IDE, either by opening the working directory in the Favorites window or opening the projects.
2. Right-click the project or folder that you want to merge from a branch and choose CVS | Merge from Branch.

- 3.** Select Merge from Branch and enter the name of the branch you are merging from.
- 4.** If necessary, choose to only merge changes from a specific tag.
- 5.** If necessary, choose to tag the trunk or branch after the merge is complete. Doing so gives you an easy starting point for the next merge.

Working with Patches

A patch file is basically a saved diff of one or more files that can then be applied to another working directory, thus recreating the changes in the second working directory. Patches are useful in the following situations:

- When you want to let other developers preview a change before you integrate it to the repository.
- When you want to share a change with someone who does not have access to the CVS repository.
- When you only want to integrate a single bug fix into the repository and do not want to merge an entire branch over.

Creating a Patch

To create a patch of the differences between your local working directory and the latest repository version:

1. Select the project, folder, or file from which you want to create the patch.
2. Choose CVS | Export "*file name*" Diff Patch.
3. Specify a name and location for the patch file and click OK.



You can also grab all of the changes to the repository that were committed in a single commit and save those changes as a patch. This is handy when you want to attach a single bug fix that you've done on a private branch to a bug report before you merge it into the trunk. To save a commit as a patch, find the commit as

described in Working with Versioning Histories below, and then choose CVS | Export "file name [Search History]" Diff Patch.

Applying a Patch

When another developer applies your patch, he or she recreates the changes in the patch in his or her own working directory. The effect is the same as if you had committed your changes and the other developer had updated his or her working directory. You must apply the patch in exactly the same location in the working directory as where the patch was created from.

To apply a patch:

- 1.** Make sure there are no locally modified files in your working directory.
- 2.** If you do not know where the patch was created from, open the patch in a text editor. The second line displays the folder to which the paths are relative.
- 3.** Right-click the correct location and choose Tools | Apply Diff Patch.
- 4.** Specify the location of the patch file and choose OK.

Working with Versioning Histories

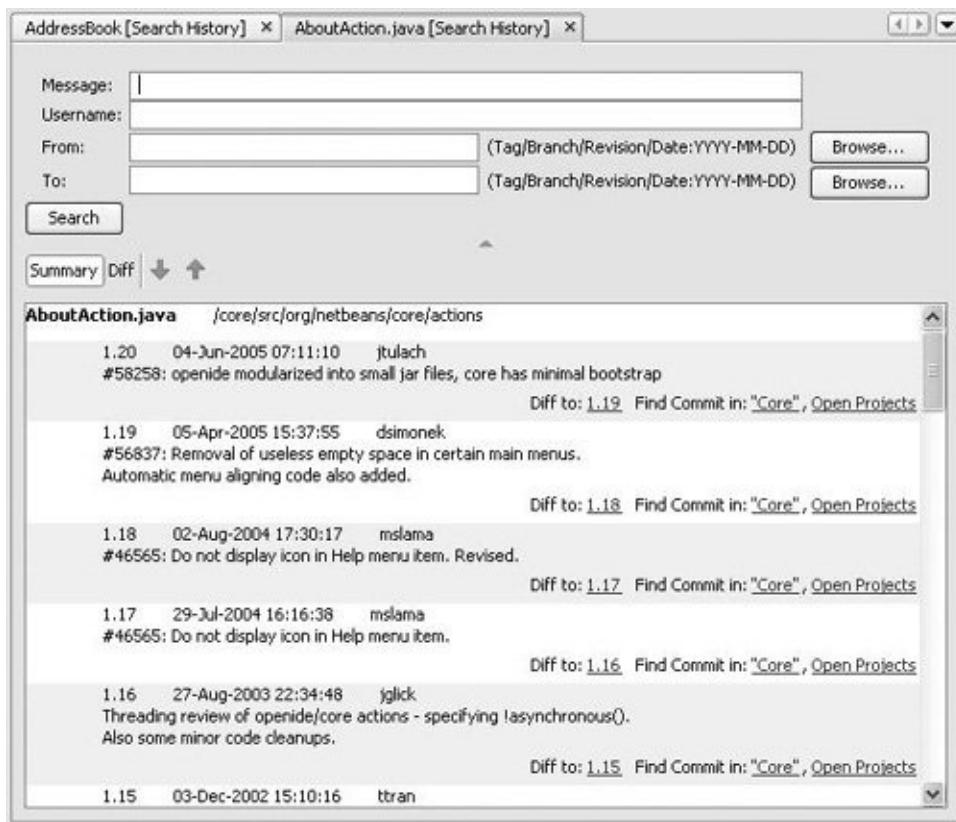
When working with a long-running development project, managing long versioning histories and multiple branches can get very tricky. You often have to roll back changes that introduce bugs or have to merge a specific set of changes from one branch to another. The IDE's Search History window gives you a powerful tool for viewing and searching through branches and past revisions of your files.

Viewing the History of a File

To view the full history for a file, right-click the file's node and choose CVS | Search History. The Search History window, shown in [Figure 4-7](#), displays all of the file's revisions. You can click the Diff button to view diffs of the various revisions. Select any one revision to diff it against the copy in your working directory. Select any two revisions to see the differences between the two revisions.

Figure 4-7. Search History window

[[View full size image](#)]



Searching for a Commit

When you need to roll back a change or create a patch from a commit, it is necessary to find all the files that were included in the commit. Doing this is fairly difficult from the command line, since regular CVS has no way to search for a particular commit message. The Search History window makes the process of finding an entire commit painless.

To find all the files that were included in a commit:

1. Make sure that all the projects on which you ran the commit command are open in the IDE.
2. Right-click any file that you know was included in the

commit and choose Search History. All of the file's revisions are displayed in the Summary view.

3. Find the revision that was created by the commit and click Find Commit in Open Projects.
4. The IDE opens a second Search History window listing all the files included in the commit. You can switch to the Diff view to see the differences of each file, or choose CVS | Export *Search Details* Diff Patch to capture the commit changes in a patch.

Reversing Changes in a File

If you have made changes to a file that you would like to reverse, you can easily do so:

- If the changes you would like to reverse have not yet been committed to the repository, you can use the Revert Modifications command to overwrite your local copy of the file with the current repository version of the file.
- If you want to reverse changes to a file that you have already committed to the repository, you can use the Roll Back command to reinstate a previous version.

To revert local modifications, right-click the file and choose CVS | Revert Modifications.

To revert changes that have already been committed to the repository:

1. Right-click the file and choose CVS | Search History. All of the file's revisions are displayed in the Summary view.

- 2.** Find the revision to which you would like to revert. Within the Summary view, you can display diffs to inspect the differences between the various revisions.
- 3.** Right-click the revision to which you would like to revert and choose Rollback to *RevisionNumber*.

After you run this command, your local copy is reverted to the previous revision, but the repository copy is not changed.

- 4.** Right-click the file's node and choose CVS | Commit.

The reversal of the changes is committed to the repository (and the revision number is incremented upwards).

Working with Other Version Control Systems

Although earlier versions of the IDE provided support for other version control systems, such as Microsoft Visual SourceSafe and PVCS, there was not enough time to convert all of the version control systems to the new system of projects-aware CVS. However, there are plans to migrate other version control systems to a similar interface and developer workflow as that which was implemented for CVS in NetBeans IDE 5.0.

If you need to work with other version control systems, you can download and install the old Generic VCS modules that support these systems from the NetBeans Update Center. You can access the Update Center by choosing Tools | Update Center.

Note that enabling these modules automatically disables the new CVS system in the IDE. If you also need to work with CVS, you should also download and install the CVS profile for Generic VCS.

[Table 4-1](#) lists the version control systems that are available and the update centers where they are located as of this writing.

Table 4-1. Additional Generic VCS Profiles and Their Locations

VCS	Location
Visual SourceSafe	NetBeans Update Center
CVS	NetBeans Update Center
PVCS	NetBeans Update Center Beta
Subversion	NetBeans Update Center Beta

Once you have installed any of these modules, help topics become available describing how to use the Generic VCS style of version control support. Choose Help | Help Contents to access these topics.

Chapter 5. Editing and Refactoring Code

- [Opening the Source Editor](#)
- [Managing Automatic Insertion of Closing Characters](#)
- [Displaying Line Numbers](#)
- [Inserting Snippets from Code Templates](#)
- [Using Editor Hints to Generate Missing Code](#)
- [Matching Other Words in a File](#)
- [Generating Methods to Implement and Override](#)
- [Creating and Using Macros](#)
- [Creating and Customizing File Templates](#)
- [Handling Imports](#)
- [Displaying Javadoc Documentation While Editing](#)
- [Formatting Code](#)
- [Text Selection Shortcuts](#)
- [Navigating within the Current Java File](#)
- [Navigating from the Source Editor](#)

- [Searching and Replacing](#)
- [Refactoring Commands](#)
- [Deleting Code Safely](#)
- [Extracting a Superclass to Consolidate Common Methods](#)
- [Changing References to Use a Supertype](#)
- [Unnesting Classes](#)
- [Tracking Notes to Yourself in Your Code](#)
- [Comparing Differences Between Two Files](#)
- [Splitting the Source Editor](#)
- [Maximizing Space for the Source Editor](#)
- [Changing Source Editor Keyboard Shortcuts](#)

NETBEANS IDE PROVIDES A WIDE VARIETY OF TOOLS to support Java application development, but it is the Source Editor where you will spend most of your time. Given that fact, a lot of attention has been put into features and subtle touches to make coding faster and more pleasurable.

Code completion and other code generation features help you identify code elements to use and then generate code for you. Refactoring features enable you to easily make complex changes to the structure of your code and have those changes propagated throughout your project. Keyboard shortcuts for these code generation features and for file navigation ensure that your hands rarely have to leave the keyboard.

Architecturally, the Source Editor is a collection of different types of editors, each of which contains features specific to certain kinds of files. For example, when you open a Java file, there is a syntax highlighting scheme specifically for Java files, along with code completion, refactoring, and other features specific to Java files. Likewise, when you open JSP, HTML, XML, `.properties`, deployment descriptor, and other types of files, you get a set of features specific to those files.

Perhaps most importantly, the Source Editor is tightly integrated with other parts of the IDE, which greatly streamlines your workflow. For example, you can specify breakpoints directly in the Source Editor and trace code as it executes. When compilation errors are reported in the Output window, you can jump to the source of those errors by double-clicking the error or pressing F12.

In this chapter, we will demonstrate the ways you can use the IDE's editing features to simplify and speed common coding tasks.

Opening the Source Editor

Before starting to work in the Source Editor, you will typically want to have an IDE project set up. You can then open an existing file or create a new file from a template. See [Chapter 3](#) for basic information on creating projects and files and for a description of the various file templates.

If you would simply like to create a file without setting up a project, you can use the Favorites window. The Favorites window enables you to make arbitrary folders and files on your system accessible through the IDE. The Favorites window is not designed for full-scale project development, but it can be useful if you just want to open and edit a few files quickly.

To use the Source Editor without creating a project:

- 1.** Choose Window | Favorites to open the Favorites window.
- 2.** Add the folder where you want the file to live (or where it already lives) by right-clicking in the Favorites window, choosing Add to Favorites, and choosing the folder from the file chooser.
- 3.** In the Favorites window, navigate to the file that you want to edit and doubleclick it to open it in the Source Editor.

If you want to create a new file, right-click a folder node, choose New | Empty File, and enter a filename (including extension).

Managing Automatic Insertion of Closing Characters

When typing in the Source Editor, one of the first things that you will notice is that the closing characters are automatically inserted when you type the opening character. For example, if you type a quote mark, the closing quote mark is inserted at the end of the line. Likewise, parentheses(), brackets ([]), and curly braces ({}) are completed for you.

While this might seem annoying at first, the feature was designed to not get in your way. If you type the closing character yourself, the automatically inserted character is overwritten. Also, you can end a line by typing a semicolon (;) to finish a statement. The semicolon is inserted at the end of the line after the automatically generated character or characters.

See the following subtopics for information on how to use the insertion of matching closing characters.

Finishing a Statement

When the Source Editor inserts matching characters at the end of the line, this would appear to force you to move the insertion point manually past the closing character before you can type the semicolon. In fact, you can just type the semicolon without moving the insertion point, and it will be placed at the end of the line automatically.

For example, to get the line

```
ArrayList ls = new ArrayList();
```

you would only have to type

```
ArrayList ls = new ArrayList();
```

Splitting a String Between Two Lines

If you have a long string that you want to split between two lines, the Source Editor adds the syntax for concatenating the string when you press Enter.

For example, to get the lines

```
String s = "Though typing can seem tedious, reading lo  
"and convoluted sentences can be even worse."
```

you could type

```
String s = "Though typing can seem tedious, reading lo  
and convoluted sentences can be even worse.
```

The final three quote marks and the plus sign (+) are added for you.

If you want to break the line without creating the concatenation, press Shift-Enter.

Displaying Line Numbers

By default, line numbers are switched off in the Source Editor to save space and reduce visual clutter. If you need the line numbers, you can turn them on by choosing View | Show Line Numbers. You can also right-click in the left margin of the Source Editor and choose Show Line Numbers.

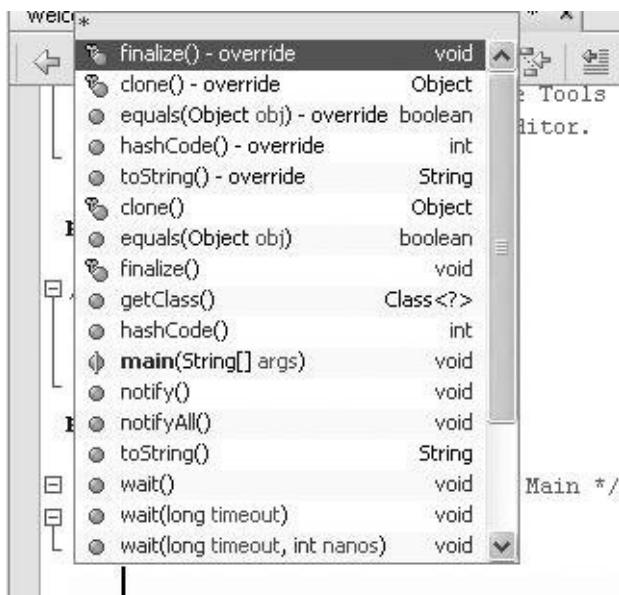
Generating Code Snippets without Leaving the Keyboard

The Source Editor has several features for reducing the keystrokes needed for typing code. And you can access many of these features without using the mouse, having to use menus, or remembering scores of keyboard shortcuts.

Arguably the most important mechanisms for generating code are the following:

- Ctrl-spacebar keyboard shortcut. This shortcut opens the code completion box, as shown in [Figure 5-1](#). The code completion box contains a context-sensitive list of ways you can complete the statement you are currently typing and of other code snippets you might want to insert in your code.

Figure 5-1. Code completion box



- Multi-keystroke abbreviations for longer snippets of code called code templates. These abbreviations are expanded into the full code snippet after you press the spacebar.
- Alt-Enter keyboard shortcut. You can use this shortcut to display suggestions the IDE has regarding missing code and then have the IDE insert that code. The IDE notifies you that it has a suggestion by displaying a lightbulb () icon in the left margin of the line you are typing.

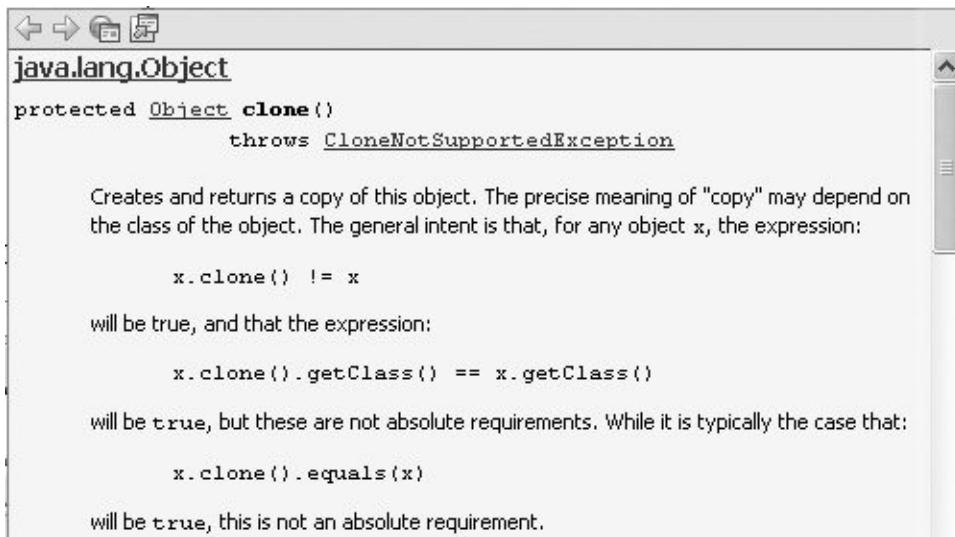
In addition to saving keystrokes and use of the mouse, these features might prevent typos and also help you find the right class and method names.

The following several sections illustrate how to get the most out of these features. These topics concentrate on features for Java files, but many of the features (such as code completion and word matching) are also available for other types of files, such as JSP and HTML files.

Using Code Completion

When you are typing Java identifiers in the Source Editor, you can use the IDE's code completion box to help you finish expressions, as shown in [Figure 5-1](#). In addition, a box with Javadoc documentation appears (as shown in [Figure 5-2](#)) and displays documentation for the currently selected item in the code completion box.

Figure 5-2. Javadoc box that accompanies the code completion box



Beginning with NetBeans IDE 5.0, many types of code generation have been added to the code completion box. Using the code completion box, you can:

- Fill in names of classes and class members. (After you select a class to fill in, an import statement is also filled in, if appropriate.)

- Browse Javadoc documentation of available classes.
- Generate whole snippets of code from dynamic code templates. You can customize code templates and create new ones. See [Inserting Snippets from Code Templates](#) for more information.
- Generate getter and setter methods.
- Generate skeletons for abstract methods of classes extended by and interfaces implemented by the current class.
- Override inherited methods.
- Generate skeletons of anonymous inner classes.

To open the code completion box, do one of the following:

- Type the first few characters of an expression and then press Ctrl-spacebar (or Ctrl-\).
- Pause after typing a period (.) in an expression.
- Type a space, and then pause for a moment.

The code completion box opens with a selection of possible matches for what you have typed so far.

To narrow the selection in the code completion box, continue typing the expression.

To complete the expression and close the code completion box, do one of the following:

- Continue typing until there is only one option left and then press Enter.
- Scroll through the list, using the arrow keys or your mouse to select a value, and then press Enter.

To close the code completion box without entering any selection, press Esc.

To complete the expression and leave the code completion box open, select a completion and press the period (.) key. This is useful if you are chaining methods. For example, if you want to type

```
getRootPane().setDefaultButton(defaultButtonName)
```

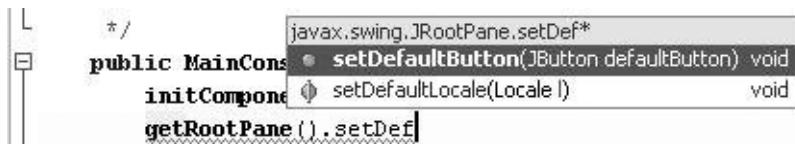
you might do the following:

1. Type `geTRo` (which would leave only `getRootPane()` in the code completion box) and press the period (.) key.
2. Type `.setDef` (which should make `setDefaultButton(JButton defaultButton)` the selected method in the code completion box, as shown in [Figure 5-3](#)) and press Enter.

`getRootPane().setDefaultButton()` should now be inserted in your code with the insertion point placed between the final parentheses. A tooltip appears with information on the type of parameter for you to enter.

Figure 5-3. Code completion box with `setDefaultButton (JButton defaultButton)`

selected



3. Type a name for the parameter.
4. Type a semicolon (;) to finish the statement. The semicolon is automatically placed after the final parenthesis.

Code Completion Tricks

When typing with the code completion box open, there are a few tricks you can use to more quickly narrow the selection and generate the code you are looking for. For example:

- You can use "camel case" when typing class names. For example, if you want to create an instance of `HashSet`, you can type `private HS` and press Ctrl+Spacebar to display `HashSet` (and other classes that have a capital H and a capital S in their names).
- You can use the comma (,) and semicolon (;) keys to insert the highlighted item from the code completion box into your code. The comma or semicolon is inserted into your code after the chosen item from the code completion box.
- You can fill in text that is common to all of the remaining choices in the list by pressing Tab. This can save you several keystrokes (or use of the arrow keys or mouse)

when the selection in the code completion box is narrowed to choices with the same prefix. For example, if you are working with a `Hashtable` object `ht`, and you have typed `ht.n`, there will be two methods beginning with `notify` (`notify()`) and `notifyAll()`). To more quickly narrow the selection to just `notifyAll()`, press Tab to expand `ht.n` to `ht.notify` and then type `A`. You can then press Enter to complete the statement with `notifyAll()`.

Disabling Automatic Appearance of the Java Code Completion Box

If you find the code completion box to be more of a nuisance than a help, you can disable automatic appearance of the code completion popup. Code completion will still work if you manually activate it by pressing Ctrl-spacebar or Ctrl-\.

You can also leave automatic appearance of the code completion popup enabled but disable the bulkier Javadoc code completion dialog box. The Javadoc popup can be manually invoked with Ctrl-Shift-spacebar.

To disable automatic appearance of the code completion box:

1. Choose Tools | Options, click Editor in the left panel, and select the General tab.
2. Deselect the Auto Popup Completion Window property and click OK.

To disable automatic appearance of the Javadoc popup when you use the code completion box:

1. Choose Tools | Options, click Advanced Options, expand the Editing | Editor Settings node, and select the Java Editor node.

2. In the Expert settings, deselect the Auto Popup Javadoc Window property.



You can also merely adjust the amount of time that elapses before the code completion box appears. To do so, choose Tools | Options, click Advanced Options, expand the Editing | Editor Settings node, and select the Java Editor node. In the Delay of Completion Window Auto Popup property, enter a new value (in milliseconds). By default, the delay is 250 milliseconds.

Changing Shortcuts for Code Completion

If you prefer to use different shortcuts for code completion, you can change those shortcuts in NetBeans IDE:

1. Choose Tools | Options, click Editor in the left panel, and select the Keymap tab.
2. Expand the Other folder and select the command for which you want to change the shortcut.

Commands that apply to code completion are Show Code Completion Popup, Show Code Completion Tip Popup, and Show Documentation Popup.

3. Click Add, and type the shortcut that you want to use.

You can remove an already assigned keyboard shortcut by selecting the shortcut and clicking Remove.

Inserting Snippets from Code Templates

As you are typing in the Source Editor, you can use code templates to greatly speed up the entry of commonly used sequences of reserved words and common code patterns, such as `for` loops and field declarations. The IDE comes with a set of templates, and you can create your own.

Some code templates are composed of segments of commonly used code, such as `private static final int`. Others are more dynamic, generating a skeleton and then letting you easily tab through them to fill in the variable text (without having to use the mouse or arrow keys to move the cursor from blank to blank). Where a code snippet repeats an identifier (such as an Iterator object, as shown in [Figure 5-4](#)), you just have to type the identifier name once.

Figure 5-4. Code template with variable text to be inserted

```
for (Iterator it = collection.iterator(); it.hasNext();) {  
    Object elem = (Object) it.next();  
}
```

Here are a few examples:

- You can use the `newo` template to quickly create a new object instance. You type `newo` and a space, the IDE generates `Object name = new Object(args);` and highlights the two occurrences of `Object`. You can then type a class name and press Tab. Both occurrences of `Object` are changed to the class name

and then `args` is selected. You can then fill in the parameters and press Enter to place the insertion point at the end of the inserted code.

You can use Shift-Tab to move backward through the parameters. You can press Enter at any time to skip any parameters and jump straight to the end of the template (or where it is specified that the cursor should rest after the template's parameters are filled in).

- You can use the `fori` template to create a loop for manipulating all of the elements in an array. Initially, the IDE generates the following:

```
for (int i = 0; i < arr.length; i++) {  
}
```

The index is automatically given a name that is unique within the current scope (defaulting to `i`). You can manually change that value (causing the IDE to change the value in all three places) or directly tab to `arr`, to type the array name. If an array is in scope, the IDE will use its name by default. The next time you press Tab, the cursor lands on the next line, where you can type the array processing code.

- You can use the `forc` template to create a skeleton `for` loop that uses an `Iterator` object to iterate over a collection as shown in [Figure 5-4](#).

This code template has the additional benefit of generating an import statement for `Iterator`.

You can access code templates in either of the following ways:

- Typing the first few letters of the code, pressing Ctrl-

spacebar, and then selecting the template from the list in the code completion box. In the code completion box, templates are indicated with the  icon, as shown in [Figure 5-5](#). The full text of the template is shown in the Javadoc box.

Figure 5-5. Code completion box and Javadoc box, with a code template selected



- Typing the abbreviation for the code template directly in the Source Editor and then pressing the spacebar. You can find the abbreviations for the builtin Java code templates in [Table 5-1](#). If you discover a code template in the code completion box, the abbreviation for that template is in the right column of that abbreviation's listing.

Table 5-1. Java Code Templates in the Source Editor

Abbreviation Expands To

Abbreviation	Expansion
ab	abstract

```
bo          boolean

br          break

ca          catch (

cl          class

cn          continue

df          default:

dowhile    do {
            ${cursor}
} while(${condition});

En          Enumeration

eq          equals

Ex          Exception

ex          extends

fa          false

fi          final

fy          finally

fl          float

forc        [View full width]
for (Iterator it = collection.iterator();
    .hasNext();) {
    Object elem = (Object) it.next();
}

fore       [View full width]
```

```
for (Iterator it = collection.iterator();
    .hasNext();) {
    Object elem = (Object) it.next();
}

for (int i = 0; i < ${arr array}.length; i =
${cursor}
}

ie interface

ifelse if (${condition}) {
    ${cursor}
} else {
}

im implements

iof instanceof

ir import

le length

newo Object name = new Object(args);

Ob Object

pst printStackTrace();

pr private

psf private static final

psfb private static final boolean

psfi private static final int

psfs private static final String
```

```
pe          protected

pu          public

Psf         public static final

Psfb        public static final boolean

Psfi        public static final int

Psfs        public static final String

psvm        public static void main(String[] args) {
            ${cursor}
}

re          return

st          static

St          String

serr        System.err.println("${cursor}");

sout        System.out.println("${cursor}");

sw          switch {

sy          synchronized

tds         Thread.dumpStack();

tw          throw

twn         throw new

th          throws

trycatch    try {
            ${cursor}
```

```
        } catch (Exception e) {  
    }  
  
wh      while (  
  
whilei    while (it.hasNext()) {  
        Object elem = (Object) it.next();  
        ${cursor}  
    }  
_____
```

If an abbreviation is the same as the text that you want to type (for example, you do not want it to be expanded into something else), press Shift-spacebar to keep it from expanding.

See [Table 5-1](#) for a list of code templates (and their abbreviations) for Java files. The IDE also comes with sets of abbreviations for JSP files (see [Chapter 8, Using Code Templates for JSP Files](#)), XML files, and DTD files. You can create your own abbreviations for these file types and for other file types as well (such as for HTML files, SQL files, etc.).

Adding, Changing, and Removing Code Templates

The code templates that come with the IDE are representative of the kind of things you can do with code templates, but they represent only a tiny fraction of the number of potentially useful templates.

You can modify existing code templates and create entirely new ones to suit the patterns that you use frequently in your code.

To create a new code template:

1. Choose Tools | Options, click Editor in the left panel, and select the Code Templates tab.
2. Click New.
3. In the New Code Template dialog box, type an abbreviation for the template and click OK.
4. In the Expanded Text field, insert the text for the template. See [Code Template Syntax](#) below for information on how to customize the behavior of your templates.
5. Click OK to save the template and exit the Options dialog box.

To modify a code template:

1. Choose Tools | Options, click Editor in the left panel, and select the Code Templates tab.
2. Select a template from the Templates table and edit its text in the Expanded Text field.
3. Click OK to save the changes and exit the Options dialog box.



In NetBeans IDE 5.0, you cannot directly change the abbreviation for a code template. If you want to assign a different shortcut to an existing template, select that shortcut, copy its expanded text, create a new code template with that text and a different abbreviation, and then remove the template with the undesired abbreviation.

To remove a code template:

1. Choose Tools | Options, click Editor in the left panel, and select the Code Templates tab.
2. Select a template from the Templates table and click Remove.
3. Click OK to save the changes and exit the Options dialog box.

Code Template Syntax

In code templates, you can set up the variable text to provide the following benefits for the template user:

- Display a descriptive hint for remaining text that needs to be typed in.
- Enable typing of an identifier once and have it generated in multiple places
- Make sure that an import statement is added for a class.
- Specify a type that a parameter of the code template is an instance of in order for the IDE to automatically generate an appropriate value for that parameter when the template is used to insert code.
- Automatically set up a variable name for an iterator, making sure that that variable name is not already used within the current scope.
- Set a location for the cursor to appear within the generated snippet once the static text has been generated and the variable text has been filled in.

For example, you might want to easily generate something like the following code for a class that you instantiate often:

```
FileWriter filewriter = new FileWriter(outputFile)
```

In the definition for such a code template, you could use something like the following:

```
 ${fw type = "java.io.FileWriter"  
    editable="false"} ${filewriter} = new ${fw}(${output
```

When the template is inserted into the code, the following things happen:

- `${fw type = "java.io.FileWriter" editable="false"}` is converted to `FileWriter` in the inserted code.
- `${fw}` is also converted to `Filewriter` (as it is essentially shorthand for the previously defined `${fw type = "java.io.FileWriter" editable="false"}`).
- `${filewriter}` and `${outputFile}` generate text (`filewriter` and `outputFile`, respectively).
- `filewriter` is selected. You can type a new name for the field and then press Tab to select `outputFile` and type a name for that parameter. Then you can press Tab or Enter to place the cursor after the whole generated snippet.

You could further refine the code template by defining an `instanceof` attribute for `${outputFile}` (e.g. `outputFile instanceof "java.io.File"`). This would enable the IDE to detect an instance

of that class and dynamically insert the name of the instance variable in the generated snippet instead of merely `outputFile`.

See [Table 5-2](#) for a description of examples of the syntax that you can use in the creation of code templates.

Table 5-2. Java Code Template Syntax

Syntax Element Example	Description
<code> \${}</code>	Used to enclose dynamic parts in the template.
<code> \${ElementName}</code>	Text you put within the braces will appear in the generated code snippet as highlighted text that you can type over. For example, if you use <code> \${Object}</code> , that would be a hint for the template user to type in an object name. If a given <code>ElementName</code> appears multiple times in the code template definition, you only have to replace the variable text once when you create a code snippet from that template.
<code> \${ElementName}type="FULLYQUALIFIED-CLASS-NAME" editable="false"</code>	Includes a class name as part of the inserted code and has the IDE automatically insert an import statement for the class, if necessary. Here, <code>ElementName</code> can be any unique identifier, but it is only used within the template syntax. The class referred to in <code>FULLY-QUALIFIED-CLASS-NAME</code> is inserted when you use the template. Specifying <code>editable="false"</code> merely ensures that the inserted class name is not highlighted for editing. For example, the forc code template uses <code> \${iter type="java.util.Iterator" editable=false}</code> to enter <code>Iterator</code> into the generated code and add an import statement for that class.
<code> \${ElementName}instanceof="FULLYQUALIFIED-CLASS-NAME"</code>	Declares that, if possible, the variable text that is initially generated should be the name of an instance variable of <code>FULLY-QUALIFIED-CLASS-NAME</code> that has been declared in the class. If there is no instance of that class available, <code>ElementName</code> is inserted.

	This construct is used in the <code>forc</code> and <code>fore</code> templates.
<code> \${ElementName}array }</code>	Declares that, if possible, the variable text that is initially generated should be the name of an array that is used in the class. If there is no array available within the current scope, <code>ElementName</code> is inserted.
	This construct is used in the <code>fori</code> template.
<code> \${ElementName}index }</code>	Generates a variable that is unused in the current scope, the default being <code>i</code> . If <code>i</code> is already used, then <code>j</code> is attempted, and then <code>k</code> , etc.
<code> \${cursor}</code>	Determines where the cursor will end up after the code snippet has been inserted and all of the variable text has been filled in.

Changing the Expander Shortcut for Code Templates

If you find that the code templates get in your way because they inadvertently get invoked when you type certain strings, you can configure the IDE to activate the templates with a different key or key combination. This enables you to continue using the code template feature without having to individually change any templates that get in your way.

To change the code template expander key:

1. Choose Tools | Options, click Editor in the left panel, and select the Code Templates tab.
2. Select the preferred key or key combination from the Expand Template On drop-down list.

3. Click OK to save the change and exit the Options dialog box.

Using Editor Hints to Generate Missing Code

When the IDE detects an error for which it has identified a possible fix, a lightbulb (💡) icon appears in the left margin of that line. You can click the lightbulb or press Alt-Enter to display a list of possible fixes. If one of those fixes suits you, you can select it and press Enter to have the fix generated in your code.

Often, the "error" is not a coding mistake but a reflection of the fact that you have not gotten around to filling in the missing code. In those cases, the editor hints simply automate the entry of certain types of code.

For example, assume you have just typed the following code, but `x` is not defined anywhere in the class.

```
int newIntegerTransformer () {  
    return x;  
  
}
```

If your cursor is still resting on the line of the `return` statement, the 💡 icon will appear. If you click the icon or press Alt-Enter, you will be offered three possible solutions as shown in [Figure 5-6](#). You can select one of those hints to generate the code.

Figure 5-6. Display of editor hints





In NetBeans IDE 5.0, editor hints only appear when you are in the line where the error is detected. In subsequent releases that is likely to change.

The IDE is able to provide hints for and generate the following solutions to common coding errors:

- Add a missing `import` statement
- Insert abstract methods that are declared in a class' implemented interfaces and abstract superclasses
- Insert a method parameter
- Create a missing method
- Create a missing field
- Create a missing local variable
- Initialize a variable
- Insert a cast
- Add a `throws` clause with the appropriate exception
- Surround the code with a `try-catch` block including the appropriate exception

Matching Other Words in a File

If you are typing a word that appears elsewhere in your file, you can use a keyboard shortcut to complete that word according to the first word found in the Source Editor that matches the characters you have typed. This word-match feature works for any text in the file. It also searches through files that you have been recently working in (in the order that you have last accessed the files).

To search backward from the cursor for a match, press Ctrl-K.

To search forward from the cursor for a match, press Ctrl-L.

For example, if you have defined the method `refreshCustomerInfo` on line 100 and now want to call that method from line 50, you can type `ref` and then press Ctrl-L. If there are no other words that start with `ref` between lines 50 and 100, the rest of the word `refreshCustomerInfo` will be filled in. If a different match is found, keep pressing Ctrl-L until the match that you want is filled in.



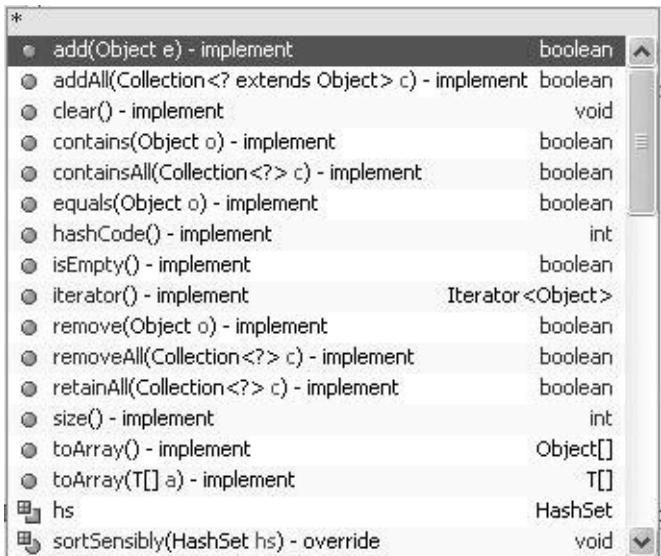
For typing variable names, you might find that the word match feature is preferable to code completion, since the IDE only has to search a few files for a text string (as opposed to code completion feature, where the IDE searches the whole classpath).

Generating Methods to Implement and Override

When you extend a class or implement an interface, you have abstract methods that you need to implement and possibly non-abstract methods that you can override. The IDE has several tools that help you generate these methods in your class:

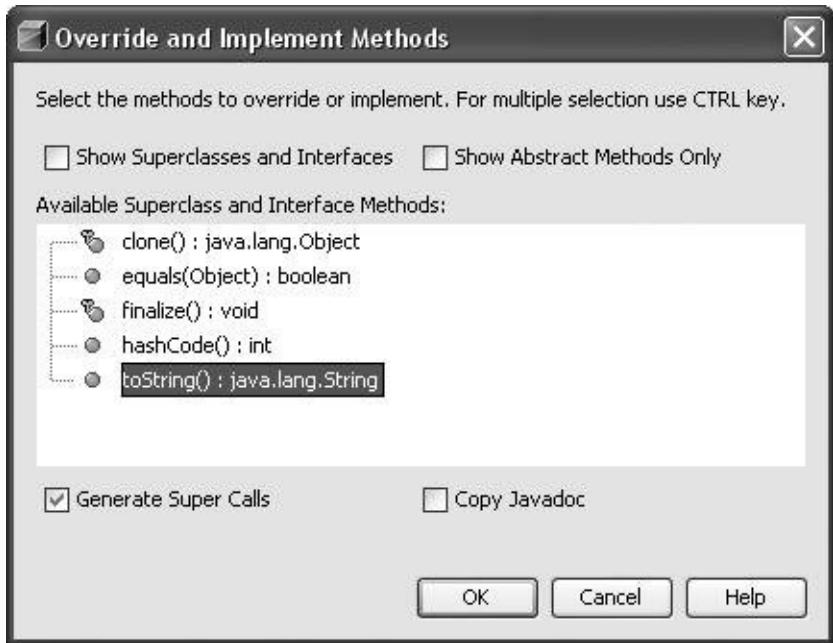
- **Editor hints.** When you add the `implements` or `extends` clause, a lightbulb (💡) icon appears in the left margin. You can click this icon or press AltEnter to view a hint to implement abstract methods. If you select the hint and press Enter, the IDE generates the methods for you. This hint only is available when your cursor is in the line of the class declaration.
- **Code completion.** You can generate methods to implement and override individually by pressing Ctrl-Space and choosing the methods from the code completion box. As shown in [Figure 5-7](#), methods to implement or override are marked `implement` and `override`, respectively.

Figure 5-7. Code completion box showing methods to implement and methods available to be overridden



- **Override and Implement Methods dialog box.** You can use this dialog box (shown in [Figure 5-8](#)) for generating any combination of the available implementable or overridable methods. This feature also enables you to generate calls to the super implementation of the methods within the body of the generated methods. To open this dialog box, choose Source | Override Methods or press Ctrl-I. To select multiple methods, use Ctrl-click.

Figure 5-8. Override and Implement Methods dialog box

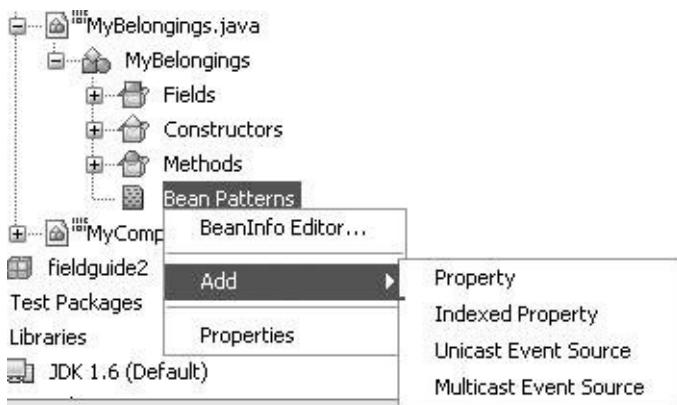


Generating JavaBeans Component Code

The IDE has a few levels of support for creating JavaBeans components. You can use the following features:

- **Code completion.** When you have a field in your class without a corresponding `get` or `set` method, you can generate that method by pressing Ctrl-Space and choosing the method from the code completion box.
- **Refactor | Encapsulate Fields command.** You can use this command to generate `get` and `set` methods, change the field's access modifier, and update code that directly accesses the field to use the getters and setters instead.
- **Bean Patterns node.** In the Projects window, each class has a subnode called Bean Patterns as shown in [Figure 5-9](#). You can right-click this node and choose from a variety of commands that enable you to generate code for bean properties, property change support, and event listening. In addition, you can generate BeanInfo classes.

Figure 5-9. Bean Patterns node and menu



Creating and Using Macros

You can record macros in the IDE to reduce what would normally involve a long set of keystrokes to one keyboard shortcut. In macros, you can combine the typing of characters in the Source Editor and the typing of other keyboard shortcuts.

To record a macro:

1. Put the insertion point in the part of a file in the Source Editor where you want to record the macro.
2. Click the  button in the Source Editor's toolbar (or press Ctrl-J and then type ) to begin recording.
3. Record the macro using any sequence of keystrokes, whether it is the typing of characters or using keyboard shortcuts. Mouse movements and clicks (such as menu selections) are not recorded.
4. Click the  in the Source Editor's toolbar (or press Ctrl-J and then type ) to finish recording.
5. In the Macro field of the Recorded Macro dialog box that appears, fine-tune the macro, if necessary.
6. Click Add to assign a keyboard shortcut to the macro.
7. In the Add Keybinding dialog box, press the keys that you want to use for the keyboard shortcut. (For example, if you want the shortcut Alt-Shift-Z, press the Alt, Shift, and Z keys.) If you press a wrong key, click the Clear button to start over.

Be careful not to use a shortcut that is already assigned. If the shortcut you enter is an editor shortcut, a warning appears in the dialog box. However, if the key combination is a shortcut

that applies outside of the Source Editor, you will not be warned.

You can assign a new shortcut in the Options window. Choose Tools | Options, click the Editor panel, select the Macros tab, and then click the Set Shortcut button.

Creating and Customizing File Templates

You can customize the templates that you create files from in the IDE and create your own templates. This might be useful if you need to add standard elements in all of your files (such as copyright notices) or want to change the way other elements are generated.

You can also create your own templates and make them available in the New File wizard.

There are several macros available for use in templates to generate text dynamically in the created files. These macros are identifiable by the double underscores that appear both before and after the macro name. See [Table 5-3](#) for a list of the macros available.

Table 5-3. Java File Template Macros

Macro	Substituted Information
<code>__USER__</code>	Your username. If you would like to change the value of <code>__USER__</code> , choose Tools Options, click Advanced Options, and select the Editing Java Sources node. Then click the button in the Strings Table property and change the value of USER.
<code>__DATE__</code>	The date the new file is created.
<code>__TIME__</code>	The time the new file is created.
<code>__NAME__</code>	The name of the class (without the file extension). It is best not to use this macro for the class and constructor name in the file (instead, use the filename).

<u>PACKAGE</u>	The name of the package where the class is created.
<u>PACKAGE_SLASHES</u>	The name of the class' package with slash (/) delimiters instead of periods (.).
<u>PACKAGE_AND_NAME</u>	The fully qualified name of the file (such as com.mydomain.mypackage.MyClass).
<u>PACKAGE_AND_NAME_SLASHES</u>	The fully qualified name of the file with slash (/)delimiters instead of periods (.).
<u>QUOTES</u>	A double quote mark ("). Use this macro if you want the substituted text to appear in quotes in the generated file. (If you place a macro within quote marks in the template, text is not substituted for the macro name in the created file.)

To edit a template:

- 1.** Choose Tools | Template Manager.
- 2.** Expand the appropriate category node and select the template that you want to edit.
- 3.** Click Open in Editor.
- 4.** Edit the template and then save it.



Not all of the templates listed in the Template Manager can be modified at the user level. In some cases, the templates are available in the New File wizard but do not represent file constructs (such as those in the Enterprise and Sun Resources categories).

To create a new file template based on another template:

- 1.** Choose Tools | Template Manager.
- 2.** Navigate to and select the template on which you want to model the new template and click Duplicate.

A new node appears for the copied template. _1 is appended to the template's name.

- 3.** Click Open in Editor.
- 4.** Edit the file, incorporating any of the template macros that you want to use (see [Table 5-3](#)), and save it.

If the template is for a Java class, you can use the filename for the class name and constructor name. These are automatically adjusted in the files you create from the template.

To import a file template:

- 1.** Choose Tools | Template Manager.
- 2.** Select the category folder for the template.
- 3.** Click Add to open the Add Template dialog box.
- 4.** Navigate to and select the file that you want to import as a template. Then click Add.

Handling Imports

When you use the IDE's code completion and editor hints features, import statements are generated for you automatically.

For example, if you have the code completion box open and you start typing a simple class name instead of its fully-qualified class name (e.g., you type `Con` and then select `ConcurrentHashMap` from the code completion box), the following import statement will be added to the beginning of the file:

```
import java.util.concurrent.ConcurrentHashMap;
```

For cases where these mechanisms are not sufficient for the management of import statements, you can use the following commands:

- Fix Imports (Alt-Shift-F), which automatically inserts any missing import statements for the whole file. Import statements are generated by class (rather than by package). For rapid management of your imports, use this command.
- Fast Import (Alt-Shift-I), which enables you to add an import statement or generate the fully qualified class name for the currently selected identifier. This command is useful if you want to generate an import statement for a whole package or if you want to use a fully qualified class name inline instead of an import statement.

Displaying Javadoc Documentation While Editing

The IDE gives you a few ways to access documentation for JDK and library classes.

To glance at documentation for the currently selected class in the Source Editor, press Ctrl-Shift-spacebar. A popup window appears with the Javadoc documentation for the class. This popup also appears when you use code completion. You can dismiss the popup by clicking outside of the popup.

To open a web browser on documentation for the selected class, right-click the class and choose Show Javadoc (or press Alt-F1).

To open the index page for a library's documentation in a web browser, choose View | Documentation Indices and choose the index from the submenu.



Documentation for some libraries is bundled with the IDE. However, you might need to register the documentation for other libraries in the IDE for the Javadoc features to work. See [Making External Sources and Javadoc Available in the IDE](#) in [Chapter 3](#) for more information.

Paradoxically, JDK documentation is available through a popup in the Source Editor but not through a browser by default. This is because the Javadoc popup in the Source Editor picks up the documentation from the sources that are included with the JDK. However, the browser view of the documentation requires compiled Javadoc documentation, which you have to download separately from the JDK. See [Referencing JDK Documentation \(Javadoc\) from the Project](#) in [Chapter 3](#).

Formatting Code

When you type or have code generated in the Source Editor, your Java code is automatically formatted in the following ways by default:

- Members of classes are indented four spaces.
- Continued statements are indented eight spaces.
- Any tabs that you enter are converted to spaces.
- When you are in a block comment (starting with `/**`), an asterisk is automatically added to the new line when you press Enter.
- The opening curly brace is put on the same line as the declaration of the class or method.
- No space is put before an opening parenthesis.

If your file loses correct formatting, you can reformat the whole file by selecting Source | Reformat Code (Ctrl-Shift-F). If you have any lines selected, the reformatting applies only to those lines.

If you have copied code, you can have it inserted with correct formatting by pasting with the Ctrl-Shift-V shortcut.

Indenting Blocks of Code Manually

You can select multiple lines of code and then indent all those

lines by pressing Tab or Ctrl-T.

You can reverse indentation of lines by selecting those lines and then pressing Shift-Tab or Ctrl-D.

Changing Formatting Rules

For various file types, you can adjust formatting settings, such as for number of spaces per tab, placement of curly braces, and so on.

To adjust formatting rules for Java files:

- 1.** Choose Tools | Options.
- 2.** Click Editor in the left panel and select the Indentation tab.
- 3.** Adjust the properties for the indentation engine to your taste.
- 4.** Reformat each file to the new rules by opening the file and pressing CtrlShift-F (with no text selected).

For other file types, you can change formatting properties in the Advanced Options part of the Options Window and by adjusting that file type's indentation engine.

To change formatting rules for non-Java file types:

- 1.** Choose Tools | Options and click Advanced Options.
- 2.** Expand Editing | Indentation Engines, and select the indentation engine for the file type for which you want to modify formatting rules.

If there is no specific indentation engine for your file type, find out which indentation is being used for that file type by

expanding Editing | Editor Settings, selecting the editor type, and looking at the value of its Indentation Engine property.

- 3.** Adjust the properties for the indentation engine to your taste.
- 4.** Reformat each file to the new rules by opening the file and pressing CtrlShift-F (with no text selected).

There are other preset indentation engines available (the "simple" and "line wrapping" indentation engines) you might prefer to use.

To change the indentation engine that is used for a file type:

- 1.** Choose Tools | Options and click Advanced Options.
- 2.** Expand Editing | Editor Settings and select the node for the editor type for which you want to change the indentation engine.
- 3.** Select the indentation engine from the Indentation Engine property's combo box.
- 4.** Reformat each file to the new rules by opening the file and pressing CtrlShift-F (with no text selected).

To create a new indentation engine:

- 1.** Choose Tools | Options and click Advanced Options.
- 2.** Expand Editing | Indentation Engines, right-click the node for the indentation engine on which you want to base your new indentation engine, and choose Copy.
- 3.** Right-click the Indentation Engines node and choose Paste | Copy.

- 4.** Modify the name of the indentation inline and adjust the properties to your taste.
- 5.** In the Advanced Options part of the Options dialog box, expand Editing | Editor Settings and select the node for the editor (such as Java Editor or HTML Editor) that you want the indentation engine to apply to.

Changing Fonts and Colors

You can adjust the fonts that are used in the Source Editor and the way colors and background highlighting are used to represent syntactic elements of your code. You can make changes that apply to all file types and changes that apply to specific file types.

To make changes in fonts and colors that are reflected throughout all editor types:

- 1.** Choose Tools | Options and click the Fonts & Colors panel.
- 2.** In the Languages drop-down list, select All Languages.
- 3.** In the Category list, select Default.
- 4.** Use the Font, Foreground, Background, Effects, and Effect Color fields to change the appearance of that code element.

These changes should be reflected in the syntax categories for all the languages, since other categories are essentially designed as customizations of this one.

To change fonts and colors for a specific code syntax element:

- 1.** Choose Tools | Options and click the Fonts & Colors panel.
- 2.** Select the editor type from the Languages drop-down list.

If the syntax element applies to multiple languages, select All Languages.

- 3.** Select a syntax element in the Category list.
- 4.** Use the Font, Foreground, Background, Effects, and Effect Color fields to change the appearance of that code element.



You can also create profiles of fonts and colors. The IDE comes with two profiles built in NetBeans (the default profile) and City Lights (which is based on a black background).

You can choose the profile from the Profile combo box at the top of the Fonts & Colors panel. You can create a new profile by clicking Duplicate to start a new profile based on the selected profile.

Text Selection Shortcuts

To enable you to keep both hands on the keyboard, a number of shortcuts allow you to select text, deselect text, and change the text that is selected. See [Table 5-4](#) for a selection of these shortcuts.

Table 5-4. Text Selection Shortcuts

Description	Shortcut
Selects the current identifier or other word that the insertion point is on.	Alt-J
Selects all the text between a set of parentheses, brackets, or curly braces. The insertion point must be resting immediately after either the opening or closing parenthesis/bracket/brace.	Ctrl-Shift-[
Selects the current code element. Upon subsequent Alt-Shift-S pressings, incrementally increases the size of the selection to include surrounding code elements. For example, if you press Alt-Shift-S once, the current word is selected. If you press it again, the rest of the expression might be selected. Pressing a third time might select the whole statement. Pressing a fourth time might select the whole method.	Alt-Shift-S
Selects the next (previous) character or extends the selection one character.	Shift-Right (Shift-Left)
Selects the next (previous) word or extends the selection one word.	Ctrl-Shift-Right (Ctrl-Shift-Left)
Creates or extends the text selection one line down (up).	Shift-Down (Shift-Up)
Creates or extends the text selection to the end (beginning) of the line.	Shift-End (Shift-Home)
Creates or extends the text selection to the end	Ctrl-Shift-End

(beginning) of the document.

(Ctrl-Shift-
Home)

Creates or extends the text selection one page down Shift-Page Down
(up). Shift-Page Up)

Navigating within the Current Java File

The IDE provides several mechanisms to make it easier to view and navigate a given Java file:

- The Navigator window, which appears below the Projects window and provides a list of members (for example, constructors, fields, and methods) in the currently selected Java file.
- Bookmarks, which enable you to easily jump back to specific places in the file.
- The Alt-K and Alt-L "jump list" shortcuts, mentioned in Jumping Between Areas Where You Have Been Working later in this chapter.
- Keyboard shortcuts to scroll the window. See [Table 5-5](#) in the following section.

Table 5-5. Cursor and Scrolling Shortcuts

Description	Shortcut
Moves the insertion point to the next word (previous word).	Ctrl-Right (Ctrl-Left)
Moves the insertion point to the top (bottom) of the file.	Ctrl-Home (Ctrl-End)
Scrolls up (down) without moving the insertion point.	Ctrl-Up (Ctrl-Down)
Scrolls the window so that the current line moves to the top of the window.	Alt-U, then T

Scrolls the window so that the current line moves to the middle of the window. Alt-U, then M

Scrolls the window so that the current line moves to the bottom of the window. Alt-U, then B

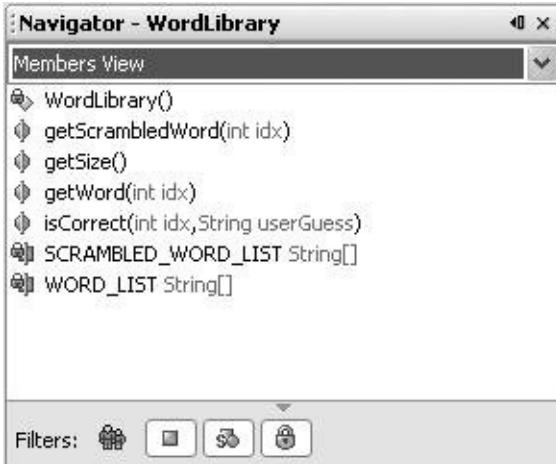
Moves the insertion point to the parenthesis, bracket, or curly brace that matches the one directly before your insertion point. Ctrl-[

- The code folding feature, which enables you to collapse sections of code (such as method bodies, Javadoc comments, and blocks of import statements), thus making a broader section of your class visible in the window at a given time.

Viewing and Navigating Members of a Class

The IDE's Navigator window (shown in [Figure 5-10](#)) provides a list of all "members" (constructors, methods, and fields) of your class. You can double-click a member in this list to jump to its source code in the Source Editor. Alternatively, instead of using the mouse, press Ctrl-7 to give focus to the Navigator window. Then begin typing the identifier until the Navigator locates it and press Enter to select that identifier in the Source Editor.

Figure 5-10. Navigator window



You can use the filter buttons at the bottom of the window to hide non-public members, static members, fields, and/or inherited members.

Moving the Insertion Point and Scrolling the Window

There is a wide range of shortcuts that you can use for moving the insertion point around and scrolling the Source Editor without moving the insertion point. See [Table 5-5](#) for a list of some of the most useful file navigation shortcuts.

Bookmarking Lines of Code

You can set bookmarks in files to make it easy to find an area of the file that you are working with frequently. You can then cycle through the file's bookmarks by pressing F2 (next bookmark) or Shift-F2 (previous bookmark).

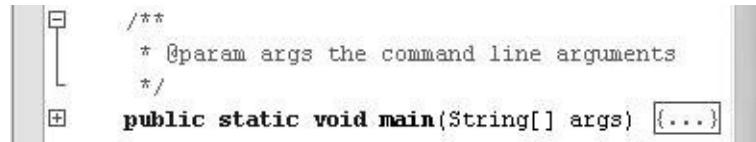
To bookmark a line in a file, click in the line and press Ctrl-F2. To remove a bookmark, also use Ctrl-F2.

Hiding Sections of Code

You can collapse (or *fold*) low-level details of code so that only one line of that block is visible in the Source Editor, leaving more room to view other lines. Methods, inner classes, import blocks, and Javadoc comments are all foldable.

Collapsible blocks of code are marked with the  icon in the left margin next to the first line of the block. The rest of the block is marked with a vertical line that extends down from the  icon. Collapsed blocks are marked with the  icon. You can click one of these icons to fold or expand the particular block it represents. See [Figure 5-11](#) for an example.

Figure 5-11. Examples of expanded and folded code in the Source Editor



You can also collapse and expand single or multiple blocks of code with keyboard shortcuts and menu items in the Edit | Code Folds menu and the Code Folds submenu in the Source Editor. See [Table 5-6](#) for a list of these commands and shortcuts.

Table 5-6. Code Folding Commands

Command	Shortcut
Collapse Fold	Ctrl-NumPad-Minus
Expand Fold	Ctrl-NumPad-Plus

Collapse All	Ctrl-Shift-NumPad-Minus
Expand All	Ctrl-Shift-NumPad-Plus
Collapse All Javadoc	none
Expand All Javadoc	none
Collapse All Java Code (collapses everything except Javadoc documentation)	none
Expand All Java Code (expands everything except Javadoc documentation)	none

By default, none of the code that you write is folded. You can configure the Source Editor to fold Java code by default when you create a file or open a previously unopened file.

To configure the IDE to fold certain elements of Java code automatically:

1. Choose Tools | Options, click the Editor panel, and select the General tab.
2. Where it says Collapse By Default, select the checkboxes for the elements that you want folded by default. You can choose from methods, inner classes, imports, Javadoc comments, and the initial comment.

Navigating from the Source Editor

The IDE includes handy shortcuts for navigating among files, different bits of code, and different windows. The more of these shortcuts you can incorporate into your workflow, the less your fingers will have to stray from your keyboard to your mouse.

Switching Between Open Files

Besides using the Source Editor's tabs and drop-down list, you can switch between open files using the keyboard shortcuts shown in [Table 5-7](#).

Table 5-7. Shortcuts for Navigating Among Open Files

Shortcut	Description
Alt-Left and Alt-Right	Select files in order of tab position.
Ctrl-Tab	Opens a popup box showing all open files. Hold down the Ctrl key and press the Tab key multiple times until the file that you want to view is selected. Then release both keys to close the box and display the file.
Shift-F4	Opens a dialog box that lists all open files. You can use the mouse or the arrow keys to select the file that you want to view and press Enter to close the dialog box and display the file.

Jumping to Related Code and Documentation

The shortcuts in [Table 5-8](#) enable you to jump to parts of the current file or other files that are relevant to the selected identifier. The first six of these shortcuts are available from the Navigate menu and the Go To submenu of the Source Editor's contextual (right-click) menu. The Show Javadoc command is available straight from the Source Editor's contextual menu.

Table 5-8. Java Class Navigation Shortcuts

Command	Shortcut	Description
Go to Source	Alt-O (or Ctrl-click)	Jumps to the source code for the currently selected class, method, or field, if the source is available. You can achieve this either by pressing Alt-O with the identifier selected or by holding down the Ctrl key, hovering the mouse over the identifier until it is underlined in blue, and then clicking it.
Go to Declaration	Alt-G	Jumps to the declaration of the currently selected class, method, or field.
Go to Super Implementation	Ctrl-B	Jumps to the super implementation of the currently selected method (if the selected method overrides a method from another class or is an implementation of a method defined in an interface).
Go to Line	Ctrl-G	Jumps to a specific line number in the current file.
Go to Class	Alt-Shift-O	Enables you to type a class name and then jumps to the source code for that class if it is available to the IDE.
Go To Test	Alt-Shift-E	Jumps to the unit test for the selected class.

Show Javadoc	Alt-F1	Displays documentation for the selected class in a web browser. For this command to work, Javadoc for the class must be made available to the IDE through the Java Platform Manager (for JDK documentation) or the Library Manager (for documentation for other libraries). See Referencing JDK Documentation (Javadoc) from the Project in Chapter 3 and Making External Sources and Javadoc Available in the IDE , also in Chapter 3 .
--------------	--------	--

Jumping Between Areas Where You Have Been Working

When you are working on multiple files at once or in different areas of the same file, you can use the "jump list" shortcuts to navigate directly to areas where you have been working instead of scrolling and/or switching windows. The "jump list" is essentially a history of lines where you have done work in the Source Editor.

You can navigate back and forth between jump list locations with the Alt-K (back) and Alt-L (forward) shortcuts. Use Alt-Shift-K and Alt-Shift-L to navigate files in the jump list without stopping at multiple places in a file.

Jumping from the Source Editor to a File's Node

When you are typing in the Source Editor, you can jump to the node that represents the current file in other windows. This can be useful, for example, if you want to navigate quickly to

another file in the same package or you want to browse versioning information for the current file.

See [Table 5-9](#) for a list of available shortcuts.

Table 5-9. Shortcuts for Selecting the Current File in a Different Window

Command	Shortcut
Select the node for the current file in the Projects window.	Ctrl-Shift-1
Select the node for the current file in the Files window.	Ctrl-Shift-2
Select the node for the current file in the Favorites window.	Ctrl-Shift-3

Searching and Replacing

There are several types of searches in the IDE for different needs. You can

- Find occurrences of an identifier for a class, method, or field in your project using the Find Usages command
- Rename a class, method, or field throughout your project by using the Rename command
- Find and replace specific character combinations in an open file by pressing Ctrl-F in the file
- Find files that match search criteria based on characters in the file, characters in the filename, file type, and/or date by right-clicking a folder or project node in the Projects window and choosing Find (or by pressing Ctrl-F)

Finding Occurrences of the Currently Selected Class, Method, or Field Name

When you are working in the Source Editor, you can quickly find out where a given Java identifier is used in your project using the Find Usages command.

Find Usages improves upon a typical Find command by being sensitive to the relevance of text in the Java language context.

Depending on what kind of identifier you have selected and which options you have selected in the Find Usages dialog box, the Find Usages command output displays lines in your project that contain a combination of the following items:

- (For classes and interfaces) a declaration of a method or variable of the class or interface
- (For classes and interfaces) a usage of the type, such as at the creation of a new instance, importing a class, extending a class, implementing an interface, casting a type, or throwing an exception
- (For classes and interfaces) a usage of the type's methods or fields
- (For classes and interfaces) subtypes
- (For fields) the getting or setting of the field's value
- (For methods) the calling of the method
- (For methods) any overriding methods
- Comments that reference the identifier

The Find Usages command does not match

- Parts of words
- Words that differ in case

To find occurrences of a specific identifier in your code:

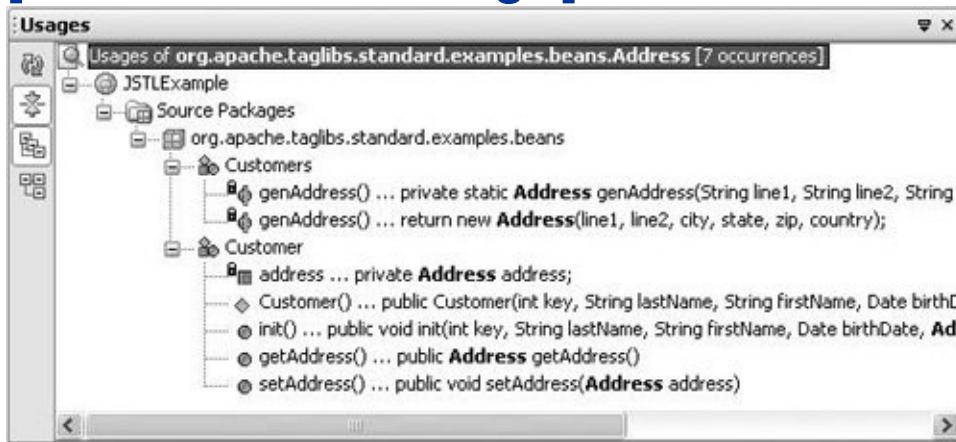
1. In the Source Editor, move the insertion point to the class, method, or field name that you want to find occurrences of.
2. Choose Edit | Find Usages, right-click and choose Find Usages, or press Alt-F7.

- 3.** In the Find Usages dialog box, select the types of occurrences that you want displayed and click Next.

The results are displayed in the Usages window (shown in [Figure 5-12](#)), which appears at the bottom of the IDE.

Figure 5-12. Usages window

[[View full size image](#)]



You can navigate to a given occurrence of a class, method, or field name by doubleclicking the occurrences line in the Usages window.

Renaming All Occurrences of the Currently Selected Class, Method, or Field Name

If you want to rename a class, method, or field, you can use the Refactor | Rename command to update all occurrences of the identifier in the Project to the new name. Unlike standard search and replace operations, the Rename command is sensitive to the Java context of your code, which makes it much

more easy and reliable to use when reworking code. In addition, with the Rename command, you get a preview of the changes to be made and can prevent renaming of specific occurrences.

To rename a class, method, or field name:

1. In the Source Editor, move the insertion point to an occurrence in the code of the class, method, or field name that you want to rename.
2. Right-click and choose Refactor | Rename or press Alt-Shift-R.
3. In the Rename dialog box, type the new name for the element.

If you want occurrences of the name in comments to also be changed, check the Apply Name on Comments checkbox.

4. In the Rename dialog box, click Next.

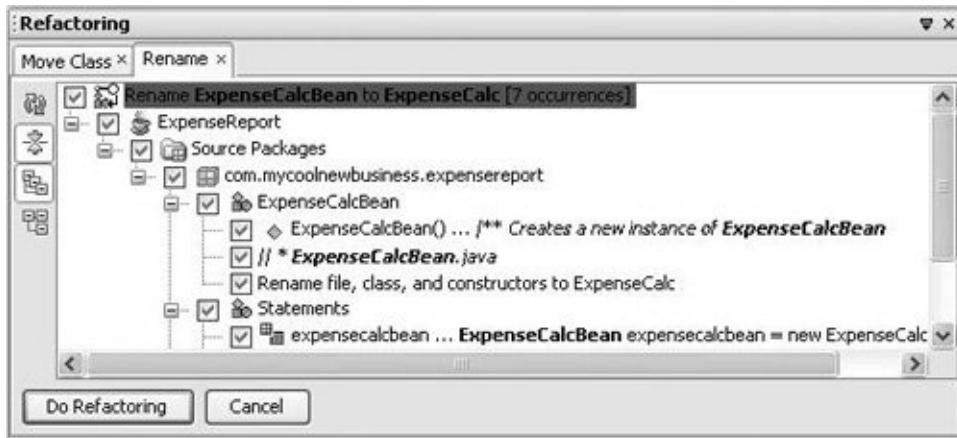
If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

5. In the Refactoring window (shown in [Figure 5-13](#)), which appears at the bottom of the IDE, verify the occurrences that are set to change. If there is an occurrence that you do not want to change, deselect that line's checkbox.

Figure 5-13. Refactoring preview window

[\[View full size image\]](#)



6. Click Do Refactoring to apply the changes.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

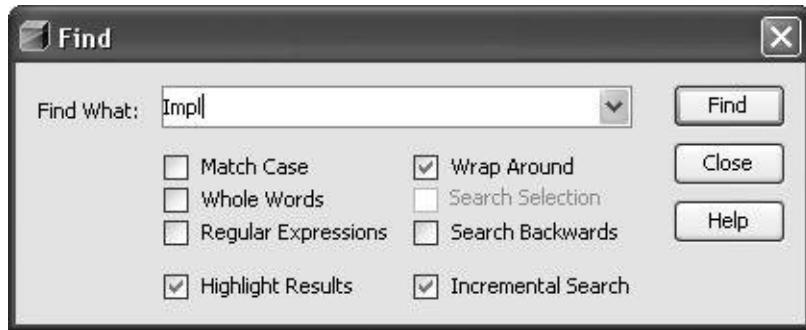


You can initiate the renaming of a class or interface by renaming it inline in the Projects window. If you rename a node but do not want that change to be reflected in other places, select the Rename Without Refactoring checkbox.

Searching and Replacing Combinations of Characters in a File

If you merely want to find a combination of characters in your file, click in the file that you want to search, choose Edit | Find (Ctrl-F), and type the text that you want to find in the Find dialog box (as shown in [Figure 5-14](#)).

Figure 5-14. Find window for the Source Editor



In the Find dialog box, you can use a regular expression as your search criterion by selecting the Regular Expressions checkbox.

Unlike the Find Usages command, the Find command allows you to search for parts of words, do case-insensitive searches, and highlight matches in the current file.

Once you have dismissed the Find dialog box, you can jump between occurrences of the search string by pressing F3 (next occurrence) and Shift-F3 (previous occurrence).

To select the word in which the cursor is resting and start searching for other occurrences of that word, press Ctrl-F3.

To search and replace text, click in the file that you want to replace text, press Ctrl-H, and fill in the Find What and Replace With fields.



By default, matches to a Find command remain highlighted in the Source Editor after you have dismissed the Find dialog box. To turn off the highlighting, press Alt-Shift-H.

Other File Searches

If you want to do a search on multiple files for something other than an occurrence of a specific Java identifier, you can use the Find and Find in Projects commands. These commands enable you to search files within a folder, project, or all projects.

You can base these commands on any combination of the following types of criteria:

- Matches to a substring or regular expression on text in the file
- Matches to a substring or regular expression on the filename
- Dates the files were modified
- File type

To initiate such a file search, do one of the following:

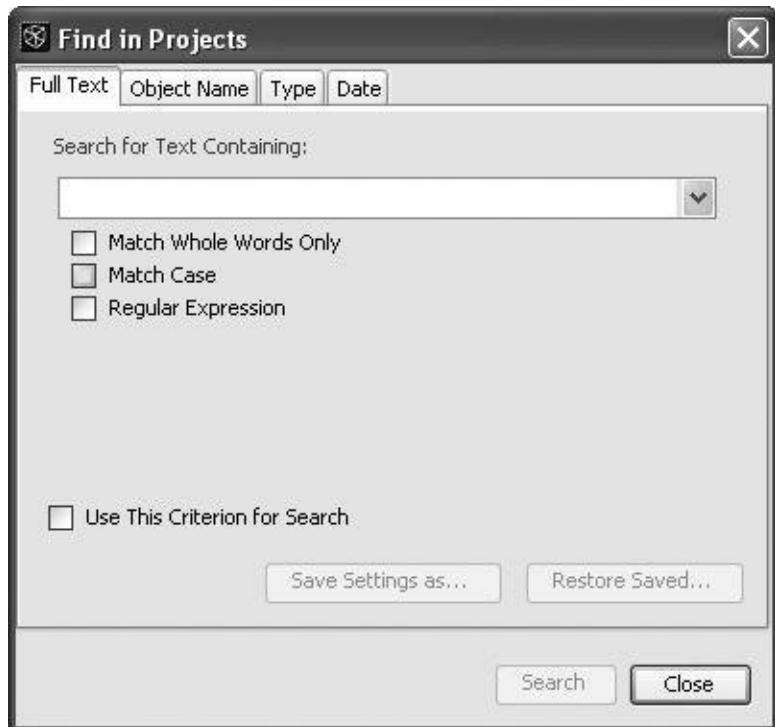
- Choose Edit | Find in Projects to search all files in all open projects (including project metadata files).
- In the Projects window, right-click the node for the folder or project that you want to search in and choose Find (or press Ctrl-F). If you choose Find this way, the project metadata, including the build script and the contents of the `nbproject` folder, are not searched.
- Right-click a folder in the Files window and choose Find. If you choose Find this way, the project metadata, including the build script and the contents of the `nbproject` folder, are

also searched.

After you initiate the search, fill as many search criteria as you would like. When you fill in a criterion on a given tab, the Use This Criterion for Search checkbox is selected. Deselect this checkbox if you decide to search according to a different type of criterion and you do not want the criterion on the currently selected tab to be used.

After you enter the criteria in the Find dialog box or the Find in Projects dialog box (shown in [Figure 5-15](#)) and click Search, the results are displayed in the Search Results window with nodes for each matched file. For full-text searches, these nodes can be expanded to reveal the individual lines where matched text occurs. You can double-click a match to open that file in the Source Editor (and, in the case of full-text matches, jump to the line of the match).

Figure 5-15. Find in Projects dialog box





The dialog box that appears when you press Ctrl-F or choose Edit | Find (or Edit | Find in Projects) depends on which IDE window has focus. If you have the Source Editor selected, the Find dialog box for an individual file appears. If you have a node selected in the Projects window (or one of the other tree-view windows), the dialog box for searching in multiple files is opened.

Deleting Code Safely

Over time, your code might gather elements that have limited or no usefulness. To make the code easier to maintain, it is desirable to remove as much of this code as possible. However, it might be hard to immediately determine whether you can delete such code without causing errors elsewhere.

Refactoring Commands

NetBeans IDE has special support for refactoring code. The term *refactoring* refers to renaming and rearranging code without changing what the code does. Reasons for refactoring include things such as the need to separate API from implementation, making code easier to read, and making code easier to reuse.

The IDE's refactoring support makes refactoring easier by enabling you to update all of the code in your project automatically to reflect changes that you make in other parts of your project.

For example, if you rename a class, references to that class in other classes are also updated.

You can access most refactoring commands from the Refactor menu on the main menu bar or by right-clicking in the Source Editor or on a class node in the Projects window and choosing from the Refactor submenu. The Find Usages command is in the Edit menu and the contextual (right-click) menu for the Source Editor and the Projects window.

Typically, the currently selected identifier is filled in as the code element to be refactored.

[Table 5-10](#) provides a summary of the refactoring commands that are available. These commands are explained more thoroughly in task-specific topics throughout this chapter.

Table 5-10. Refactoring Commands

Command	Description
Find Usages	Displays all occurrences of the name of a given class, method, or field. See Finding Occurrences of the Currently Selected Class, Method, or Field Name earlier in this chapter.
Rename	Renames all occurrences of the selected class, interface, method, or field name. See Renaming All Occurrences of the Currently Selected Class, Method, or Field Name earlier in this chapter.
Safely Delete	Deletes a code element after making sure that no

other code references that element. See [Deleting Code Safely](#) earlier in this chapter.

Change Method Parameters	Enables you to change the parameters and the access modifier for the given method. See Changing a Method's Signature later in this chapter.
Encapsulate Fields	Generates accessor methods (getters and setters) for a field and changes code that accesses the field directly so that it uses those new accessor methods instead. See Encapsulating a Field later in this chapter.
Move Class	Moves a class to a different package and updates all references to that class with the new package name. See Moving a Class to a Different Package later in this chapter.
Pull Up	Moves a method, inner class, or field to a class' superclass. You can also use this command to declare the method in the superclass and keep the method definition in the current class. See Moving Class Members to Other Classes later in this chapter.
Push Down	Moves a method, inner class, or field to a class' direct subclasses. You can also use this command to keep the method declaration in the current class and move the method definition to the subclasses. See Moving Class Members to Other Classes later in this chapter.
Extract Method	Creates a new method based on a selection of code in the selected class and replaces the extracted statements with a call to the new method. See Creating a Method from Existing Statements later in this chapter.
Extract Interface	Creates a new interface based on a selection of methods in the selected class and adds the new interface to the class' <code>implements</code> clause. See Creating an Interface from Existing Methods later in this chapter.
Extract Superclass	Creates a new superclass based on a selection of methods in the selected class. You can have the class created with just method declarations, or you

can have whole method definitions moved into the new class. See [Extracting a Superclass to Consolidate Common Methods](#) later in this chapter.

Use Supertype Where Possible	Change code to reference objects of a superclass (or other type) instead of objects of a subclass. See Changing References to Use a Supertype later in this chapter.
Move Inner to Outer Level	Moves a class up one level. If the class is a top-level inner class, it is made into an outer class and moved into its own source file. If the class is nested within the scope of an inner class, method, or variable, it is moved up to the same level as that scope. See Unnesting Classes later in this chapter.
Convert Anonymous Class to Inner	Converts an anonymous inner class to a named inner class. See Unnesting Classes later in this chapter.

The IDE's Safely Delete command can help you with the process of removing unused code, saving you cycles of manual searches and compilation attempts. When you use this command, the IDE checks to see if the selected code element is referenced elsewhere. If the code element is not used, the IDE deletes it. If the code element is used, the IDE displays where the code is used. You can then resolve references to the code you want to delete and then try the Safely Delete operation again.

You can use the Safely Delete command on any type (such as a class, interface, and or enumeration), method, field, or local variable.

To safely delete a code element:

1. In the Source Editor or Projects window, right-click the code element that you want to delete and choose Refactor | Safely Delete.

2. In the Safe Delete dialog box, make sure that the item to be deleted is listed.
3. If you want the IDE to look inside comments for mentions of the code element, select the Search in Comments checkbox.

If this checkbox is not selected, comments referring to the code element are not affected if you delete the code element.

If this checkbox is selected, any found references to the code element in comments will prevent the code element from being immediately deleted, even if the code element is not used.

4. Click Next.

If no references to the code element are found, the Safe Delete dialog box closes and the code element is deleted.

If references to the code element are found, no code is deleted and the Safely Delete dialog box remains open. If you click the Show Usages button, the Usages window opens and displays the references to that code element. Double-click an item in the list to jump to the line of code that it represents. If you remove the cited references, you can click the Rerun Safe Delete button to repeat the attempt to safely delete the code element.



To undo the Safely Delete command, choose Refactor | Undo.

Changing a Method's Signature

If you want to change a method's signature, you can use the IDE's Refactor | Change Method Parameters command to update other code in your project that uses that method. Specifically, you can

- Add parameters.
- Change the order of parameters.
- Change the access modifier for the method.
- Remove unused parameters.

You cannot use the Change Method Parameters command to remove a parameter from a method if the parameter is used in your code.

To change a method's signature:

1. Right-click the method in the Source Editor or the Projects window and choose Refactor | Change Method Parameters.
2. Click Add if you want to add parameters to the method. Then edit the Name, Type, and (optionally) the Default Value cells for the parameter. You have to double-click a cell to make it editable.
3. To switch the order of parameters, select a parameter in the Parameters table and click Move Up or Move Down.
4. Select the preferred access modifier from the Access Modifier combo box.

5. Click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

6. In the Refactoring window, look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.

7. Click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

Encapsulating a Field

One common design pattern in Java programs is to make fields accessible and changeable only by methods in the defining class. In the convention used by JavaBeans components, the field is given private access and accessor methods are written for the field with broader access privileges. The names of the accessor methods are created by prefixing `get` and `set` to the field's name.

If you have fields that are visible to other classes and would like to better control access to those fields, you can use the IDE's Encapsulate Fields command to automate the necessary code modifications. The Encapsulate Fields command does the following things:

- Generates getter and setter methods for the desired fields.
- Enables you to change the access modifier for the fields and accessor methods.
- Changes code elsewhere in your project that accesses the fields directly to instead use the newly generated accessor methods.

To encapsulate fields in a class:

1. Right-click the field or the whole class in the Source Editor or the Projects window and choose Refactor | Encapsulate Fields.
2. In the Encapsulate Fields dialog box, select the Create Getter and Create Setter checkboxes for each field that you want to have encapsulated.

If you have selected a specific field, the checkboxes for just that field should be selected by default.

If you have selected the whole class, the checkboxes for all of the class' fields should be selected by default.

3. In the Fields' Visibility drop-down list, set the access modifier to use for the fields that you are encapsulating.

Typically, you would select `private` here. If you select a different visibility level, other classes will still have direct access to the fields for which you are generating accessor methods.

4. In the Accessors' Visibility drop-down list, set the access modifier to use for the generated getters and setters.
5. If you decide to leave the fields visible to other classes but you want to have current references to the field replaced with references to the accessor methods, select the Use Accessors Even When Field Is Accessible checkbox. Otherwise, those direct references to the field will remain in the code.

This checkbox is only relevant if you decide to leave the fields accessible to other classes and there is code in those classes that accesses the fields directly.

6. Click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

7. In the Refactoring window, verify the changes that are about to be made and click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

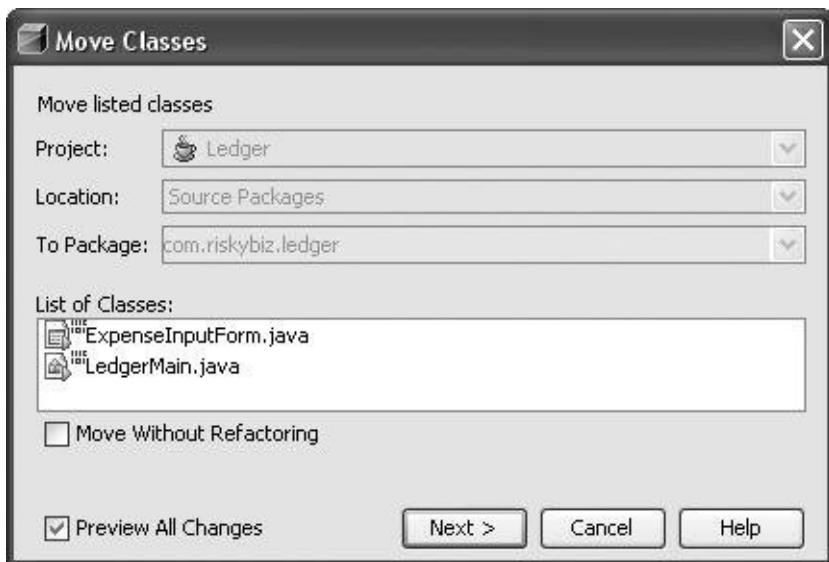
Moving a Class to a Different Package

If you want to place a class in a different package, you can use the IDE's refactoring features to move the class and then update references to that class automatically throughout your project.

To move a class:

1. In the Projects window, drag the class from its current package to the package you want to place it in. (You can also use the Cut and Paste commands in the contextual menus or the corresponding keyboard shortcuts.)
2. In the Move Class or Move Classes dialog box (shown in [Figure 5-16](#)), click Next after verifying that the To Package and This Class fields reflect the destination package and the class you are moving. (If you move multiple classes, a List of Classes text area is shown instead of the This Class field.)

Figure 5-16. Move Classes dialog box



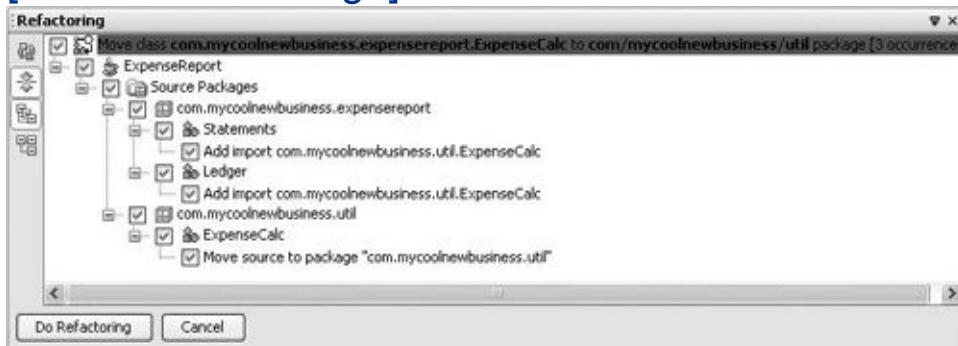
If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

3. In the Refactoring window (shown in [Figure 5-17](#)), look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.

Figure 5-17. Refactoring window

[\[View full size image\]](#)



4. Click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.



If you want to create a new package and move all of the classes in the old package to the new package, you can do an in-place rename of a package in the Projects window (or of a folder in the Files window).

Moving Class Members to Other Classes

The IDE has the Pull Up and Push Down refactoring commands for moving methods and fields to other classes and interfaces. When you use these commands to move class members, the IDE updates references to those members throughout your project. These commands are useful for improving the inheritance structure of your code. You can:

- Move methods and fields to a superclass or super-interface.
- Leave method implementations in the current class but create abstract declarations for those methods in a superclass.
- Move methods and fields to the class's subclasses or sub-interfaces.
- Move the implementations of methods to subclasses while leaving abstract method declarations in the current class.
- Move the interface name from the `implements` clause of a class to the `implements` clause of another class.

Moving Code to a Superclass

To move a member, a method's declaration, or an `implements` clause from the current class to a superclass:

1. In the Source Editor or the Projects window, select the class or interface that contains the member or members that you want to move.

- 2.** Choose Refactor | Pull Up to open the Pull Up dialog box.
- 3.** In the Destination Supertype drop-down list, select the superclass or interface that you want to move the members to.
- 4.** Select the checkbox for each member that you want to move.

If you want to leave the method implementation in the current class and create an abstract declaration for the method in the superclass, select the Make Abstract checkbox for the method.

If the class from which you are moving members implements an interface, a checkbox for that interface is included in the dialog box. If you select the checkbox for that interface, the interface is removed from the `implements` clause of the current class and moved to the `implements` clause of the class to which you are moving members. (If you select a checkbox for an interface, be sure that all the checkboxes for the methods declared in that interface are also selected.)

- 5.** Click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

- 6.** In the Refactoring window, look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.
- 7.** Click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

Moving Code to Subclasses

To move a member, a method's implementation, or an `implements` clause from the current class to that class' subclasses:

1. In the Source Editor or the Projects window, select the class or interface that contains the member or members that you want to move.
2. Choose Refactor | Push Down to open the Push Down dialog box.
3. Select the checkbox for each member you want to move.

If you want to leave an abstract declaration for the method in the current class and move the implementation to the subclasses, select the Keep Abstract checkbox for the method.

If the class from which you are moving members implements an interface, a checkbox for that interface is included in the dialog box. If you select the checkbox for that interface, the interface is removed from the `implements` clause of the current class and moved to the `implements` clause of the class to which you are moving members.

4. Click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

- 5.** In the Refactoring window, look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.
- 6.** Click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

Creating a Method from Existing Statements

As your code evolves, you might find it desirable to break some methods up into multiple methods. You can use the Extract Method command to simplify this process. The Extract Method command does the following:

- Creates a new method and moves the selected statements to that method.
- Adds a call to the new method in the location from where the statements were moved.

To extract a method from existing statements:

1. In the Source Editor, select the statements that you want to be extracted into the new method.
2. Right-click the selection and choose Refactor | Extract Method.
3. In the Extract Method dialog box, enter a name for the method and select an access level.
4. If you want the method to be static, select the Static checkbox.
5. Click Next.

If you left the Preview All Changes checkbox clear, the method is immediately extracted.

If you have selected the Preview All Changes checkbox, the changes to be made to your code are listed in the Refactoring window. After verifying the changes, click Do

Refactoring to complete the method extraction.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

Creating an Interface from Existing Methods

If you decide to divide your code into API and implementation layers, you can get started on that conversion by using the Extract Interface command to create an interface from methods in an existing class. The Extract Interface command does the following:

- Creates a new interface containing declarations for selected public methods.
- Adds the name of the created interface to the `implements` clause of the class from which the interface is extracted. (If the interface is extracted from another interface, the name of the newly created interface is added to the `extends` clause of the other interface.)

To extract an interface from existing methods:

1. In the Source Editor or the Projects window, select the class that contains the methods that you want to be extracted into the new interface.
2. Choose Refactor | Extract Interface.
3. In the Extract Interface dialog box, select the checkbox for each method that you want to be declared in the new interface.

If the class from which you are extracting an interface already implements an interface, a checkbox for that interface is included in the Extract Interface dialog box. If you select the checkbox for that interface, the interface is removed from the `implements` clause of the previously implementing interface and moved to the `extends` clause of

the new interface.

4. Click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

5. In the Refactoring window, look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.

6. Click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.



When you use the Extract Interface command, the interface is always created in the same package as the class from which it was extracted. If you want to move the interface to another package, you can use the Refactor | Move Class command to do so.

Extracting a Superclass to Consolidate Common Methods

As a project evolves, you might need to add levels to your inheritance hierarchy. For example, if you have two or more classes with essentially duplicate methods that are not formally related, you might want to create a superclass to hold these common methods. Doing so will make your code easier to read, modify, and extend, whether now or in the future.

You can use the Extract Superclass command to create such a superclass based on methods in one of the classes that you want to turn into a subclass. For each method that you add to the superclass, the Extract Superclass command enables you to choose between the following two options:

- Moving the whole method to the superclass
- Creating an abstract declaration for the method in the superclass and leaving the implementation in the original class

To extract a new superclass:

1. In the Source Editor or the Projects window, select the class that contains the methods that you want to be extracted into the new superclass.
2. Choose Refactor | Extract Superclass.
3. In the Extract Superclass dialog box, select the checkbox for each method and field that you want to be moved to the new superclass. Private methods and private fields are not included.

If you want to leave a method implementation in the current class and create an abstract declaration for the method in the superclass, select the Make Abstract checkbox for the method.

If the class from which you are extracting a superclass implements an interface, a checkbox for that interface is included in the Extract Superclass dialog box. If you select the checkbox for that interface, the interface is removed from the `implements` clause of the class that you are extracting from and moved to the `implements` clause of the new superclass.

4. Click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

- 5.** In the Refactoring window, look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.
- 6.** Click Do Refactoring.

If you later find that the refactoring has had some consequences you would like to reverse, you can choose Refactor | Undo.

After you have extracted the superclass, you can use a combination of the following techniques to complete the code reorganization:

- Add the name of the new superclass to the `extends` clause of any other classes that you want to extend the new

superclass.

- Use the Pull Up command to move methods from other classes to the new superclass. As with the Extract Superclass command, you can move whole methods or merely create abstract declarations for the methods in the new superclass. See [Moving Class Members to Other Classes](#) earlier in this chapter.
- For other classes that you want to extend the new superclass, use the Override and Implement Methods feature (Ctrl-I) to add methods from the superclass to those classes. See [Generating Methods to Override from Extended Classes](#) earlier in this chapter.
- Use the Use Supertype Where Possible command to change references in your code to the original class to the just created superclass. See [Changing References to Use a Supertype](#) below.

Changing References to Use a Supertype

You can use the Use Supertype Where Possible refactoring command to change code to reference objects of a superclass (or other type) instead of objects of a subclass. The operation only changes the reference in places where your code can accommodate such upcasting.

Typically you would use this refactoring operation to enable a single method to take as an argument different types of objects (all deriving from the same superclass).

This operation might be particularly useful after you have used the Extract Superclass command.

To change references to a supertype:

1. Select the class to which you want to replace references and choose Refactor | Use Supertype Where Possible.
2. In the Select Supertype to Use list, select the class or other type that should be referenced instead of the type currently referenced and click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

3. In the Refactoring window, look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.
4. Click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

Unnesting Classes

As a project grows, you might find that some classes become tangled with a dense structure of inner classes that are hard to read and which cannot be elegantly modified or extended. If this is the case, you might want to simplify the nesting structure and move some classes into their own source files.

The IDE has a pair of useful commands for simplifying your code's nesting structure:

- Move Inner to Outer Level. This command moves a class up one level.

If the class is a top-level inner class, it is made an outer class and moved into its own source file.

If the class is nested within an inner class, method, or variable scope, it is moved up one level.

- Convert Anonymous Class to Inner. This command converts an anonymous inner class (i.e., a class that is unnamed and has no constructor) into a named inner class (inner class that has a name and a constructor). This also makes it possible for other code to reference this class.

Moving an Inner Class up One Level

To move an inner class up one level:

1. In the Source Editor, right-click the inner class that you want to move and choose Refactor | Move Inner to Outer Level.

2. In the Class Name field of the Move Inner to Outer Level dialog box, set the name of the class.
3. Select the Declare Field for the Current Outer Class checkbox if you want to generate a field in the moved inner class to hold the outer class instance and include a reference to that instance as a parameter in the moved class' constructor.

If you select this option, fill in the Field Name text field with a name for the the outer class' instance field.

4. Click Next.

If you have deselected the Preview All Changes checkbox, the changes are applied immediately.

If you leave the Preview All Changes checkbox selected, the Refactoring window appears with a preview of the changes.

5. In the Refactoring window, look at the preview of the code to be changed. If there is a modification that you do not want to be made, deselect the checkbox next to the line for that change.
6. Click Do Refactoring.

If you later find that the refactoring has had some consequences that you would like to reverse, you can choose Refactor | Undo.

Unless you have selected the Preview All Changes box, the inner class is immediately moved up one level.

If you have selected the Preview All Changes box, the changes to be made are shown in the Refactoring window. You can then apply the changes by clicking Do Refactoring.

If the result is different from what you expected, you can

reverse the command by choosing Refactor | Undo.

Converting an Anonymous Inner Class to a Named Inner Class

To convert an anonymous inner class to a named inner class:

1. In the Source Editor, right-click the anonymous inner class that you want to convert and choose Refactor | Convert Anonymous Class to Inner.
2. In the Inner Class Name field of the Convert Anonymous Class to Inner dialog box, enter a name for the class.
3. In the Access field, select the access modifier for the class.
4. Select the Declare Static checkbox if you want the class to be static.
5. In the Constructor Parameters list, use the Move Up and Move Down buttons to set the order of the parameters.
6. Click Next.

Unless you have selected the Preview All Changes box, the anonymous class is converted to the named inner class.

If you have selected the Preview All Changes box, the changes to be made are shown in the Refactoring window. You can then apply the changes by clicking Do Refactoring.

If the result is different from than expected, you can reverse the command by choosing Refactor | Undo.

Tracking Notes to Yourself in Your Code

The IDE has a task list feature that provides a way for you to write notes in your code and then view all of these notes in a single task (or "to do") list. You can use the task list as the center of operations when cleaning up loose ends in your code.

A line is displayed in the task list if it is "tagged" with (contains) any of the following text:

- @todo
- TODO
- FIXME
- XXX
- PENDING
- <<<<<<



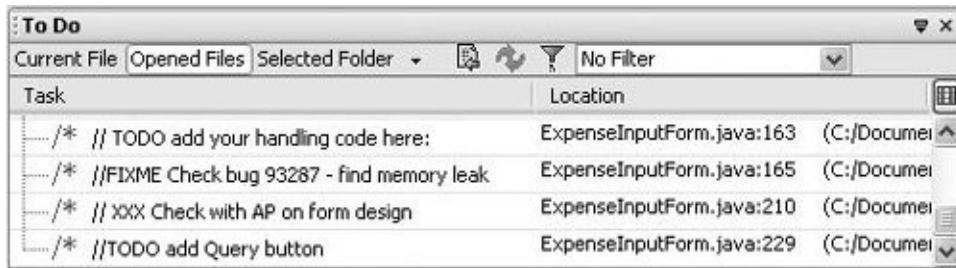
When you type a tag in your code, it must be typed as a whole word for the IDE to recognize it. For example, if you do not put a space between the tag and the note, the note will not appear in the task list.

To view the task list, choose Window | To Do (or press Ctrl-6).

Once you have displayed the To Do window (shown in [Figure 5-](#)

[18](#)), you can view tasks for the current file, for all open files, or for a specific folder by clicking the corresponding button at the top of the To Do window.

Figure 5-18. To Do window



You can sort task-list items by task, location, or priority by clicking the corresponding column titles. See [Displaying Tasks by Priority](#) later in this chapter for information on displaying the Priority column.

You can jump from an entry in the task list straight to the line in the code where you wrote the note by double-clicking the entry.

Adding, Removing, and Changing Task-List Tags

To change the tags that are used for the task list:

1. Choose Tools | Options, click Advanced Options, and select the Editing | To Do Settings node.
2. Click the button in the Task Tags property.
3. In the To Do Settings dialog box, use the Add, Change, and Delete buttons to modify the contents of the Task List table.

Displaying Tasks by Priority

You can also display priorities for each task-list item. The available priorities are High, Medium-High, Medium, Medium-Low, and Low.

By default, the Priority column is not displayed. You can display the Priority column by clicking the  icon and selecting the Priority checkbox in the Change Visible Columns dialog box.

The priority values can be assigned by tag. By default, all tags are assigned Medium priority except the <<<<< tag, which is given High priority.

To change a priority value for a tag:

1. Choose Tools | Options, click Advanced Options, and select the Editing | To Do Settings node.
2. Click the  button in the Task Tags property.
3. In the To Do Settings dialog box, select the new priority in the combo box in the Priority column for the tag that you want to change.

Filtering Task-List Entries

You can further limit the entries displayed in the task list by creating and using filters. When you use a filter, only entries that match criteria specified by the filter are displayed. Criteria include text that needs to appear in the note, the priority of the task, and/or the filename.

To create a filter:

1. Click the  icon in the To Do window's toolbar.

- 2.** In the Edit Filters dialog box, click the New button and then type a name for the filter in the Name field.
- 3.** Fill in the details for the criterion.
- 4.** Optionally, add additional criteria by clicking the More button and then filling in the details for the filters. You can select to have the filter match all or any of the criteria using the radio buttons at the top of the dialog box.

An entry for the newly defined filter appears in a combo box in the To Do Window toolbar.

Comparing Differences Between Two Files

You can generate a side-by-side comparison of two files with the differing lines highlighted. To compare two files, select the nodes for the two files in the Projects window and choose Tools | Diff.

The "diff" appears as a tab in the Source Editor.



The Diff command appears in the Tools menu only when two (and no more than two) files are selected in the Projects, Files, or Favorites window.

Splitting the Source Editor

You can split the Source Editor to view two files simultaneously or to view different parts of the same file.

To split the Source Editor window:

- 1.** Make sure at least two files are already open.
- 2.** Click a tab on one file; hold down the mouse button; and drag the tab to the far left, far right, or bottom of the Source Editor window.
- 3.** Release the mouse button when the red outline that appeared around the tab when you started dragging changes to a rectangle indicating the placement of the split window.

To view different parts of the same file simultaneously:

- 1.** Click the file's tab in the Source Editor and choose Clone Document to create a second tab for the same document.
- 2.** Drag and drop one of the file tabs to create a split Source Editor area. (See the procedure above for info on dragging and dropping Source Editor tabs.)

Maximizing Space for the Source Editor

There are a number of things you can do to make more space for your code in the IDE, such as:

- Maximize a file in the Source Editor within the IDE by double-clicking that file's tab. When you do this, the file takes the entire space of the IDE except for the main menu and row of toolbars. You can make the other windows reappear as they were by double-clicking the tab again.
- Make other windows "sliding" so that they appear only when you click or mouse over a button representing that window on one of the edges of the IDE. You can make a window sliding by clicking its  icon. You can return the window to its normal display by clicking the  button within the sliding window. See [Managing IDE Windows](#) in [Chapter 2](#) for information on working with windows in the IDE.
- Hide the IDE's toolbars. You can toggle the display of the main toolbars by choosing View | Toolbars and then individually choosing the toolbars that you want to hide (or display). You can toggle the display of the Source Editor's toolbar by choosing View | Show Editor Toolbar.

Changing Source Editor Keyboard Shortcuts

You can change existing keyboard shortcuts or map other available commands to shortcuts.

To add a keyboard shortcut for a command:

1. Choose Tools | Options and click the Keymap panel.
2. In the Actions panel, navigate to a command that you want to change, and click Add.
3. In the Add Shortcut dialog box, type in the key combination that you want to use and click OK.



The IDE also comes with keyboard shortcut profiles for the Eclipse IDE and Emacs, either of which you can select from the Profiles drop-down box in the Keymap panel. You can also create your own profiles.

Chapter 6. Building Java Graphical User Interfaces

- [Using Different Layout Managers](#)
- [Placing and Aligning a Component in a Form](#)
- [Setting Component Size and Resizability](#)
- [Setting Component Alignment](#)
- [Specifying Component Behavior and Appearance](#)
- [Generating Event Listening and Handling Methods](#)
- [Customizing Generated Code](#)
- [Previewing a Form](#)
- [Using Custom Beans in the Form Editor](#)
- [Deploying GUI Applications Developed with Matisse](#)

ONE OF THE AREAS IN WHICH NETBEANS IDE PROVIDES THE MOST INNOVATIVE SUPPORT is for creating Java clients using Java Foundation Classes (JFC or "Swing") and AWT packages. In addition to support for building applications on top of the NetBeans Platform, NetBeans IDE also provides cutting-edge tools to design individual forms, components, and dialogs.

In the 5.0 release, NetBeans IDE revolutionized the approach to GUI layout management by introducing Project Matisse the

combination of a new layout manager called GroupLayout and a visual interface used to design forms with this layout manager.

The Matisse approach arranges components using aligning and relative proximity rather than nested containers and absolute X and Y coordinates. Components are anchored to the sides of a container and to other components. You can set two components to, for example, be aligned along the baseline or along the left or right edge. When the size of components changes, either when the form is localized or when the user resizes the form, all of the components remain aligned correctly.

Another advantage of the Matisse approach is that the default spacing between components is defined by the look and feel of the program rather than in absolute pixels. This means that your form will always look like a native application on an operating system.

NetBeans IDE includes a number of tools for working with Java GUI forms, including:

- **Form Editor.** Provides both Design and Source views for you to create visual components. The Design view is an area where you can drag, drop, and rearrange the visual components that make up the user interface of the client you are building. The Source view contains the generated source code for the class you are designing and also allows you to enter your own code for the class.
- **Inspector window.** Provides a tree view of all of the components in the form, whether visual (such as menus, text fields, labels, and buttons) or nonvisual (such as button groups and data sources). This window appears in the same space as the Navigator window.
- **Palette window.** Provides a list of components that you

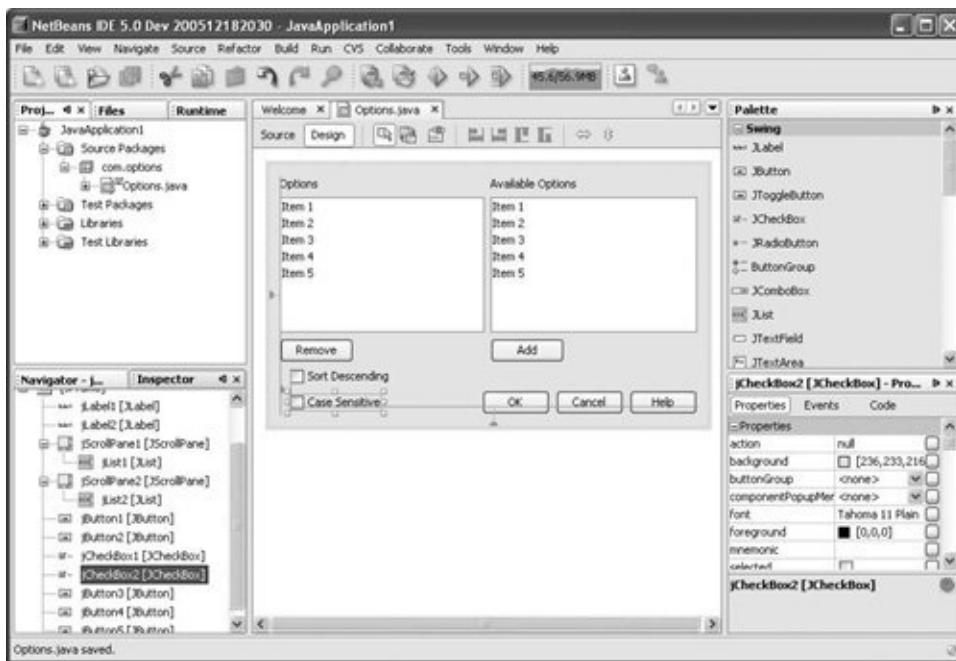
can drag and drop onto your form. You can choose from Swing and AWT components or add your own custom beans.

- **Properties window.** Contains a list of editable properties for the selected component and access to special property editors for the more complex properties.
- **Connection wizard.** Helps you create event listener and event handler code that links two components.
- **Form Tester.** Quickly displays a runtime view of the form under construction, allowing checks of resizing and other behavior.
- **Palette Manager.** Enables you to add custom components to the Palette window. Choose Tools | Palette Manager | Swing/AWT Components to open this window.

The Form Editor, Inspector window, Palette window, and Properties window are shown in [Figure 6-1](#). You can access the Connection wizard and Form Tester from buttons in the toolbar area of the Form Editor.

Figure 6-1. Form Editor windows, including the Inspector, Form Editor (Design View), Palette window, and Properties window

[\[View full size image\]](#)



This section does not provide a complete guide to developing visual applications with Swing in NetBeans IDE; a whole book could be devoted to that. Instead, it focuses on a few of the unique but somewhat tricky features of Swing and the IDE that assist in the designing of visual applications.

Using Different Layout Managers

By default, NetBeans IDE uses the GroupLayout layout manager for any new forms that you create. This chapter deals exclusively with building Java GUIs with Matisse, which is by far the best way to develop Java GUIs. If, however, you need or prefer to work with other standard layouts, the IDE provides full support for that as well. Actions such as placing components in a form and working with component properties work the same as when working with Matisse. Only component alignment and resizing behavior differs and is controlled by the layout manager you are using.

The IDE provides full support for all of the standard Swing layouts managers:

- FlowLayout
- BorderLayout
- GridLayout
- GridBagConstraints
- CardLayout
- BoxLayout (note that the "struts" and "glue" normally used with BoxLayout are not provided as out-of-the-box components)

In addition, support for forms without a layout manager (NullLayout) and forms with absolutely positioned components (AbsoluteLayout) is provided. Neither of these is recommended for production use, as their behavior across platforms or when

resizing is unlikely to be acceptable.

To set the layout manager for a container:

1. Right-click a container in the Inspector window and choose the layout manager from the Set Layout menu.
2. To set the form to use a layout manager, choose the layout manager name. To set the form to use GroupLayout (Matisse), choose Free Design.



To set the layout manager back to GroupLayout, choose Set Layout | Free Design.

Tips for GUI Editing

The most important thing to remember when you begin designing a GUI is that whenever you place a component into a form, you create spacing and alignment relationships that can be tricky to change later. You therefore should never just start throwing components into a form and then try to arrange them into the desired layout.

Instead, it is good to start out with a clear idea of how you want your form to look. Sketch the form on a piece of paper or find an example of a similar form that you want to imitate.

As you work, start in the top-left corner, fill out each row, and work your way down. Use the recommended spacing and alignment that the Form Editor suggests as much as possible to guarantee that your form will look right on all operating systems.

Of course, at some point you will have to modify existing forms. Here are a few tips that help modify your forms without tearing your hair out:

- If you have to move components that are grouped, like a row of labels and text fields, Ctrl-click the components to select them all and move them as a unit. This helps preserve the grouping relationship between the components.
- If you get into trouble, use the Undo command to revert to your previous state and start over. Often, trying to correct the error yourself only makes things worse.

Placing and Aligning a Component in a Form

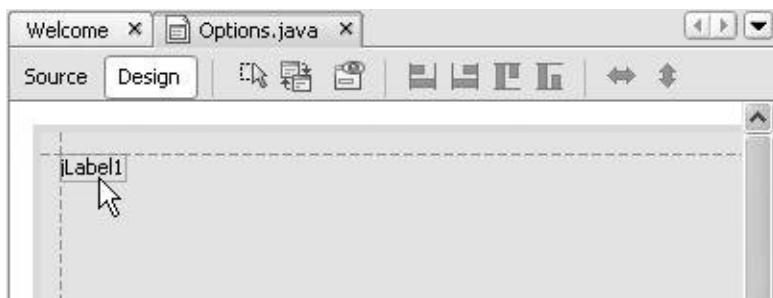
To design forms with Matisse, you drag components from the Palette into your form. As you drag the component close to other components or the container edges, the component automatically snaps to align with other components.

To add a component to a form:

1. Click a component in the Palette to select it.
2. Move the cursor to the location in the form where you want to place the component. The IDE suggests alignment and anchoring as you near other components.
3. Click to place the component. If you want to enter multiple components, hold down the Shift key and click multiple times.

For instance, in [Figure 6-2](#), the first JLabel is anchored to the top and left edges of the JFrame, with the default spacing defined by the look and feel.

Figure 6-2. JLabel snapped to the top-right corner of a JFrame



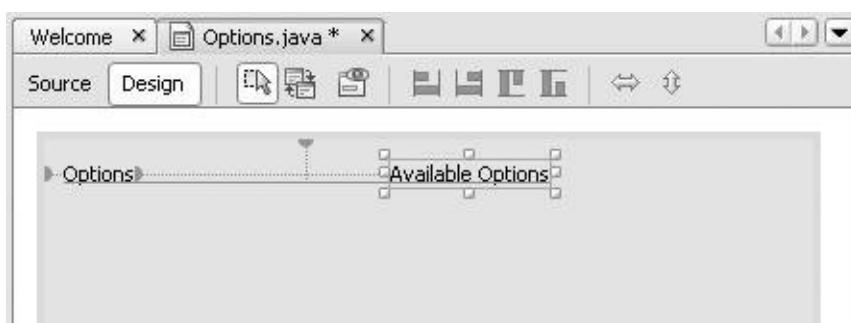
In [Figure 6-3](#), JLabel2 is snapped to the baseline of the Options JLabel.

Figure 6-3. JLabel snapped to the baseline of another JLabel



When you've placed the components, you can select any component to see what it is anchored to, as shown in [Figure 6-4](#). The ▶ graphic shows the component or container edge to which a component is anchored.

Figure 6-4. Guidelines showing the anchoring relationships of the two JLabel components





You can edit the display text of any JLabel, JTextField, or JButton by double-clicking it in the Design view.

Setting Component Size and Resizability

To set a component's size, grab one of its corners or edges and drag it to the desired size. Many components automatically expand in size to accommodate any text and icons that are inserted into them.

Another important function is setting components to be the same size. For example, a form could have the standard OK, Cancel, and Help buttons. By default, each button would only be as wide as its containing text. The form looks much better, however, when you set all three buttons to be the same width.

To set multiple components to be the same size:

1. Shift-click the components to select them.
2. Right-click any of the selected components and choose Same Size | Same Width/Same Height.

To set a component to automatically resize as its container resizes, do one of the following:

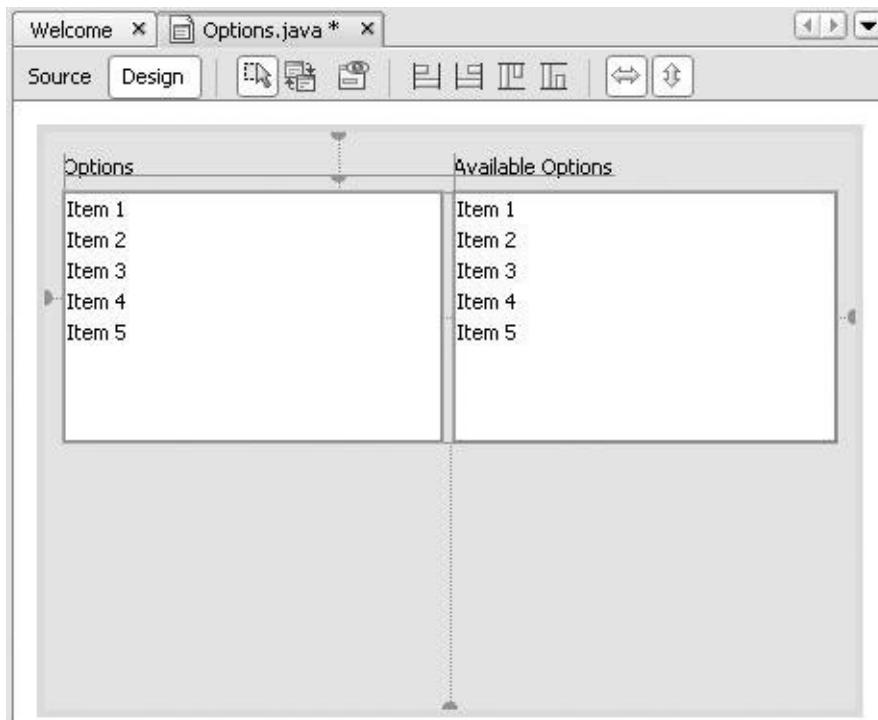
- Resize the component to snap to the edges of its container.
- Use the  and  buttons in the Form Designer toolbar.

[Figure 6-5](#) shows all of these concepts at work together. The two JLists must have the same width and automatically resize both vertically and horizontally when you resize the dialog. To achieve this state, do the following:

1. Insert two JLists, each snapped to the left edge of the JLabel above them.

- 2.** Expand the JList on the left to snap to the side of the JList on the right.
- 3.** Expand the JList on the right to snap to the edge of the JFrame.
- 4.** Select both JLists by shift-clicking them.
- 5.** Right-click either of the JLists and choose Same Size | Same Width.
- 6.** Click the and buttons to set the resizing behavior.

Figure 6-5. Two JList components set to be the same size and resize automatically



Note that you cannot set the resizing behavior for

components that are set to be the same size. In the example in [Figure 6-5](#), you set the forms to be the same initial size when you used the Same Width command. When you set the resizing behavior, however, you broke the Same Width link between the two lists. Because they are both set to resize both vertically and horizontally, they will retain their same size automatically.

Setting Component Alignment

When you snap a component to an edge of another component or the side of the a container, you also set the component's alignment. For example, in [Figure 6-6](#), the JLabel is right-aligned to the edge of the JTextField. As you enter longer text, the JLabel expands to the left.

Figure 6-6. JLabel right-aligned to a JTextField's edge



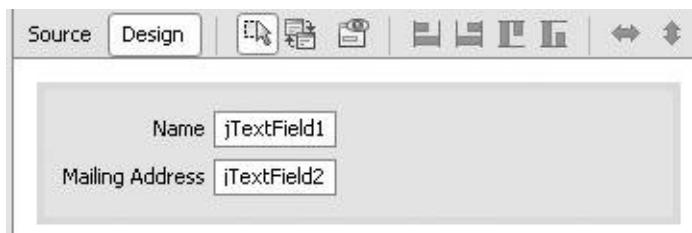
In some cases, you have to manually adjust the alignment of components. Consider the form in [Figure 6-7](#). The two labels were entered first, and their display text set. The text fields were then entered and snapped to the right edges of the labels.

Figure 6-7. Form before realigning the JLabels



We want to make the JLabels right-aligned so the two JTextFields line up along their left edges. To do so, shift-click the two JLabels to select them and click the  button in the Form Designer toolbar. The result is shown in [Figure 6-8](#).

Figure 6-8. Form after realigning the JLabels



Specifying Component Behavior and Appearance

You can use the Properties window to set the behavior and appearance of components that you have added to a form. The Properties window displays the properties of the component that is selected in the Inspector or the Form Editor. The properties come in three categories:

- **Properties.** A configurable list of characteristics for the component. Technically speaking, these are the JavaBeans properties, layout constraints, and accessibility properties for the component.
- **Events.** A list of event listeners that you can attach to a component. You can specify event listeners here (or remove them here) or use the Connection wizard. See [Generating Event Listening and Handling Methods](#) later in this chapter.
- **Code.** Some NetBeans IDE-specific properties you can use to customize the way the code is generated. See [Customizing Generated Code](#) later in this chapter.

To edit component properties:

1. Select the property category by clicking the appropriate button at the top of the Properties window (Properties, Events, or Code).
2. Edit the component's properties in the Properties window by selecting the property and entering the desired value.
3. If a property has a  button, you can click it to open a special property editor that enables you to modify the

property and the initialization code generated for it.

4. In the property editor, use the Select Mode combo box to choose each custom editor for the property and make the necessary changes.



The Properties window lists any properties that have been changed from their default values in bold. Many components come with preset properties when you place them in a form. Often, these are the most important properties for the component.

Generating Event Listening and Handling Methods

The IDE relieves you of the task of providing the infrastructure required around event handling by generating the code to link the occurrence of the event with the invocation of a private method in the form class. For example, a JButton named `myBtn` might have the code

```
myBtn.addActionListener(new java.awt.event.ActionListener()
    public void actionPerformed(java.awt.event.ActionEvent evt)
        myBtnActionPerformed(evt);
    }
```

added to its initialization, where the method `myBtnActionPerformed` is generated as:

```
private void myBtnActionPerformed(java.awt.event.ActionEvent evt)
    // TODO add your handling code here:
}
```



Comments generated or otherwise within your Java code that start with "TODO" have special significance. To see a list of your "TODO" lines, display the To Do window via Window | To Do or press Ctrl-6. From the displayed window, you can navigate to the source line with the TODO with a double-click in the To Do window. See Tracking Notes to Yourself in Your Code in [Chapter 5](#) for more information.

Generation of this event infrastructure code can be done in a few different ways:

- By right-clicking the component and choosing the event to be handled from the Events menu.

The IDE generates the event handler and positions the cursor to the appropriate TODO line in the generated private method for completion of the event handling code.

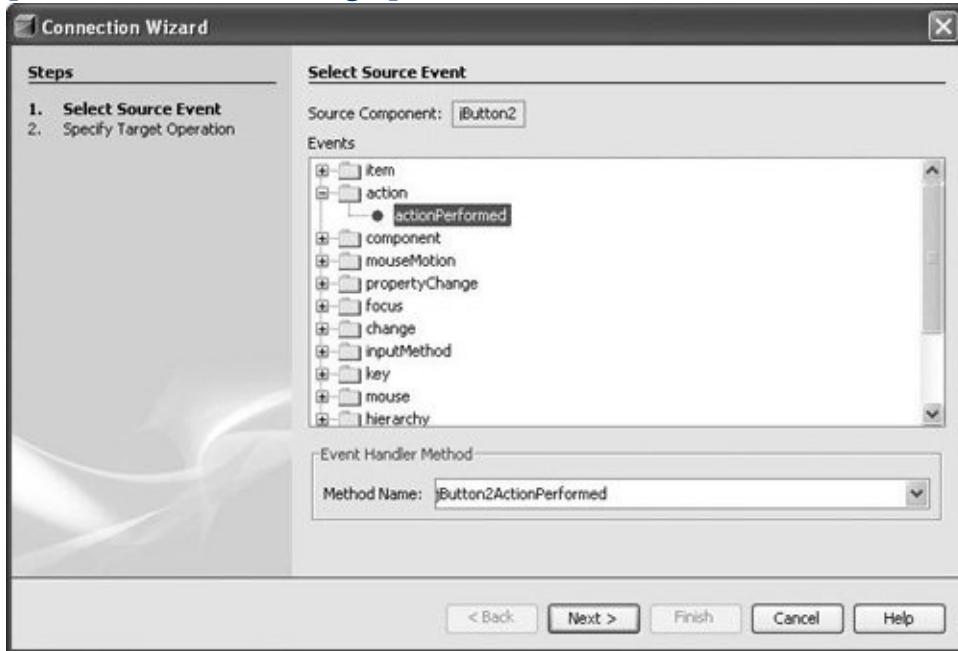
- By selecting the component and specifying the event to be handled in the Events tab of the Properties window.
- By using the Connection Wizard to generate code for the case when an event on a component should result in the modification of another component.

To use the Connection wizard:

1. Enter "connection mode" by clicking the  icon in the Form Editor's toolbar.
2. Open the Connection Wizard by clicking successively on the two components first the component that will fire the event and then the component upon which an operation is to be performed.
3. In the Select Source Event page of the wizard (shown in [Figure 6-9](#)), select the event to be fired.

Figure 6-9. Connection Wizard, Select Source Event page

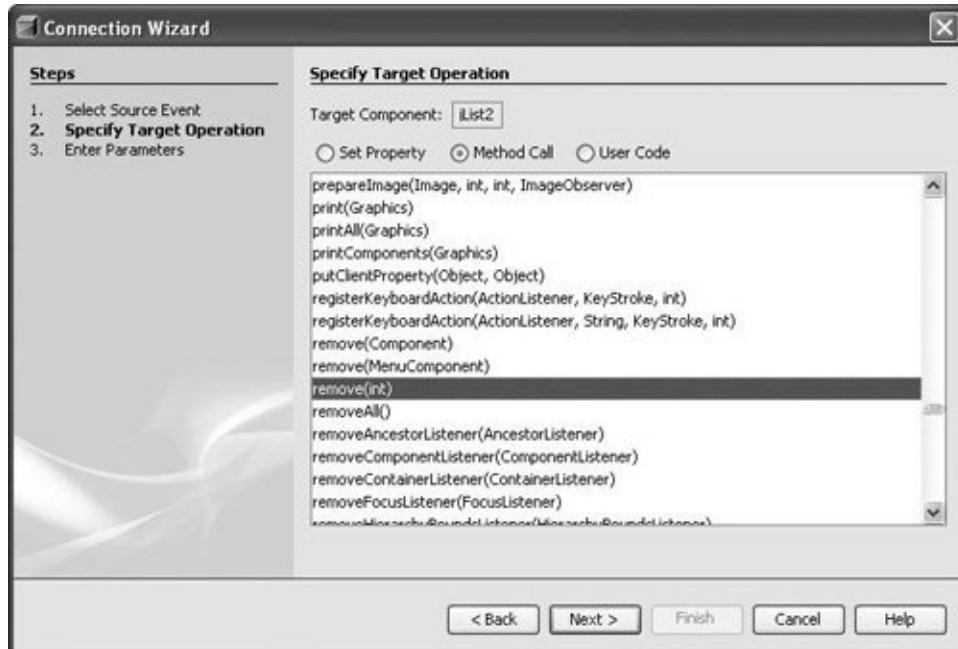
[\[View full size image\]](#)



4. In the Specify Target Operation page (shown in [Figure 6-10](#)), specify the operation to be performed on the target component. You can specify a property to set, call a method, or write your own custom code.

Figure 6-10. Connection Wizard, Specify Target Operation page

[\[View full size image\]](#)



The Connection Wizard approach is simply a "point and click" approach to the task. The code generated by the wizard is not guarded and can be modified in the editor after generation.

Customizing Generated Code

NetBeans IDE dynamically generates the code for GUI construction. You can view this code in the Source view of the Form Editor (click the Source button in the Form Editor's toolbar). In addition to the code generated within the class in its `class-name.java` file, the IDE maintains an XML file called `class-name.form` that details the structure of the form. Note that the source-code control systems (such as CVS) supported by NetBeans ensure by default that the `.form` file is maintained in the repository in addition to the `.java` file.

The generated code within the `.java` source file is delimited by special comments (for example, `//GEN-BEGIN:initComponents`
`...//GEN-END:initComponents`). The editor does not allow this code to be modified and indicates the unmodifiable code with a pale blue background. Although you could modify this code outside the IDE, it is not recommended, because those modifications would be lost if you reopened the form in the IDE. (The IDE regenerates the `.java` file of a form created in the IDE from the `.form` files each time you open the file in the IDE.)

The use of delimited generated code prompted vigorous discussion in the NetBeans IDE team, but the advantages are significant: It is extremely difficult to reliably "reverse-engineer" arbitrary Swing code without requiring restrictive coding discipline on the developer's part.

Instead, NetBeans IDE provides "hooks" where you can add (almost) any arbitrary code to be part of the code to be generated. This code is added via a codeaware window accessed from the Code tab of the Properties window for the component. The properties used are:

- **Custom Creation Code.** Code to be inserted instead of the

default `newComponentClassName()`; statement.

- **Pre-Creation Code.** One or more lines of code to precede the statement that instantiates the component.
- **Post-Creation Code.** One or more lines of code to follow the statement that instantiates the component.
- **Pre-Init Code.** One or more lines of code to precede the first statement that initializes the properties of the component.
- **Post-Init Code.** One or more lines of code to follow the last statement that initializes the properties of the component.

In addition, the initial values of the various properties of components can be specified in various ways:

- A static value.
- A property from a component written to the JavaBeans architecture.
- A property of another component on the form.
- A call to a method of the form or one of its components. You can choose from a list of methods that return the appropriate data type.
- Code you define, which will be included in the generated code.

Previewing a Form

You can quickly preview any form without having to compile and run your project. Just click the  button in the Form Editor toolbar. Although you can resize the form, type text into text fields, and otherwise manipulate the form, the form is not "live." You have to run the project to test any event handling code.

Using Custom Beans in the Form Editor

In addition to the standard Swing and AWT components available in the palette, you can also build your own custom components and add them to the palette.

To add a component to the palette:

1. If you are developing the component palette in an IDE project, build the project.
2. Choose Tools | Palette Manager | Swing/AWT Components.
3. Click one of the following to specify the component location:

Add from JAR. Choose this option if the component is in a built library.

Add from Library. Choose this option if the component is in one of the libraries registered in the Library Manager.

Add from Project. Choose this option if the form is in an IDE project. If you choose this option, the IDE adds the project containing the component to the present project's classpath. Whenever you build your project, the component's project is built as well.

4. Click Next. The wizard displays all JavaBeans components in the specified location.
5. Select the component and click Next.
6. Specify the palette category for the component and click Finish.



If the project containing the component is open in the IDE, you can right-click the component's node in the

Projects window and choose Tools | Add to Palette.

Associating an Icon with a Component

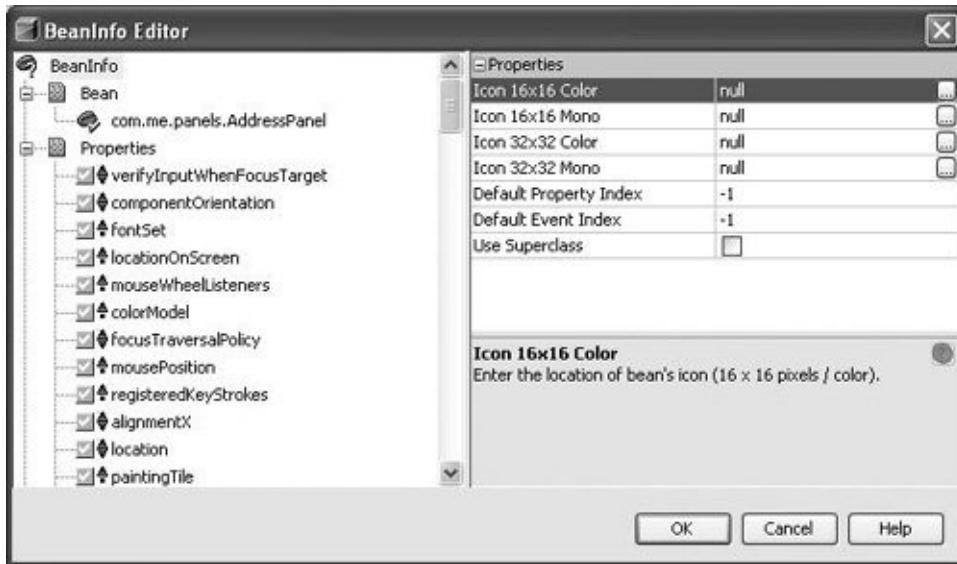
If you develop a component that you want to add to the Palette window, you can easily associate an icon with the component. The icon is then displayed with the component in the Palette window.

To associate an icon with a component:

1. In the Projects window, expand the class node that contains your component.
2. Right-click the Bean Patterns subnode and choose BeanInfo Editor from the pop-up menu.
3. In the BeanInfo Editor, select the BeanInfo node. Properties are displayed in the right panel for all four icon types, as shown in [Figure 6-11](#).

Figure 6-11. The BeanInfo Editor for a component

[\[View full size image\]](#)



4. Use the icon properties to set the small and large icons for the component.

Deploying GUI Applications Developed with Matisse

As of Java Platform Standard Edition 5.0, the Matisse binary is not included in the standard Java distribution. This means that you have to package the JAR file containing the GroupLayout classes with your application. The JAR file is located in `NetBeans_installation_folder/platform6/modules/ext/swinglayout installed_version.jar`.

Whenever you build a project with a main class, the IDE automatically copies all of the JAR files on the classpath, including the Swing Layout Extensions JAR file, to the project's `dist/lib` folder. The IDE also configures the classpath for the application JAR file in the manifest file. No further configuration is necessary. See [Chapter 3](#) for more information.

Chapter 7. Debugging Java Applications

- [Starting a Debugging Session](#)
- [Debugger Windows](#)
- [Attaching the Debugger to a Running Application](#)
- [Starting the Debugger Outside of the Project's Main Class](#)
- [Stepping through Code](#)
- [Setting Breakpoints](#)
- [Managing Breakpoints](#)
- [Customizing Breakpoint Behavior](#)
- [Monitoring Variables and Expressions](#)
- [Backing up from a Method to Its Call](#)
- [Monitoring and Controlling Execution of Threads](#)
- [Fixing Code During a Debugging Session](#)
- [Viewing Multiple Debugger Windows Simultaneously](#)

NETBEANS IDE PROVIDES A RICH ENVIRONMENT for troubleshooting and optimizing your applications. Built-in debugging support allows you to step through your code incrementally and monitor aspects of the running application,

such as values of variables, the current sequence of method calls, the status of different threads, and the creation of objects.

When using the IDE's debugger, there is no reason for you to litter your code with `System.out.println` statements to diagnose any problems that occur in your application. Instead, you can use the debugger to designate points of interest in your code with breakpoints (which are stored in the IDE, not in your code), pause your program at those breakpoints, and use the various debugging windows to evaluate the state of the running program.

In addition, you can change code while debugging and dynamically reload the class in the debugger without having to restart the debugging session.

Following are some of the things that you can do within the IDE's debugger:

- Step through application code line by line.
- Step through JDK source code.
- Execute specific chunks of code (using breakpoints as delimiters).
- Suspend execution when a condition that you have specified is met (such as when an iterator reaches a certain value).
- Suspend execution at an exception, either at the line of code that causes the exception or in the exception itself.
- Track the value of a variable or expression.
- Track the object referenced by a variable (fixed watch).

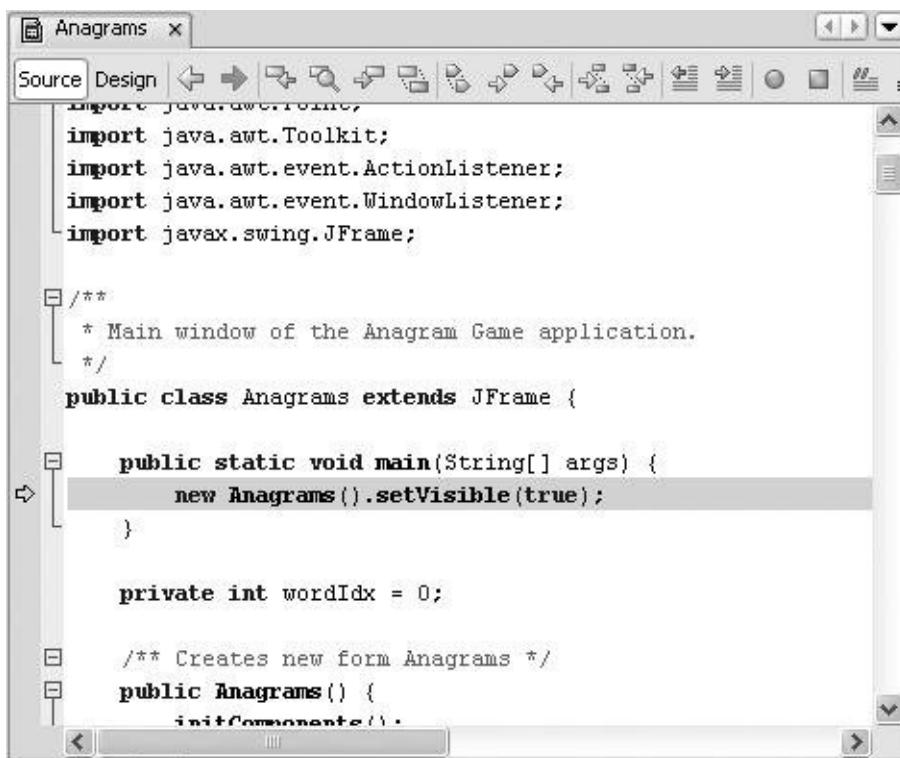
- Fix code on the fly and continue the debugging session with the Apply Code Changes command.
- Suspend threads individually or collectively.
- Step back to the beginning of a previously called method (pop a call) in the current call stack.
- Run multiple debugging sessions at the same time. For example, you might need this capability to debug a client-server application.

To analyze the performance of your application in more detail, use the IDE's profiler, described in [Chapter 15](#).

Starting a Debugging Session

The simplest way to start using the debugger is to choose Run | Step Into. The program counter (marked by green background highlighting and the icon, as shown in [Figure 7-1](#)) stops one line into the main method of your main project.

Figure 7-1. A suspended program with the green program counter showing the next line to be executed



You can then step through your code incrementally with any of the Step commands to observe the program flow and monitor the evolving values of variables in the Local Variables window.

See [Stepping through Code](#) later in this chapter for a description of all the Step commands and the ensuing topics for information on how to take advantage of the debugger's capabilities.



You can also use the Run to Cursor command to start a debugging session. In the Source Editor, click in the line where you want execution to suspend initially and choose Run | Run to Cursor. This command works for starting a debugging session only if you select a line of code in the project's main class or a class directly called by the main class in the main project.

More likely, you will want to start stepping through code at some point after the start of the main method. In this case, you can specify some point in the program where you want to suspend the debugged execution initially and then start the debugger. To do so:

1. Set a line breakpoint in your main project by opening a class in the Source Editor and clicking in the left margin next to the line where you want to set the breakpoint (or by pressing Ctrl-F8).

You know that the breakpoint has been set when the pink □ glyph appears in the margin and the line has pink background highlighting (as shown in [Figure 7-2](#)).

Figure 7-2. Code in the Source Editor with a debugger breakpoint set

```
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.event.ActionListener;
import java.awt.event.WindowListener;
import javax.swing.JFrame;

/**
 * Main window of the Anagram Game application.
 */
public class Anagrams extends JFrame {

    public static void main(String[] args) {
        new Anagrams().setVisible(true);
    }

    private int wordIdx = 0;

    /** Creates new form Anagrams */
    public Anagrams() {
        initComponents();
    }
}
```

2. Press F5 to start debugging the main project.

When the execution of the program stops at the breakpoint (which you can see when the pink breakpoint highlight is replaced by the green highlight of the program counter), you can step through the code line by line while viewing the status of variables, threads, and other information.

See the ensuing topics for details on stepping and viewing program information.



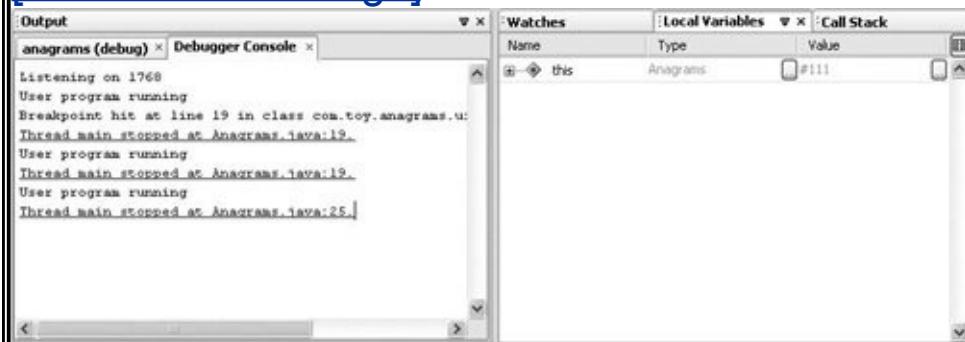
If you have set up a free-form project, you need to do some extra configuration to get the debugging commands to work. See [Chapter 16](#) for more details.

Debugger Windows

When you start debugging a program, the Debugger Console appears as a tab in the lower-left corner of the IDE (as shown in [Figure 7-3](#)). The Debugger Console logs the execution status of the debugged program (such as whether the code is stopped at a breakpoint). In addition, a tab opens in the Output window to log any application output (as well as the output from the Ant build script the IDE uses when running the command).

Figure 7-3. Windows that appear when you start debugging in the IDE, including the Debugger Console, and windows for Watches, Local Variables, and the Call Stack

[[View full size image](#)]



In the lower-right corner, several windows (Watches, Local Variables, and Call Stack) open as tabs and provide current information on the debugging session, such as the current values of variables and a list of current method calls. You can also open individual debugging windows by choosing them from the Windows | Debugging menu.

Most of the windows display values according to the debugger's current *context*. In general, the current context corresponds to one method call in one thread in one session. You can change the context (for example, designate a different current thread in

the Threads window) without affecting the way the debugged program runs.

See [Table 7-1](#) for a list of all of the windows available and how to open them.

Table 7-1. Debugger Windows

Debugger Window	Open With	Description
Local Variables	Alt-Shift-1 (or Window Debugging Local Variables)	Displays all fields and local variables in the debugger's current context and their current values. Fields are listed under the <code>this</code> node.
Watches	Alt-Shift-2 (or Window Debugging Watches)	Displays the names of fields, local variables, or expressions that you have placed a watch on. Although all of your watches are displayed no matter the current context, the value displayed is the value for that context (not for the context that the watch was set in). For example, if you have a watch on the <code>this</code> keyword, the <code>this</code> referred to in the Watches window will always correspond to the object referred to from the current method call.
Call Stack	Alt-Shift-3 (or Window Debugging Call Stack)	Displays all method calls in the current chain of calls. The Call Stack window enables you to jump directly to code of a method call, back up the program's execution to a previous method call, or select a context for viewing local variable values.
Classes	Alt-Shift-4 (or Window Debugging Classes)	Provides a tree view of classes for the currently debugged application grouped by classloader.

Breakpoints	Alt-Shift-5 (or Window Debugging Breakpoints)	Displays all breakpoints you have set in all running debugging sessions.
Threads	Alt-Shift-6 (or Window Debugging Threads)	Displays the threads in the current debugging session. In this window, you can switch the context by designating another thread as the current thread.
Sessions	Alt-Shift-7 (or Window Debugging Sessions)	Displays a node for each debugging session in the IDE. From this window, you can switch the current session.
Sources	Alt-Shift-8 (or Window Debugging Sources)	Displays sources that are available for debugging and enables you to specify which ones to use. For example, you can use this window to enable debugging with JDK sources.

Attaching the Debugger to a Running Application

If you need to debug an application that is running on another machine or is running in a different virtual machine, you can attach the IDE's debugger to that application:

1. Start in debug mode the application that you are going to debug. This entails adding some special arguments to the script that launches the application.

For Windows users using a Sun JDK, the argument list might look like the following (all in one line and no space after `-Xrunjdwp:`):

```
java -Xdebug -Xnoagent -Djava.compiler=NONE  
-Xrunjdwp:transport=dt_shmem,server=y,address=MyApp  
-classpath C:\my_apps\classes mypackage.MyApp
```

On other operating systems (or on a Windows machine when you are debugging an application running on a different machine), the argument list might look something like the following:

```
java -Xdebug -Xnoagent -Djava.compiler=NONE  
-Xrunjdwp:transport=dt_socket,server=y,address=8888  
-classpath HOME/my_apps/classes mypackage.MyApp
```

See [Table 7-2](#) for a key to these options. For more complete documentation of the options, visit <http://java.sun.com/products/jpda/doc/conninv.html>.

Table 7-2. Debugger Launch Parameters

Launch

Parameter or Subparameter	Description
<code>-Xdebug</code>	Enables the application to be debugged.
<code>-Xnoagent</code>	Disables the <code>sun.tools.debug</code> agent so that the JPDA debugger can properly attach its own agent.
<code>-Djava.compiler=NONE</code>	Disables the JIT (Just-In-Time) compiler.
<code>-Xrunjdwp</code>	Loads the reference implementation of the Java Debug WireProtocol, which enables remote debugging.
<code>transport</code>	Name of the transport to be used when debugging the application. The value can be <code>dt_shmem</code> (for a shared memory connection) or <code>dt_socket</code> (for a socket connection). Shared memory connections are available only on Windows machines.
<code>server</code>	If this value equals <code>n</code> , the application attempts to attach to the debugger at the address specified in the <code>address</code>

subparameter. If this value equals *y*, the application listens for a connection at this address.

address

For socket connections, specifies a port number used for communication between the debugger and the application. For shared memory connections, specifies a name that refers to the shared memory to be used. This name can consist of any combination of characters that are valid in filenames on a Windows machine except the backslash. You use this name in the Name field of the Attach dialog box when you attach the debugger to the running application.

suspend

If the value is *n*, the application starts immediately. If the value is *y*, the application waits until a debugger has attached to it before executing.

2. In the IDE, open the project that contains the source code for the application to be debugged.
3. Choose Run | Attach Debugger.

4. In the Attach dialog box, select the connector from the Connector combo box.

Choose SharedMemoryAttach if you want to attach to an application that has been started with the `dt_shem` transport. Choose SocketAttach if you want to attach to an application that has been started with the `dt_socket` TTransport.

See the Connector row of [Table 7-3](#) for information on the different types of connectors.

Table 7-3. Attach Dialog Box Fields

Field	Description
Connector	Specifies the type of JPDA connector to use. On Windows machines, you can choose between shared memory connectors and socket connectors. On other systems, you can only use a socket connector.
Variant	For both shared memory connectors and socket connectors, there are Attach and Listen variants. You can use an Attach connector to attach to a running application.
Host	You can use a Listen connector if you want the running application to initiate the connection to the debugger. If you use the Listen connector, multiple applications running on different JVMs can

connect to the debugger.

Transport	Specifies the JPDA transport protocol to use. This field is automatically filled in according to what you have selected in the Connector field.
Host	(Only for socket attach connections.) The host name of the computer on which the debugged application is running.
Port	(Only for socket connections.) The port number that the application attaches to or listens on. You can assign a port number in the <code>address</code> subparameter of the <code>Xrunjdwp</code> parameter that you pass to the JVM of the application that is to be debugged. If you do not use this suboption, a port number is assigned automatically, and you can determine the assigned port number by looking at the output of the process.
Timeout	The number of seconds that the debugger waits for a connection to be established.
Name	(Only for shared memory connections.) Specifies the shared

memory to be used for the debugging session. This value must correspond to the value of the `address` subparameter of the `Xrunjdwp` parameter that you pass to the JVM of the application that is to be debugged.

Local Address	(Only for socket listen connections.) The host name of the computer that you are running on.
---------------	--

- Fill in the rest of the fields. The fields that appear after Connector depend on the kind of connector that you have selected. See [Table 7-3](#) for a key to the different fields.

Starting the Debugger Outside of the Project's Main Class

If you have multiple executable classes in your project, there might be times when you want to start the debugger from a class different from the one that is specified as the project's main class.

To start the debugger on a class other than the project's main class, right-click the file's node in the Projects window or Files window and choose Debug File.

You can start the debugger on a file only if it has a `main` method.

Stepping through Code

Once execution of your program is paused, you have several ways of resuming execution of the code. You can step through the code line by line (Step In) or in greater increments. See [Table 7-4](#) for the commands available for stepping or continuing execution and the following subtopics for a task-based look at the commands.

Table 7-4. Debugger Step Commands

Step	Command Description
Step Into (F7)	Executes the current line. If the line is a call to a method or constructor, and there is source available for the called code, the program counter moves to the declaration of the method or constructor. Otherwise, the program counter moves to the next line in the file.
Step Over (F8)	Executes the current line and moves the program counter to the next line in the file. If the executed line is a call to a method or constructor, the code in the method or constructor is also executed.
Step Out Of (Alt-Shift-F7)	Executes the rest of the code in the current method or constructor and moves the program counter to the line after the caller of the method or constructor. This command is useful if you have stepped into a method that you do not need to analyze.
Run to Cursor (F4)	Executes all of the lines in the program between the current line and the insertion point in the Source Editor.
Pause	Stops all threads in the current session.
Continue (Ctrl-F5)	Resumes execution of the program until the next breakpoint.

Executing Code Line by Line

You can have the debugger step a line at a time by choosing Run | Step Into (F7). If you use the Step Into command on a method call, the debugger enters the method and pauses at the first line, unless the method is part of a library that you have not specified for use in the debugger. See [Stepping into the JDK and Other Libraries](#) later in this chapter for details on making sources available for use in the debugger.

Executing a Method without Stepping into It

You can execute a method without having the debugger pause within the method by choosing Run | Step Over (F8). After you use the Step Over command, the debugger pauses again at the line after the method call.

Resuming Execution through the End of a Method

If you have stepped into a method that you do not need to continue analyzing, you can have the debugger complete execution of the method and then pause again at the line after the method call.

To complete execution of a method in this way, choose Run | Step Out Of (Alt-Shift-F7).

Continuing to the Next Breakpoint

If you do not need to observe every line of code while you are debugging, you can continue execution until the next breakpoint or until execution is otherwise suspended.

To continue execution of a program that has been suspended at a breakpoint, choose Run | Continue or press Ctrl-F5.

Continuing to the Cursor Position

When execution is suspended, you can continue to a specific line without setting a breakpoint by placing the cursor in that line and choosing Run | Run to Cursor (F4).

Stepping into the JDK and Other Libraries

When you are debugging, you can step into the code for the JDK and any other libraries if you have the source code that is associated with them registered in the IDE's Library Manager. See Making External Sources and Javadoc Available in the IDE in [Chapter 3](#) for information on associating source code with a library.

By default, the IDE does not step into JDK sources when you are debugging. If you use the Step In command on a JDK method call, the IDE executes the method and returns the program counter to the line after the method call (as though you used the Step Over command).

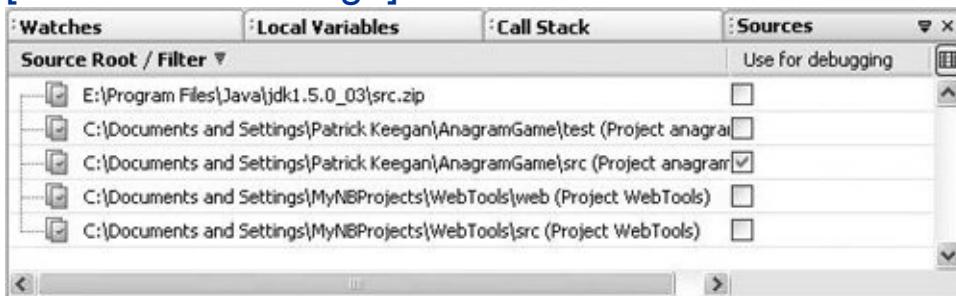
To enable stepping into JDK sources for a debugged application:

1. Start the debugger for the application.

- 2.** Open the Sources window (shown in [Figure 7-4](#)) by choosing Window | Debugging | Sources or by pressing Alt-Shift-8.

Figure 7-4. Sources window

[[View full size image](#)]



- 3.** Select the Use for Debugging checkbox for the JDK.

Limiting the Classes That You Can Step into for a Library

If you are using a library for debugging, you can set a filter to exclude some of the sources from being used.

To exclude classes from being used in the debugger:

- 1.** Start the debugger for the application.
- 2.** Open the Sources window by choosing Window | Debugging | Sources or by pressing Alt-Shift-8.
- 3.** Right-click the line for the library that you want to create an exclusion filter for and choose Add Class Exclusion Filter.

4. Type a filter in the Add Class Exclusion Filter dialog box.

The filter can be

- A fully qualified class name.
- A package name or class name with an asterisk (*) at the end to create a wildcard. For example, you could type the following to exclude all classes in the `javax.swing` package:
`javax.swing.*`
- An expression with a wildcard at the beginning. For example, to exclude all classes that have Test at the end of their names, you could use: `*Test`

You can create multiple class exclusion filters.

To disable the filter, deselect the Use in Debugging checkbox next to the filter in the Sources window.

To delete a class exclusion filter, right-click the filter and choose Delete.

Setting Breakpoints

A *breakpoint* is a marker that you can set to specify where execution should pause when you are running your application in the IDE's debugger. Breakpoints are stored in the IDE (not in your application's code) and persist between debugging sessions and IDE sessions.

When execution pauses on a breakpoint, the line where execution has paused is highlighted in green in the Source Editor, and a message is printed in the Debugger Console with information on the breakpoint that has been reached.

In their simplest form, breakpoints provide a way for you to pause the running program at a specific point so that you can

- Monitor the values of variables at that point in the program's execution.
- Take control of program execution by stepping through code line by line or method by method.

However, you can also use breakpoints as a diagnostic tool to do things such as:

- Detect when the value of a field or local variable is changed (which, for example, could help you determine what part of code assigned an inappropriate value to a field).
- Detect when an object is created (which might, for example, be useful when trying to track down a memory leak).

You can set multiple breakpoints, and you can set different

types of breakpoints. The simplest kind of breakpoint is a line breakpoint, where execution of the program stops at a specific line. You can also set breakpoints on other situations, such as the calling of a method, the throwing of an exception, or the changing of a variable's value. In addition, you can set conditions in some types of breakpoints so that they suspend execution of the program only under specific circumstances. See [Table 7-5](#) for a summary of the types of breakpoints.

Table 7-5. Breakpoint Categories

Breakpoint Type	Description
Line	Set on a line of code. When the debugger reaches that line, it stops before executing the line. The breakpoint is marked by pink background highlighting and the  icon. You can also specify conditions for line breakpoints.
Class	Execution is suspended when the class is referenced from another class and before any lines of the class with the breakpoint are executed.
Exception	Execution is suspended when an exception occurs. You can specify whether execution stops on caught exceptions, uncaught exceptions, or both.
Method	Execution is suspended when the method is called.
Variable	Execution is suspended when the variable is accessed. You can also configure the breakpoint to have execution suspended only when the variable is modified.
Thread	Execution is suspended whenever a thread is started or terminated. You can also set the breakpoint on the thread's death (or both the start and death of the thread).

Setting a Line Breakpoint

To set a line breakpoint, click the left margin of the line where you want to set the breakpoint or click in the line and press Ctrl-F8.

To delete the breakpoint, click the left margin of the line or click in the line and press Ctrl-F8.

If you want to customize a line breakpoint, you can do so through the Breakpoints window. Choose Window | Debugging | Breakpoints or press Alt-Shift-5. In the Breakpoints window, right-click the breakpoint and choose Customize.

Setting a Breakpoint on a Class Call

You can set a breakpoint on a class so that the debugger pauses when code from the class is about to be accessed and/or when the class is unloaded from memory.

To set a breakpoint on a class:

1. Choose Run | New Breakpoint (Ctrl-Shift-F8).
2. In the New Breakpoint dialog box, select Class from the Breakpoint Type combo box.
3. Enter the class and package names. These fields should be filled in automatically with the class currently displayed in the Source Editor.



You can specify multiple classes for the breakpoint to

apply to, either by using wildcards in the Package Name and Class Name fields or by selecting the Exclusion Filter checkbox.

Use asterisks to create wildcards in the Package Name and Class Name fields if you want the breakpoint to apply to multiple classes or all classes in a package. For example, if you enter just an asterisk (*) in the Class Name field, the breakpoint will apply to all classes in the package specified in the Package Name field.

Use the Exclusion Filter checkbox if you want the breakpoint to apply to all classes (including JDK classes) except for the ones that match the classes or packages specified in the Package Name and Class Name fields. You can set multiple breakpoints with the Exclusion Filter on. For example, you might set an exclusion filter on `com.mydomain.mypackage.mylib.*` because you want the class breakpoint to apply to all of your classes except those in the package `mylib`. However, if you do not want the debugger to pause at the loading of each JDK class that is called, you could also set a class breakpoint with an exclusion filter on `java.*`.

Setting a Breakpoint on a Method or Constructor Call

You can set a breakpoint so that the debugger pauses when a method or constructor is called before any lines of the method or constructor are executed.

To set a breakpoint on a method or constructor:

1. Choose Run | New Breakpoint.
2. In the New Breakpoint dialog box, select Method from the Breakpoint Type combo box.

3. Enter the class, package, and method names. These fields are filled in automatically according to the class open in the Source Editor and the location of the insertion point.

You can make the breakpoint apply to all methods and constructors in the class by checking the All Methods for Given Classes checkbox.

Setting a Breakpoint on an Exception

You can set a breakpoint so that the debugger pauses when an exception is thrown in your program.

To set a breakpoint on an exception:

1. Choose Run | New Breakpoint (Ctrl-Shift-F8).
2. In the New Breakpoint dialog box, select Exception from the Breakpoint Type combo box.
3. In the Exception Class Name field, select the type of exception that you would like to set the breakpoint on.
4. In the Stop On combo box, select whether you want the breakpoint to apply to caught exceptions, uncaught exceptions, or both.

Setting a Breakpoint on a Field or Local Variable

You can set a breakpoint so that the debugger pauses when a field or variable is accessed (or only when the field or variable is modified).

To set a breakpoint on a field or variable:

1. Choose Run | New Breakpoint (Ctrl-Shift-F8).
2. In the New Breakpoint dialog box, select Variable from the Breakpoint Type combo box.
3. Fill in the Package Name, Class Name, and Field Name fields.
4. Select an option from the Stop On combo box.

If you select Variable Access, execution is suspended every time that field or variable is accessed in the code.

If you select Variable Modification, execution is suspended only if the field or variable is modified.



Most of the fields of the New Breakpoint dialog box are correctly filled in for you if you have the variable selected when you press Ctrl-Shift-F8. You might have to select (highlight) the whole variable name for this to work. Otherwise, information for the method that contains the variable might be filled in instead.

Setting a Breakpoint on the Start or Death of a Thread

You can monitor the creation or death of threads in your program by setting a breakpoint to have execution suspended every time a new thread is created or ended.

To set a breakpoint on threads:

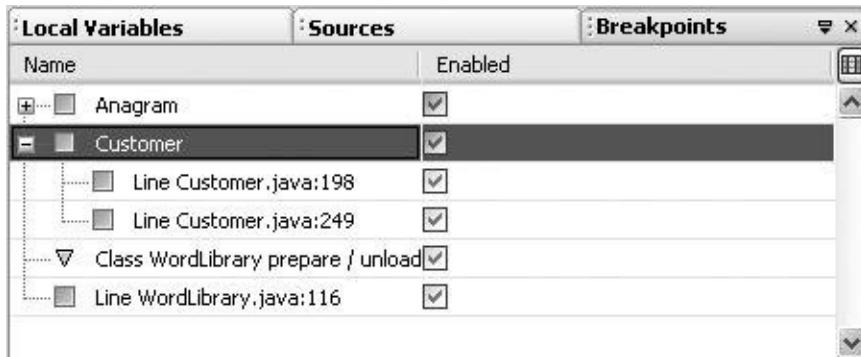
1. Choose Run | New Breakpoint (Ctrl-Shift-F8).

- 2.** In the New Breakpoint dialog box, select Thread from the Breakpoint Type combo box.
- 3.** In the Set Breakpoint On field, select Thread Start, Thread Death, or Thread Start or Death.

Managing Breakpoints

You can use the Breakpoints window, shown in [Figure 7-5](#), to manage breakpoints in one place. You can put breakpoints in groups, temporarily disable breakpoints, and provide customizations to the breakpoints from this window. To open the Breakpoints window, choose Window | Debugging | Breakpoints or press Alt-Shift-5.

Figure 7-5. Breakpoints window



Grouping Related Breakpoints

In some cases, you might have several related breakpoints that you would like to be able to enable, disable, or delete together. Or maybe you merely want to consolidate some breakpoints under one node to make the Breakpoints window less cluttered.

To group some breakpoints:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).

2. Shift-click or Ctrl-click to select the breakpoints that you want to group. Then right-click the selection and choose Set Group Name.

The breakpoints are grouped under an expandable node.

Enabling and Disabling Breakpoints

You might find it useful to keep breakpoints set throughout your application, but you might not want to have all of the breakpoints active at all times. If this is the case, you can disable a breakpoint or breakpoint group and preserve it for later use.

To disable a breakpoint or breakpoint group:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).
2. In the Breakpoints window, right-click the breakpoint or breakpoint group and choose Disable.

Deleting a Breakpoint

To delete a line breakpoint, click the left margin of the line that has the breakpoint or click in the line and press Ctrl-F8.

To delete another type of breakpoint:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).
2. In the Breakpoints window, right-click the breakpoint and choose Delete.

Customizing Breakpoint Behavior

There are a number of things that you can do to customize when a breakpoint is hit and what happens in the IDE when a breakpoint is hit. The following subtopics cover some of those things.

Logging Breakpoints without Suspending Execution

If you would like to monitor when a breakpoint is hit without suspending execution each time the breakpoint is hit, you can configure the breakpoint so that it does not cause suspension of execution. When such a breakpoint is hit in the code, a message is printed in the Debugger Console window.

To turn off suspension of execution when a breakpoint is hit:

- 1.** Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).
- 2.** In the Breakpoints window, double-click the breakpoint to open the Customize Breakpoint window. (For line breakpoints, right-click the breakpoint and choose Customize.)
- 3.** In the Action combo box, select No Thread (Continue).

Customizing Console Messages When Breakpoints Are Hit

You can customize the text that is printed to the console when a

breakpoint is hit in your code.

To customize the console message that is printed when a breakpoint is reached:

- 1.** Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).
- 2.** In the Breakpoints window, double-click the breakpoint to open the Customize Breakpoint window. (For line breakpoints, right-click the breakpoint and choose Customize.)
- 3.** In the Print Text combo box, modify the text that you want printed.

To make the printed text more meaningful, you can use some substitution codes to have things like the thread name and the line number printed. See [Table 7-6](#) for a list of the substitution codes available.

Table 7-6. Substitution Codes for Breakpoint Console Text

Substitution Code Prints

<code>{className}</code>	The name of the class where the breakpoint is hit. This code does not work for thread breakpoints.
<code>{lineNumber}</code>	The line number at which execution is suspended. This code does not work for thread breakpoints.
<code>{methodName}</code>	The method in which execution is suspended. This code does not work for thread breakpoints.
<code>{threadName}</code>	The thread in which the breakpoint is hit.
<code>{variableValue}</code>	The value of the variable (for breakpoints set on variables) or the value of the exception (for

exception breakpoints).

{variableType}	The variable type (for breakpoints set on variables) or the exception type (for exception breakpoints).
----------------	---

Making Breakpoints Conditional

You can set up a breakpoint to suspend execution of the code only if a certain condition is met. For example, if you have a long `for` loop, and you want to see what happens just before the loop finishes, you can make the breakpoint contingent on the iterator's reaching a certain value.

Here are some examples of conditions you can place on a breakpoint:

- `i==4` (which means that the execution will stop on the breakpoint only if the variable `i` equals 4 in the current scope)
- `ObjectVariable!=null` (which means that execution will not stop at the breakpoint until `ObjectVariable` is assigned a value)
- `MethodName` (where `Method` has a Boolean return type and execution will stop at the breakpoint only if `Method` returned `true`)

`CollectionX.contains(ObjectX)` (which means that execution will stop at the breakpoint only if `ObjectX` was in the collection

To make a breakpoint conditional:

1. Open the Breakpoints window by pressing Alt-Shift-5.
2. In the Breakpoints window, right-click the breakpoint that you want to place a condition on and choose Customize.
3. In the Customize Breakpoint dialog box, fill in the Condition field with the condition that needs to be satisfied for execution to be suspended at the breakpoint.

Conditional breakpoints are marked with the  icon.

Monitoring Variables and Expressions

As you step through a program, you can monitor the running values of fields and local variables in the following ways:

- By holding the cursor over an identifier in the Source Editor to display a tooltip containing the value of the identifier in the current debugging context.
- By monitoring the values of variables and fields displayed in the Local Variables window.
- By setting a watch for an identifier or other expression and monitoring its value in the Watches window.

The Local Variables window (shown in [Figure 7-6](#)) displays all variables that are currently in scope in the current execution context of the program, and provides and lists their types and values. If the value of the variable is an object reference, the value is given with the pound sign (#) and a number that serves as an identifier of the object's instance. You can jump to the source code for a variable by double-clicking the variable name.

Figure 7-6. Local Variables window

Watches		Breakpoints		Local Variables	
Name	Type			Value	
this	Anagrams\$3			#1375	
evt	ActionEvent			#1384	

You can also create a more custom view of variables and expressions by setting watches and viewing them in the Watches window.

The Watches window (Alt-Shift-2), shown in [Figure 7-7](#), is distinct from the Local Variables window in the following ways:

- The Watches window shows values for variables or expressions that you specify, which keeps the window uncluttered.
- The Watches window displays all watches that you have set, whether or not the variables are in context. If the variable exists separately in different contexts, the value given in the Watches window applies to the value in the current context (not necessarily the context in which the watch was set).
- Watches persist across debugging sessions.

Figure 7-7. Watches window

[\[View full size image\]](#)

Name	Type	Value
wordIdx	int	...
FeedbackLabel	JLabel	#1315
guessButton	JButton	#1370
FeedbackLabel	JLabel	#1315
idx		>"idx" is not a known variable in current context<

Setting a Watch on a Variable, Field, or Other Expression

To set a watch on an identifier such as a variable or field, right-click the variable or field in the Source Editor and choose New Watch. The identifier is then added to the Watches window.

To create a watch for another expression:

1. Choose Run | New Watch.
2. In the New Watch dialog box, type an expression that you want evaluated.

The expression must be written in Java syntax and can include local variables, fields of the current object, static fields, and method calls.

You can even create an instance of an object and call one of its methods. When doing so, be sure to refer to the class by its fully qualified name.

Monitoring the Object Assigned to a Variable

You can create a so-called *fixed watch* to monitor an object that is assigned to a variable (rather than the value of the variable itself).

To create a fixed watch:

1. After starting a debugging session, open the Local Variables window (AltShift-1).
2. Right-click the variable that you would like to set the fixed watch for and choose Create Fixed Watch.

A fixed watch is added to the Watches window with an ♦ icon. Because a fixed watch applies to a specific object instance created during the debugging session, the fixed watch is removed when the debugging session is finished.

Displaying the Value of a Class's `toString()` Method

You can add a column to the Local Variables and Watches windows to display the results of an object's `toString()` method. Doing so provides a way to get more useful information (such as the values of currently assigned fields) on an object than the numeric identifier of the object's instance that the Value column provides.

To display the `toString()` column in one of those windows:

1. Open the Local Variables window (Alt-Shift-1) or the Watches window (AltShift-2).
2. Click the ■ button in the upper-right corner of the window.
3. In the Change Visible Columns dialog box, select the `toString()` checkbox.



If you do not see the `toString()` column appear initially, try narrowing the width of the Value column to provide room for the `toString()` column to appear.

Changing Values of Variables or Expressions

As you debug a program, you can change the value of a variable or expression that is displayed in the Local Variables or Watches window. For example, you might increase the value of an iterator to get to the end of a loop faster.

To change the value of a variable:

1. Open the Watches window or the Local Variables window.
2. In the Value field of the variable or expression, type the new value and press Enter.

Displaying Variables from Previous Method Calls

The Call Stack window displays all of the calls within the current chain of method calls. If you would like to view the status of variables at another call in the chain, you can open the Call Stack window (Alt-Shift-3), right-click the method's node, and choose Make Current.

Making a different method current does not change the location of the program counter. If you continue execution with one of the step commands or the Continue command, the program will resume from where execution was suspended.

Backing up from a Method to Its Call

Under some circumstances, it might be useful for you to step back in your code. For example, if you hit a breakpoint and would like to see how the code leading up to that breakpoint works, you can remove (*pop*) the current call from the call stack to re-execute the method.

You can open the Call Stack window to view all of the method calls within the current chain of method calls in the current thread. The current call is marked with the  icon. Other calls in the stack are marked with the  icon.

To back up to a previous method call:

1. Open the Call Stack window (Alt-Shift-3).
2. Right-click the line in the Call Stack window that represents the place in the code that you want to return to and choose Pop to Here.

The program counter returns to the line where the call was made. You can then re-execute the method.



When you pop a call, the effects of the previously executed code are not undone, so reexecuting the code might cause the program to behave differently than it would during normal execution.

Monitoring and Controlling Execution of Threads

The IDE's Threads window (Alt-Shift-7), shown in [Figure 7-8](#), enables you to view the status of threads in the currently debugged program. It also enables you to change the thread that is being monitored in other debugger windows (such as Call Stack and Local Variables) and to suspend individual threads. See [Table 7-7](#) for a guide to the icons used in the Threads window.

Figure 7-8. Threads window

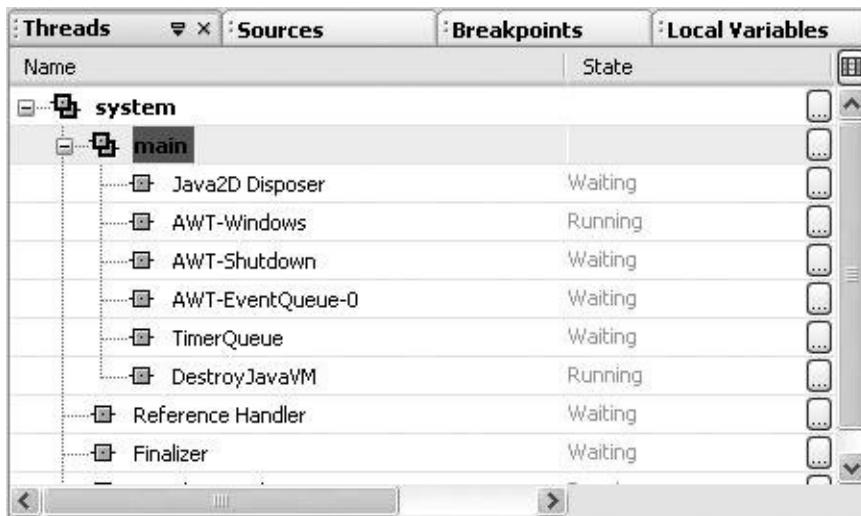


Table 7-7. Key to Icons in the Threads Window

Icon	Meaning
	Currently monitored thread
	Currently monitored thread group



Running thread



Suspended thread



Thread group

Changing the current thread does not affect the way the program executes.

Switching the Currently Monitored Thread

The contents of the Call Stack and Local Variables windows are dependent on the thread being currently monitored in the debugger (otherwise known as the *current thread*). To switch the currently monitored thread:

1. Open the Threads window by pressing Alt-Shift-7.
2. Right-click the thread that you want to monitor and choose Make Current.

Suspending a Single Thread

By default, when your program hits a breakpoint, all threads are suspended. However, you can also configure a breakpoint so that only its thread is suspended when the breakpoint is hit:

1. Open the Breakpoints window by pressing Alt-Shift-5.
2. In the Breakpoints window, right-click the breakpoint and choose Customize.

3. In the Customize Breakpoint dialog box, select Current from the Suspend combo box.

Isolating Debugging to a Single Thread

By default, all threads in the application are executed in the debugger. If you would like to isolate the debugging so that only one thread is run in the debugger:

1. Make sure that the thread that you want debugged is designated as the current thread in the Threads window (Alt-Shift-7). The current thread is marked with the  icon.
2. Open the Sessions window by pressing Alt-Shift-6.
3. In the Sessions window, right-click the session's node and choose Scope | Debug Current Thread.

Fixing Code During a Debugging Session

Using the IDE's Apply Code Changes feature (called Fix in NetBeans IDE 4.0), it is possible to fine-tune code in the middle of a debugging session and continue debugging without starting a new debugging session. This can save you a lot of time that would otherwise be spent waiting for sources to be rebuilt and restarting your debugging session.

The Apply Code Changes feature is useful for situations such as when you need to:

- Fine-tune the appearance of a visual component that you have created.
- Change the logic within a method.

Applying code changes does not work if you do any of the following during the debugging session:

- Add or remove methods or fields.
- Change the access modifiers of a class, field, or method.
- Refactor the class hierarchy.
- Change code that has not yet been loaded into the virtual machine.

To use the Apply Code Changes command while debugging:

1. When execution is suspended during a debugging session, make whatever code changes are necessary in the Source

Editor.

- 2.** Choose Run | Apply Code Changes to recompile the file and make the recompiled class available to the debugger.
- 3.** Load the fixed code into the debugger.

If you are changing the current method, this is done automatically. The method is automatically "popped" from the call stack, meaning that the program counter returns to the line where the method is called. Then you can run the changed code by stepping back into the method (F7) or stepping over the method (F8).

For a UI element, such as a JDialog component, you can close the component (or the component's container) and then reopen it to load the fixed code in the debugger.

- 4.** Repeat steps 1 through 3 as necessary.

Viewing Multiple Debugger Windows Simultaneously

By default, the debugger's windows appear in a tabbed area in the lower-right corner of the IDE in which only one of the tabs is viewable at a time. If you want to view multiple debugger windows simultaneously, you can use drag-and-drop to split a tab into its own window or to move the tab into a different window (such as the window occupied by the Debugger Console). You can also drag the splitter between windows to change the size of each window.

To create a separate window area for a debugger tab, drag the window's tab to one side of the current window until a red outline for the location of the new window appears. Then release the mouse button. See [Figures 7-9, 7-10, and 7-11](#) for snapshots of the process.

Figure 7-9. Call Stack window before moving it by dragging its tab to the left and dropping it in the lower left corner of the screen

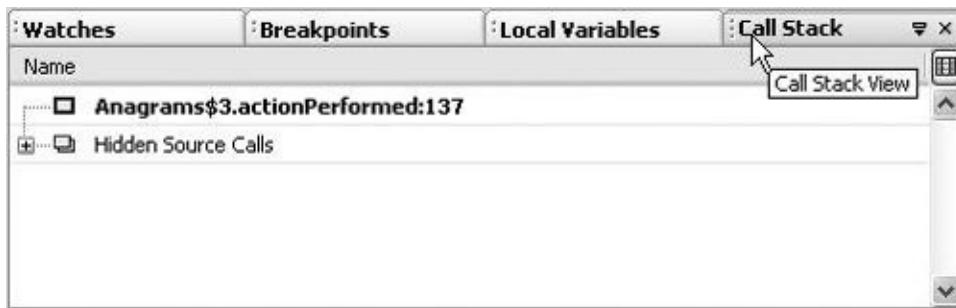


Figure 7-10. Call Stack window and the outlined

area to the left where it is to be dropped

[View full size image]

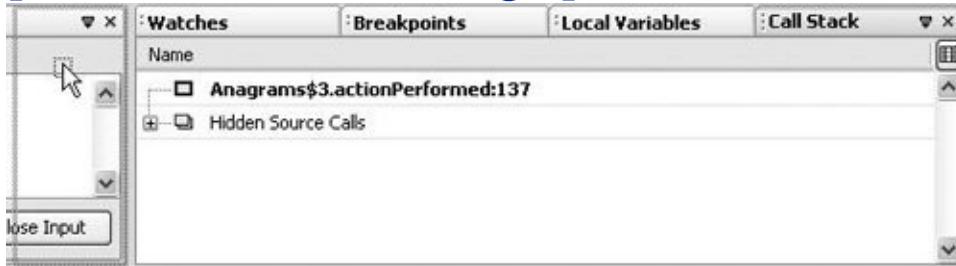
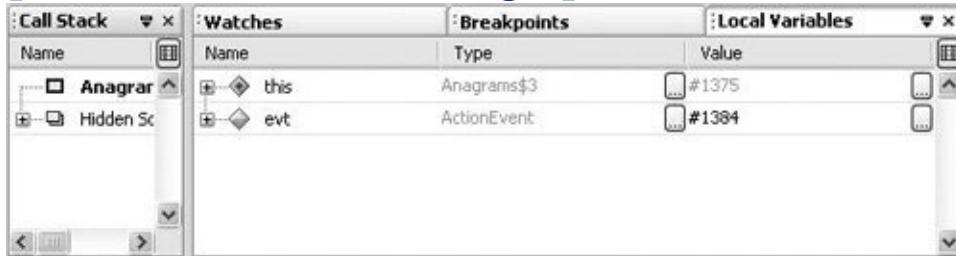


Figure 7-11. Call Stack after it has been moved to the new window area

[View full size image]



Chapter 8. Developing Web Applications

- [Representation of Web Applications in the IDE](#)
- [Web Application Structure](#)
- [Adding Files and Libraries to Your Web Application](#)
- [Editing and Refactoring Web Application Files](#)
- [Deploying a Web Application](#)
- [Testing and Debugging Your Web Application](#)
- [Creating and Deploying Applets](#)
- [Changing the IDE's Default Web Browser](#)
- [Monitoring HTTP Transactions](#)

NETBEANS IDE IS AN IDEAL ENVIRONMENT for developing web applications. The IDE eliminates many of the nuisances you normally would encounter, particularly in setting up the application and in the steps between coding, deploying, debugging, and redeploying your application. And because Ant is the basis for this automation, there are no proprietary mysteries you need to unravel if you want to make the project work without the IDE as an intermediary.

Following are some of the things the IDE does to make web application development easier:

- Provides a built-in Tomcat web server on which to deploy, test, and debug your applications.
- Sets up the file and folder structure of a web application for you.
- Generates and maintains the content of deployment descriptors, including the registering of any servlets that you add to your project.
- Generates and maintains an Ant script with targets (commands) for compiling, cleaning, testing, WAR file creation, and deployment to a server. This script saves you from having to move files manually to the web server.
- Ensures that the configuration files that appear in the WEB-INF folder of your application are not deleted when you run the Clean command to remove results of previous builds.
- Provides syntax highlighting; code completion; and other aids for editing servlet, JSP, HTML, and tag library files.
- Provides the Compile JSP command, which enables you to detect syntax errors in JSP files before deploying to your server, whether the errors occur at compile time or during the translation of the JSP file into a servlet.
- Provides comprehensive debugging support, which includes stepping into JSP files and tracking HTTP requests.

This chapter focuses on issues specific to web applications—creating and editing web components, debugging HTTP transactions, and so on—but does not include information on project creation or the IDE's support for the JavaServer Faces (JSF) and Struts web frameworks. See [Chapter 3](#) for

information on creating projects. See [Chapter 9](#) for information on using the IDE's support for web frameworks.

Most of the topics in this chapter assume that you are using the Tomcat web server, but it is also possible to use the Sun Java System Application Server, which supports full Java EE applications and includes full support for web services. The IDE also makes it possible to connect with other application servers, such as JBoss and BEA WebLogic. Most of the tasks detailed here that involve the Tomcat server are very similar to the equivalent tasks you would perform if deploying to an application server. See [Chapter 13](#) for more information on working with the Sun Java System Application Server and [Chapter 12](#) for information on developing, exposing, and consuming web services.

Representation of Web Applications in the IDE

Web applications are based on a somewhat intricate architecture where the development-time layout of files differs from that of a built application. The IDE helps you manage this process by:

- Providing a development-time-oriented view of your project in the Projects window. This view gives you easy access to your sources and information about your classpath but hides build results and project metadata. Perhaps more importantly, working in the Projects window ensures that none of the files you create will be inadvertently deleted when you run the Clean command on your project (as could happen, for example, if you worked directly in the **WEB-INF** folder of a built application).
- Providing a file-oriented view of your project in the Files window. This window is particularly useful for accessing and customizing your build script and for browsing your project outputs, such as the project's WAR file and its contents.
- Creating and maintaining an Ant script, which is used when you run typical commands such as Build Project and Run Project. Among other things, the Ant script automates the placement of your files in the built application, the packaging of those files into a WAR file, and deployment to the specified server.

Project View of Web Applications

The Projects window provides a "logical" representation of the application's source structure, with nodes for the following:

- Web Pages (for HTML, JSP, and image files that users of the application will have direct access to through their web browsers)
- Source Packages (for Java source packages, which in turn contain servlets and other Java classes)
- Test Packages (for unit tests)
- Configuration Files (for your deployment descriptor and other files)
- Web Services (where you can create and register web services; see [Chapter 12](#) for more on development of web services)
- Libraries (where you can add libraries or include the results of other IDE projects)
- Test Libraries (where you can add any libraries necessary for running unit tests on your application)

File View of Web Applications

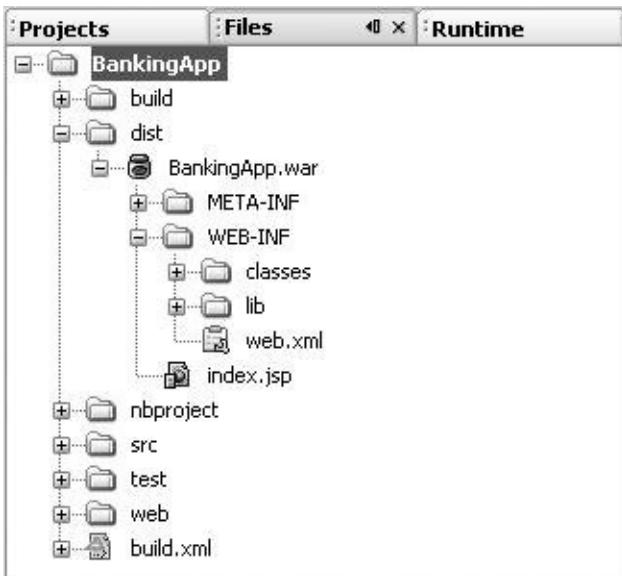
If you open the Files window, you will see the physical organization of the files on disk (as shown in [Figure 8-1](#)). The IDE also adds the `nbproject` (to hold project metadata) and `test` (for unit tests) folders, but these are not included in the final built application.

Figure 8-1. Files window with web application folder structure



When you build the application, either through the IDE or by directly running an Ant target, a `build` folder is created to hold the compiled classes, and the `dist` folder is created to hold the generated WAR file, as shown in [Figure 8-2](#).

Figure 8-2. Files window showing the structure of the built WAR file



Web Application Structure

Apache Jakarta provides guidelines on how to structure your web applications to ensure that they work properly with the Tomcat server. When you create a project in the IDE and select the Jakarta source structure, this structure is respected. (Similarly, you can set up a project to use the Java BluePrints structure, which is preferred if you will be deploying to the Sun Java System Application Server.)

The following is a quick rundown on the important structural elements of the built application according to the Jakarta guidelines:

- The root folder (known as the document base), which contains all of the other files in folders in the application.
- Files that are directly available to the users of the application through their web browsers, such as HTML files, images, and JSP files.
- The WEB-INF folder, which contains the deployment descriptor file (`web.xml`) and the `classes`, `lib`, `tags`, and other folders and files. The contents of WEB-INF comprise the bulk of the application and are not directly available to users.
- The `classes` folder contains compiled class and servlet files with their package hierarchy reflected by subfolders.

You can find additional information on Tomcat source structure at <http://jakarta.apache.org/tomcat/tomcat-5.0-doc/appdev/source.html>.

You can also use the Java BluePrints structure for web applications. This structure is designed primarily with enterprise applications in mind, so it is useful if you plan to later extend your web application to include Enterprise JavaBeans components. For web applications, the main practical difference between the Java BluePrints and Apache Jakarta guidelines is in file layout. For example, under the Java BluePrints guidelines, the `src` folder contains a `java` folder, which holds the source packages and a `conf` folder, which contains the manifest.

You can find more information about Java BluePrints at <http://java.sun.com/blueprints/code/projectconventions.html>.

See [Table 8-1](#) for information on how the various source elements of a web application map to their representation in the IDE and where they end up in the deployed application.

Table 8-1. Matrix of Web Application Elements (Using the Apache Jakarta Structure) and Their Representation in the IDE

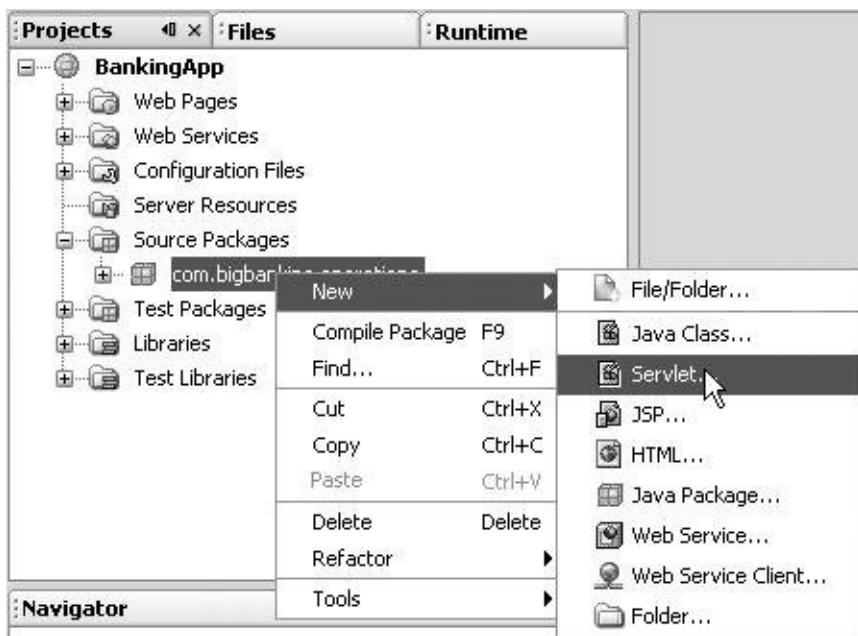
Content	Representation in the Projects Window	Representation in the Files Window	Location within the Built WAR File (Located in the <code>dist</code> Folder)
web pages	Web Pages node	<code>web</code> folder	root of the file
Java source files, Source Packages, servlets, and so on	Source Packages node	<code>src</code> folder	WEB-INF/classes folder
unit tests	Test Packages node	<code>test</code> folder	N/A
deployment descriptor (<code>web.xml</code>)	Configuration Files node	<code>web/WEB-INF</code> folder	WEB-INF folder
Tomcat context configuration file (<code>context.xml</code>)	Configuration Files node	<code>web/META-INF</code> folder	META-INF folder
libraries	Libraries node	<code>web/WEB-INF/lib</code> folder	WEB-INF/lib folder
Test classpath entries	Test Libraries node	<code>test</code> folder	N/A
project metadata including build script	Project Properties dialog box, which you can open by right-clicking the project's node and choosing Properties.	<code>build.xml</code> file, <code>nbproject</code> folder	N/A

Adding Files and Libraries to Your Web Application

Once you have created a web project through the New Project wizard, you can start populating it with web pages and code.

The most straightforward way to create files is by opening the Projects window, right-clicking the project's node or the specific folder where you want to place the file, and choosing New and then a template from the submenu (see [Figure 8-3](#)). A short wizard appears for the template, enabling you to set the name and other characteristics of the file. In general, the wizard guides you to help make sure that the files are placed in an appropriate directory to fit the structure of a well-designed web application.

Figure 8-3. Creating a new servlet from a package's contextual menu





The New submenu of a node's contextual menu directly displays a selection of commonly used templates. If you want to see the full selection, choose | New File/Folder.

The templates are grouped into several categories, such as Java Classes, Web, and Database. In addition to servlet and JSP file templates, the Web category contains templates for filters, web application listeners, tag files, and tag handlers, which contain useful sample code for those specific types of files.

Conceptually, files in a web application break down into a few different types of files, the function of which determines where you place the file in the application. The main types of files that you need to add to the project are

- Web pages and other public files, meaning files that users of the application can access directly through their web browsers. Typically, these include JSP files, HTML files, and image files.
- Private files, meaning files that are not directly viewable to the end users and that typically do the bulk of the processing in the application. These include Java classes, JSP files, servlets, and tag libraries, and end up within the `web/WEB-INF/classes` folder of the compiled web application.
- External resources, meaning files created outside of the project that the files in the project depend on. These can include tag libraries, JAR files that are output from other IDE projects, and other JAR files, and are kept within the `WEB-INF/lib` folder of the compiled web application.

In addition, there are configuration files such as the deployment descriptor (`web.xml`) and files specific to the server you are deploying to, but the IDE generates and maintains these files for you. For example, if you create a servlet in the IDE, the servlet is registered in the `web.xml` file automatically.



Tag libraries can be added to your web application as libraries or as source. See [Adding Tags and Tag Libraries](#) later in this chapter for more information on working with tag libraries.

Creating Web Pages and Other Public Files

Generally, you add web pages and other public files directly within the Web Pages node of the Projects window or in a folder of your creation within that node. When the application is built, these files are placed by the project's Ant script in the application's `web` folder.

To add a publicly viewable HTML file, right-click the Web Pages node and choose New | HTML.

To add a publicly viewable JSP file, right-click the Web Pages node and choose New | JSP. The ensuing wizard enables you to specify whether the JSP file uses standard syntax (and has the `.jsp` extension) or XML syntax (has the `.jspx` extension). You can also have the file created as a fragment (using the `.jspf` extension), which you would later reference from other pages with `include` statements.

Creating Classes, Servlets, and Other Private

Files

As with general Java projects, classes are organized within packages under the Source Packages node. For most projects, this node corresponds with a folder on your disk called `src`. When the application is built, these files are placed by the Ant script in the application's `WEB-INF/classes` directory.

To add a class to your project, right-click the Source Packages node or the node of a specific package and choose New | Java Class. If you have not created a package for the class you are adding, you can do so in the wizard as you are creating the class.

To add a servlet to your project, right-click the Source Packages node or the node of a specific package and choose New | Servlet. The wizard for creating the servlet also guides you through registering the servlet in the application's deployment descriptor (`web.xml` file).

If you would like to add a file based on a more specific template, right-click a package node and choose New | File/Folder to get a broader list of templates, including some for JavaBeans components, `.properties` files, XML files, and specialized web components. Note that many of the web templates have useful skeleton code and suggestions within the comments to get you started developing these kinds of objects.

See [Table 8-2](#) for a list of templates that you can find in the Web and Web Services categories of the New File wizard.

Table 8-2. Web Templates

Template	Description
JSP	Enables you to create a JSP file (standard syntax), JSP document (XML syntax), or a JSP fragment (which would be statically referenced)

	from another file).
Servlet	Creates a Java class that extends the <code>HttpServlet</code> class. Also enables you to register the servlet in the project's deployment descriptor (<code>web.xml</code>) file.
Filter	Creates a Java class that implements the <code>javax.servlet.Filter</code> interface. Filters enable you to modify HTTP requests to a servlet and responses from the servlet. In the wizard, you can create either a basic filter or one that wraps the <code>ServletRequest</code> and <code>ServletResponse</code> objects. In the template's wizard, you can register the filter in the project's deployment descriptor (<code>web.xml</code>) file.
Web Application Listener	Creates a Java class that implements one or more of the listener interfaces available for servlets, such as <code>ServletContextListener</code> and <code>HttpSessionListener</code> . Depending on the interfaces you select in the wizard, the created class will listen for events, such as when servlet contexts are initialized or destroyed, the servlet session is created or destroyed, or attributes are added to or removed from a context or session. In the template's wizard, you can register the listener in the project's deployment descriptor (<code>web.xml</code>) file.
Tag Library Descriptor	Creates a descriptor for a custom tag library. You can then register tag files and tag handlers in this file manually or when you use the New File wizard to create new tag files and tag handlers.
Tag File	Creates an empty <code>.tag</code> file with comments suggesting JSP syntax elements that you can use to create a custom tag.
Tag Handler	Creates a Java class for custom JSP tags. The template includes code comments with sample code and suggestions for how you might go about creating the custom tags.
HTML	Creates an HTML file with the basic tags entered.
Web Service	Creates a simple web service. See Chapter 12

for more information on extending web applications with web services.

Message Handler	Creates a SOAP-based message handler for web services. See Adding Message Handlers to a Web Service in Chapter 12 .
Web Service Client	Creates a web service client based on JSR 109. See Chapter 12 for more information on consuming web services with web applications.
WSDL File	Creates a WSDL (Web Services Description Language) file with the basic tags entered.

Adding External Resources to Your Project

If your web application needs to be packaged with any libraries, you can add them through the Libraries node. If your project has been set up to work with the Tomcat server, the JSP and servlet libraries are included automatically and listed under a subnode for the Tomcat server.

You can add external resources to your project in one of the following three forms:

- An individual folder or JAR file.
- A cluster of resources (library). This cluster might include multiple JAR files, sources for the JAR files (which are necessary for code completion or if you want to step through the library's code with the debugger), and Javadoc documentation. You can create such a cluster in the Library Manager (Tools menu).

- The output of another IDE project.

To add an individual folder or JAR file to your project, right-click the Libraries node of your project, choose Add JAR/Folder, and navigate to the folder or JAR file in the file chooser.

To add a cluster of related resources to your project, right-click the Libraries node, choose Add Library, and select the library from the list in the Add Library dialog box. If the library you are looking for is not there, you can add it to the list by choosing Manage Libraries.



Besides providing a way to cluster resources, the Library Manager makes it easy to access commonly used resources. Even if you do not need to cluster resources, you still might want to add individual resources to the Library Manager to save yourself from having to dig through file choosers to add resources to other projects.

To add the results of another IDE project, right-click the Libraries node of the current project, choose Add Project, and navigate to the project's folder on your disk. In the file chooser, NetBeans IDE project folders are marked with the icon.



If you want to add a JAR file or the output of a project to the WAR file without making it part of the project's compilation classpath, you can do so through the Packaging node in the Project Properties dialog box. See [Customizing Contents of the WAR File](#) later in this chapter.

Adding Tags and Tag Libraries

Tags and tag libraries can be added to a web project in any of the following forms:

- Packaged in a JAR file containing a tag library descriptor file (TLD) and the associated tags, in the form of tag files (using JSP syntax) and/or tag handlers (written in Java). Such tag libraries appear in a web application's `WEB-INF/lib` folder or in the server's shared libraries folder. You can add a tag library to a web project through the web project's Libraries node (see [Adding External Resources to Your Project](#) earlier in this chapter).

If you are developing a tag library from scratch, you can create a Java Library project for that library and then add the library to a web project by rightclicking the web project's Libraries node and choosing Add Project. See [Chapter 3](#) for information on creating Java Library projects.

- As tag files (using either standard or document syntax) included in the web application's `WEB-INF/tags` folder. You can add new tag files to your project by right-clicking the project's node and choosing New | File/Folder and then selecting the Tag File template from the Web category in the wizard.
- As a TLD file located in the `WEB-INF/tlds` folder and tag handlers (written as Java files) within the Source Packages node. You can add new TLD files and tag handlers to your project by right-clicking the project's node and choosing New | File/Folder and then selecting the templates from the Web category in the wizard.

Editing and Refactoring Web Application Files

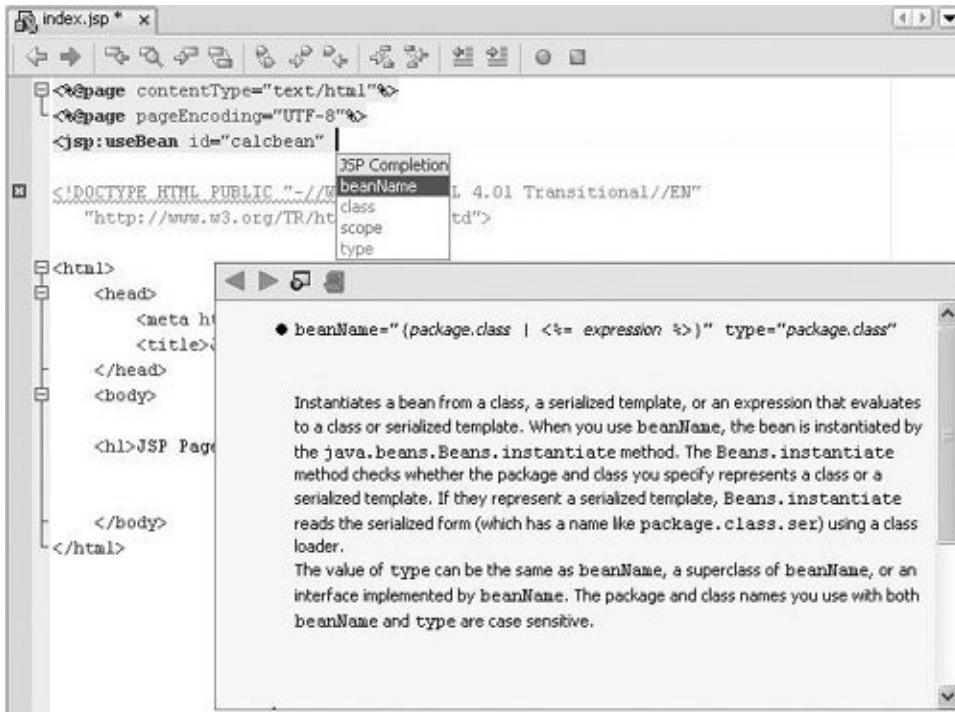
The IDE's Source Editor provides many features that make the typing and changing of code for various web application components easier. This section goes over a few of the features that are particularly useful for web applications. See [Chapter 5](#) for more information on IDE editing and refactoring features.

Completing Tags

The IDE enables you to have tags and tag attributes completed automatically in JSP, HTML, and XML files. The tag completion feature not only reduces the number of keystrokes that you type, but also provides popup documentation for the possible ways to complete the tag or attribute that you are typing, as shown in [Figure 8-4](#).

Figure 8-4. The code completion feature in a JSP file

[[View full size image](#)]



Open the tag completion popup by typing the beginning of a tag and pressing Ctrl-spacebar or waiting a second for code completion to kick in automatically. If there is only one way to complete the word you are typing, the end of the word is filled in automatically. If there are multiple ways to complete the tag, a popup list of those possibilities is displayed. You can keep typing to narrow the list or select the text you want, using the mouse or arrow keys.

For example, if you want to add the following statement in a JSP file

```
<jsp:useBean id="hello" scope="page"
class="org.mydomain.mypackage.MyClass" />
```

you can enter it in the following steps (don't worry the number of steps illustrated here may seem daunting, but it is only

because they are presented in such minute detail):

- 1.** Type `<jsp:u`
- 2.** Press Ctrl-spacebar (`seBean` is appended).
- 3.** Press the spacebar and type `i`
- 4.** Press Ctrl-spacebar (`d=""` is appended with the insertion point left between the quotation marks).
- 5.** Type `hello`
- 6.** Press the right-arrow key and then the spacebar.
- 7.** Type `s`
- 8.** Press Enter (`cope=""` is appended with the insertion point left between the quotation marks).
- 9.** Type `p`
- 10.** Press Ctrl-spacebar (`age` is appended).
- 11.** Press the right-arrow key and then the spacebar.
- 12.** Type `c`
- 13.** Press Enter (`lass=""` is appended with the insertion point left between the quotation marks).
- 14.** Type `o`
- 15.** Press Ctrl-spacebar (`rg` is appended, assuming that the class is part of your project).

- 16.** Type a period `(.)` and press Ctrl-spacebar (`mydomain` is filled in).
- 17.** Type a period `(.)` and press Ctrl-spacebar (`mypackage` is filled in).
- 18.** Type a period `(.)` and press Ctrl-spacebar (`MyClass` is filled in).
- 19.** Press the right-arrow key and then the spacebar and type `/>`

See Generating Code Snippets in [Chapter 5](#) for information on completing Java expressions and configuring code completion.

Using Code Templates for JSP Files

For commonly used JSP code snippets, you can take advantage of code templates in the Source Editor to reduce the number of keystrokes. You can access a JSP code template by typing the abbreviation for the code template and then pressing the spacebar.

See [Table 8-3](#) for a list of code templates for JSP files. Code templates are available for other types of files as well. See [Chapter 5](#), [Table 5-1](#), for a list of code templates for Java classes.

Table 8-3. JSP Abbreviations in the Source Editor

Abbreviation	Expands To
<code>ag</code>	<code>application.getAttribute(" ")</code>

```
ap application.putAttribute("|")

ar application.removeAttribute("|")

cfgi config.getInitParameter("|")

oup out.print("|")

oupl out.println("|")

pcg pageContext.getAttribute("|")

pcgn pageContext.getAttributeNamesInScope()

pcgs pageContext.getAttributesScope("|")

pcr pageContext.removeAttribute("|")

pcs pageContext.setAttribute("|,")

rg request.getParameter("|")

sg session.getAttribute("|")

sp session.putAttribute("|")

sr session.removeAttribute("|")

jspf <jsp:forward page="|"/>

jspg <jsp:getProperty name="|" property="" />

jspi <jsp:include page="|"/>

jspx <jsp:plugin type="|" code="" codebase="" /> </jsp:plugin>

jsps <jsp:setProperty name="|" property="" />

jspu <jsp:useBean id="|" type="" />

pg <%@page |%>
```

```
pga <%@page autoFlush="false"%>

pgb <%@page buffer="|kb"%>

pgc <%@page contentType="| "%>

pgerr <%@page errorPage="| "%>

pgex <%@page extends="| "%>

pgie <%@page isErrorPage="true"%>

pgim <%@page import="| "%>

pgin <%@page info="| "%>

pgit <%@page isThreadSafe="false"%>

pgl <%@page language="java"%>

pgs <%@page session="false"%>

tgb <%@taglib uri="| "%>
```

If an abbreviation for a code template is the same as text that you want to type (for example, you do not want it to be expanded into something else), press Shift-spacebar to keep the text from expanding into the code template.

You can modify the list of code templates in the Options window. See [Adding, Changing, and Removing Code Templates in Chapter 5](#).

Editing the Deployment Descriptor Manually

Although the IDE guides you through the adding of entries for

the deployment descriptor (`web.xml` file) as you add servlets, filters, and listeners to your project, you might have occasion to edit the file by hand.

To open the deployment descriptor in the Source Editor, open the Projects window, expand the Configuration Files node (or the Web Pages | Web-INF node), and double-click the `web.xml` file. The file opens as a mult tab document in the Source Editor with the General tab open (as shown in [Figure 8-5](#)). You can edit different parts of the `web.xml` file with the visual editors for different elements of the file (General, Servlets, Filters, and Pages), or you can click the XML tab to edit the file's XML source directly (see [Figure 8-6](#)).

Figure 8-5. Deployment descriptor visual editor

[[View full size image](#)]

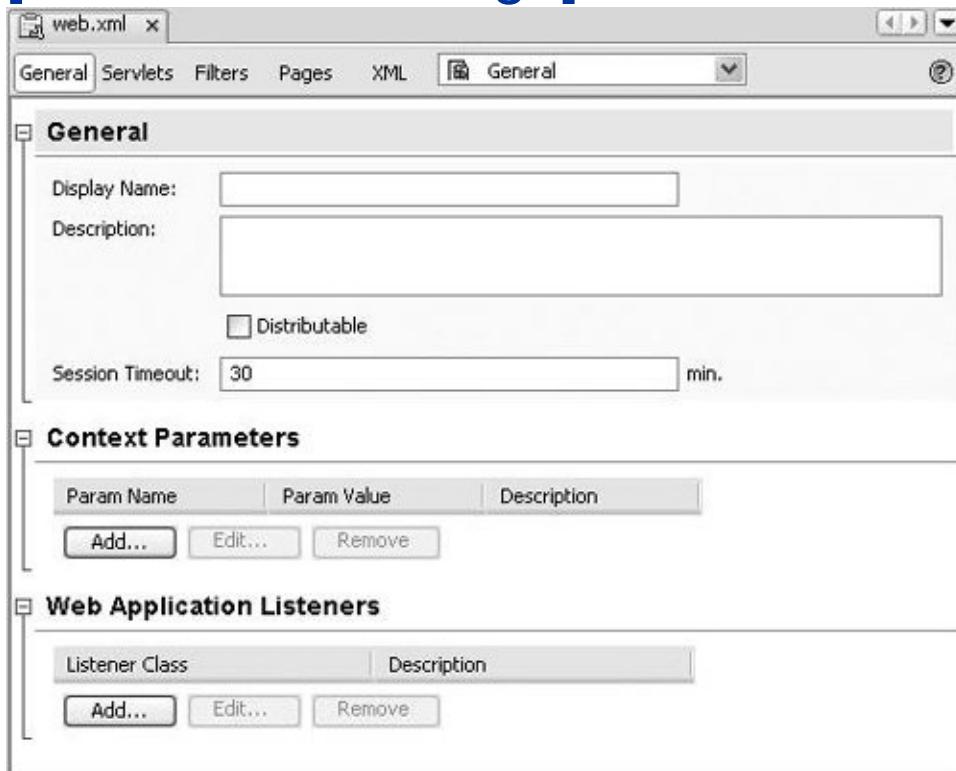
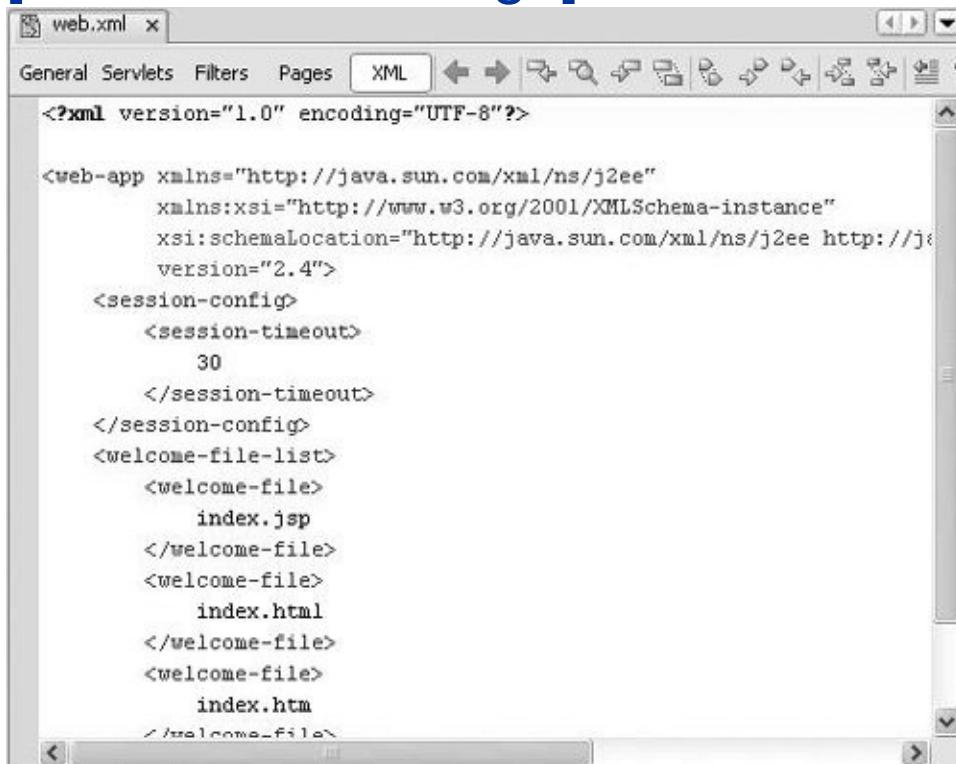


Figure 8-6. Deployment descriptor XML editor

[[View full size image](#)]



The screenshot shows the NetBeans IDE interface with the XML editor open. The title bar says "web.xml x". The tab bar has "General", "Servlets", "Filters", "Pages", and "XML" selected. Below the tabs is a toolbar with various icons. The main area contains the XML code for a web application's deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
          version="2.4">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>
            index.jsp
        </welcome-file>
        <welcome-file>
            index.html
        </welcome-file>
        <welcome-file>
            index.htm
        </welcome-file>
    </welcome-file-list>
</web-app>
```

Refactoring Web Components

NetBeans IDE's refactoring support extends to web applications and enterprise applications. For example, you can do the following:

- Rename classes, methods, and fields of servlets, tag handlers, and other web components. See [Renaming All](#)

Occurrences of the Currently Selected Class, Method, or Field Name in Chapter 5.

- Move classes to a different package or project (resulting in the class' being packaged in a different WAR file). See [Moving a Class to a Different Package in Chapter 5](#).
- Change method parameters, including parameter names, parameter types, and method visibility. You can also add method parameters. See [Changing a Method's Signature in Chapter 5](#).
- Changing visibility of fields and adding getter and setter accessor methods. See [Encapsulating a Field in Chapter 5](#).

When you rename a web component class (such as a servlet or tag handler), any corresponding name entries in the application's deployment descriptor (`web.xml` file) and/or tag library descriptor (TLD) are updated as well. When you move a class and the deployment descriptor is affected, you are prompted with a Confirm Changes dialog box to make sure that you want to process changes to the deployment descriptor.



Be careful not to rename or change the parameters of a servlet method that must be implemented with a given name according to the Servlet or Enterprise JavaBeans specifications.

Deploying a Web Application

By default, a web application is deployed to the server you have specified when you run that project.

For applications that you deploy to the Tomcat server, the application is deployed *in place*, meaning that the IDE creates an XML file that is placed in the server's `conf/Catalina/localhost/` directory and points Tomcat to the IDE project's `build` directory where the application files reside.

When you build a web project in the IDE, a WAR file is also created, which you can manually deploy to a server.

Customizing Contents of the WAR File

By default, a web project's generated WAR file includes

- All files displayed within the project's Web Pages node, including the `web.xml` and `context.xml` files
- Compiled class files of the source files within the Java Sources node, plus any other files placed there without the `.java` and `.form` file extensions
- Any libraries that you have added to the project's Libraries node

You can also add JAR files to the WAR file and filter out contents that would normally appear.

To customize a WAR file's contents:

1. Right-click the project's node in the Projects window and choose Properties.
2. Select the Build | Packaging node.
3. If you want to filter out contents from the generated WAR file, modify the regular expression in the Exclude from WAR File field.
4. If you want to add folders or files, do so through the Add JAR/Folder, Add Library, or Add Project button.

The Add JAR/Folder button enables you to add individual JAR files or folders, whether or not they come from IDE projects.

The Add Library button enables you to add any JAR files or clusters of JAR files that you have designated in the IDE's Library Manager.

The Add Project button enables you to add the JAR file that is output from another IDE project. When you add an IDE project to the WAR file, that project's JAR file is rebuilt every time you build the web application project.

Undeploying a Web Application

When you stop a web application that you have run through the IDE, the application remains deployed through a reference to that application in the form of an XML file in Tomcat's `conf/Catalina/localhost` directory.

To undeploy such an application from Tomcat:

1. Open the Runtime window.
2. Expand the Servers node; then expand the node for the

Tomcat server and the server's Web Applications node.

3. Right-click the node for the running web application and choose Undeploy.

Redeploying a Web Application

To remove your application from the server and then redeploy it, right-click the project's node and choose Redeploy Project.

Creating a WAR File

When you run the Build Project command on a web project in the IDE, a WAR file is created automatically and placed in the `dist` folder of the project. You can access this file and browse its contents in the Files window.

Deploying to a Different Tomcat Server

The IDE comes bundled with the Tomcat server, which facilitates web application development and testing. If you have a different Tomcat installation that you want to test on and/or deploy to, you can register that installation with the IDE. You can easily switch your application to work with the different server installations. This is particularly useful if you want to develop and test on one installation and then deploy to a production environment.

To set up the IDE to recognize a different Tomcat installation:

1. Choose Tools | Server Manager.
2. Click the Add Server button.

3. On the Choose Server page of the wizard that opens, select a server from the Server combo box, enter an IDE display name for that server in the Name field, and click Next.
4. On the Tomcat Server Instance Properties page, specify the Tomcat installation directory (and base directory, if it is a shared installation), and fill in an administrator username and password.

You can also determine whether to enable the IDE's HTTP Monitor. If the HTTP Monitor is enabled, you can monitor your application's server requests, cookies, and so on, which makes it easier to debug your application. However, this option slows the server, so you will probably want this option disabled if you are using this server as your production server. See [Monitoring HTTP Transactions](#) later in this chapter for information on using the HTTP Monitor.

5. Verify that the server is not using a port number used by another server. In the IDE, you can view the server's port number by mousing over the server's node and viewing the node's tooltip.

If another server instance is using the same port, you need to stop one of the servers and change the port it is using. Stop the server by right-clicking the server's node and choosing Stop. Then right-click the server node, choose Properties, and change the Server Port property. Restart the server by right-clicking the node, choosing Start.

6. If you have any existing web applications within the IDE that you want to run on the newly added server, modify the properties for each project to use the server. You can do so by right-clicking the project's node in the Projects window, choosing Properties, selecting the Run node, and choosing the server from the Server combo box.



If you later want to change the server's configuration, you can access the server's properties by opening the Runtime window, expanding the Servers node, right-clicking the specific server's node, and choosing Properties.

Testing and Debugging Your Web Application

NetBeans IDE provides a rich environment for troubleshooting and optimizing your web applications. Some of the features to ease testing of web applications include:

- The Compile JSP command, which enables you to check individual JSP files for errors before deploying to the server.
- Debugger integration with JSP files, which means that you can set break-points in JSP files and step through a JSP in the debugger (as opposed to having to step through the generated servlet code).
- Ability to step through tag files.
- Ability to evaluate Expression Language (EL) expressions in JSP files during a debugging session (by mousing over the expression or setting a watch).
- The HTTP Monitor, which keeps track of HTTP communication between servlets and the server. This feature is covered in detail in Monitoring HTTP Transactions later in this chapter.

See [Chapter 7](#) for more information on the IDE's general debugging features that these features extend.

Checking for JSP Errors

JSP files are not compiled like typical Java files before they are deployed. Instead, they are compiled by the server after they

have been deployed (where in fact they are first translated to servlets, which are then compiled). This makes it more cumbersome to correct errors that normally are detected when compiling, because it forces you to deploy the application, discover the error, undeploy, correct the error, and redeploy.

However, NetBeans IDE enables you to compile JSP files to check for errors before you package and deploy the application. You can either compile individual JSP files manually or specify that they be compiled when you build the project.

To compile a JSP file manually, select the file in the Projects window or in the Source Editor, and select Build | Compile File or press F9.

To have all JSP files compiled when you run the Build Project command, right-click the project's node, choose Properties, select the Compiling node, and select the Test Compile All JSP Files During Builds checkbox.

The compilation results are reported in the Output window, where you can discover any errors, whether they occur in the translation to the servlet or in the compilation of the servlet.

The compiled files themselves are placed in the project's `build/generated` folder, which you can view from the Files window. These files are not used when you are building and packaging the application for deployment.

Viewing a JSP File's Servlet

The generation of a servlet from a JSP file happens dynamically on the server on which the web application is deployed. You can view this generated servlet once you have run the project or the specific JSP associated with it by right-clicking the file and choosing View Servlet.



If you would like to see the servlet code that is generated when you run the Compile JSP command, open the Files window; open the `build/generated/src` folder; and navigate to the file, which is named according to the JSP filename but with a `_jsp` suffix and `.java` extension.

Viewing a File in a Web Browser

You can open components of a web application in a web browser from the IDE.

To view a specific JSP page in a web browser, you need to run that file individually by right-clicking the file in the Source Editor and choosing Run File (or pressing Shift-F6).

To open an HTML file in the web browser, right-click the HTML page's node in the Projects window and choose View.



The View command for HTML files is not available from the Source Editor. If you want to view the current HTML file in the Source Editor without your fingers leaving the key-board, press Ctrl-Shift-1 to jump to the file's node in the Projects window, press Shift-F10 to open the node's contextual menu, press the down-arrow key to select View, and press Enter.

Passing Request Parameters to a Web Application

You can manually test the way the web application will respond to certain input by running the application with certain request parameters specified ahead of time.

To pass request parameters to a JSP page:

1. Right-click the JSP file's node and choose Properties.
2. In the Request Parameters property, enter the parameters in URL query string format (where the expression begins with a URL; continues with a question mark [?] to mark the beginning of the query; and completes with the parameters as name/value pairs, where the pairs are separated by amper-sands [&]).

To pass request parameters to a servlet:

1. Right-click the servlet's node in the Projects window and choose Tools | Set Servlet Execution URI.
2. In the dialog box, append a question mark plus the name/value pairs, with each pair separated by an ampersand.

Debugging JSP and Tag Files

One of the IDE's features that has long made NetBeans IDE a favorite with web developers is the capability of the debugger to step into JSP files. You can set breakpoints in JSP files and step through the JSP line by line while monitoring the values of variables and other aspects of the running program. In addition, you can step into tag files.

To set a breakpoint in a JSP or tag file, select the line where you would like to pause execution and press Ctrl-F8. See [Chapter 7](#) for more information on debugging.

Creating and Deploying Applets

NetBeans IDE 5.0 does not have a specific project type for applets, so the development cycle for applets is a little different from that for other types of projects.

You cannot designate an applet as a main project, which means that several project-specific commands (such as Run Project) do not apply to applets.

However, you can still create, test, and deploy applets fairly easily. The general outline of applet development is as follows:

- 1.** Create a Java Library project to hold the applet.
- 2.** Create an applet from one of the templates in the New File wizard and fill in code for the applet. The different templates are described below in Creating an Applet.
- 3.** Test the applet in the JDK's applet viewer by right-clicking the applet's node in the Projects window and choosing Run File.
- 4.** Create a JAR file for the applet by right-clicking the applet's project node and choosing Build Project.
- 5.** If you want to add the applet to a web application, add the applet's project (or just the applet's JAR file) through the web project's Project Properties dialog box (Build | Packaging panel).

Creating an Applet

To create an applet:

1. Choose New Project, select the General category, select the Java Library template, and click Next.
2. Enter a name and location for the project, and click Finish to exit the wizard.
3. In the Projects window, expand the node for the project you have just created. Then right-click the Source Packages node and choose New | File/Folder.
4. In the New File wizard, select one of the available applet templates. There are four available:

Java Classes category, JApplet template. This template extends `javax.swing.JApplet` and is recommended over the Applet template, which is based on the less flexible `java.applet.Applet` class.

Java GUI Forms category, JApplet template. This template extends `javax.swing.JApplet` and enables you to use the IDE's Form Editor to design your applet visually. This template is recommended over the JApplet template in the AWT Forms subcategory.

Java Classes category, Applet template. This template extends `java.applet.Applet`.

Java GUI Forms | AWT Forms category, Applet template. This template extends `java.applet.Applet` and enables you to use the IDE's Form Editor to design your applet visually.

5. Click Next, specify a name and a package for the applet, and then click Finish.

You can then code the applet, either by hand or with the assistance of the Form Editor.

Running and Debugging an Applet in the Applet Viewer

As you are developing the applet, you can use the JDK's applet viewer to test the applet's functionality. When you use the Run File and Debug File commands, the applet is automatically displayed in the applet viewer.

To run an applet, right-click the applet's node in the Projects window and choose Run File.

To start debugging an applet:

1. Set a breakpoint in the code by selecting the line where you first want execution to pause and press Ctrl-F8.
2. Right-click the applet's node in the Projects window and choose Debug File.

Running an Applet in a Web Browser

If you want to see how your applet behaves in an actual web browser, you can open an HTML launcher for the applet.

To run an applet in a web browser:

1. Open the Files window and expand the project's `build` directory.
2. Right-click the HTML launcher file (it should have the same name as the applet class but with an HTML extension) and choose View.

The applet opens in the default web browser specified in the IDE. See [Changing the IDE's Default Web Browser](#) later in this chapter if you would like to change the IDE's default web

browser.



If you want to customize the HTML launcher file, you can copy the generated launcher file into the folder that contains the applet source file. This prevents the launcher file from being overwritten every time you run the applet.

When you run the applet or build the applet's project, the HTML file is copied into the folder with the compiled applet class. If you do not want this file to be included in the JAR that is created when you build the project, you can modify the filter for the JAR file's contents. In the Projects window, right-click the project's node and choose Properties. In the dialog box, select the Packaging node and modify the regular expression in the Exclude From JAR File field. For example, you could add a comma plus an expression like `**/Myapplet.html` to make sure that your launcher file (no matter which directory it is in) is excluded from the built JAR file.

Packaging an Applet into a JAR File

If you want to put an applet into a JAR file, you can do so by right-clicking the applet's project node in the Projects window and choosing Build Project.

The applet is compiled, and the compiled class files are placed in a JAR file in the `dist` folder, which you can view with the Files window.

Packaging an Applet into a WAR File

To add an applet to a web application:

1. Put the applet into a JAR file. See Packaging an Applet into a JAR File above for information on how to do this in the IDE.
2. Right-click the web application's project node in the Projects window and choose Properties.
3. In the Project Properties dialog box, select the Build | Packaging node.
4. Click the Add Project button, navigate to the applet's project folder, and click Add Project JAR Files.

Setting Applet Permissions

When you create and run an applet through the IDE, an `applet.policy` file is created with all permissions granted and is placed in the root folder of the project (which you can view through the Files window). You can modify this file by double-clicking its node to open it in the Source Editor.

You can also specify a different policy file for the applet. To specify a different policy file:

1. Right-click the web application's project node in the Projects window and choose Properties.
2. In the Project Properties dialog box, select the Run node.
3. In the VM Options field, modify the value of the `-Djava.security.policy` option to point to the policy file.

Changing the IDE's Default Web Browser

To change the IDE's default web browser:

- 1.** Choose Tools | Options and click General in the left side of the Options dialog box.
- 2.** Select the browser from the Web Browser drop-down list.

If the IDE cannot find a given web browser on your system, you might need to specify the executable for that web browser. To specify a web browser's executable:

- 1.** Choose Tools | Options and click Advanced Options.
- 2.** Expand the IDE Configuration | Server and External Tools Settings | Web Browsers node.
- 3.** Select the subnode for the browser and modify the Browser Executable property to point to that browser's executable.

To add to the IDE's list of web browsers:

- 1.** Choose Tools | Options and click Advanced Options.
- 2.** Expand the IDE Configuration | Server and External Tools Settings | Web Browsers node.
- 3.** Right-click the Web Browsers node, choose New | External Browser, type the browser's name, and click Finish.
- 4.** Select the subnode for the added browser and modify the Browser Executable property to point to that browser's executable.

Monitoring HTTP Transactions

NetBeans IDE provides a built-in HTTP Monitor to help isolate problems with data flow from JSP and servlet execution on a web server. There is no need to add logic to your web application to trace HTTP requests and associated state information. The NetBeans IDE built-in HTTP Monitor can do this for you.

When the IDE is configured with a web container, or a web application is deployed with a NetBeans HTTP Monitor servlet filter and filter mapping, the HTTP Monitor will automatically record all HTTP requests made to the web container. For each HTTP request that is processed by the web container, the HTTP Monitor not only records the request, but also records state information maintained in the web container.

By using the HTTP Monitor, you can analyze HTTP requests and store HTTP GET and HTTP POST requests for future analysis sessions. You can also edit these stored requests and replay them. This is a powerful feature to help isolate data flow and state information passed within an HTTP request to a web container. HTTP requests are stored until you exit the IDE. You can also save them so that they are available in subsequent IDE sessions.

Following are some of the things that you can do with the IDE's HTTP Monitor:

- Analyze HTTP request records
- Save HTTP request records
- Edit HTTP request records

- Refresh HTTP request records
- Sort HTTP request records
- Delete HTTP request records
- Replay HTTP request records

In the following sections, you will learn how to set up the HTTP Monitor, analyze the data the HTTP Monitor collects, and replay recorded HTTP requests.

Setting up the HTTP Monitor

When you run a web application and the HTTP Monitor is enabled, the HTTP Monitor should appear at the bottom of the IDE. The HTTP Monitor generally is automatically enabled for Tomcat, but not for Sun Java System Application Server or for other application servers.



For JBoss, WebLogic, WebSphere, and other applications servers, the HTTP Monitor can be used by following the instructions to set up the HTTP Monitor for servers started outside NetBeans IDE later in this chapter.

If the HTTP Monitor is not displayed at the bottom of the IDE when you run an application, you can enable it by performing the following tasks.

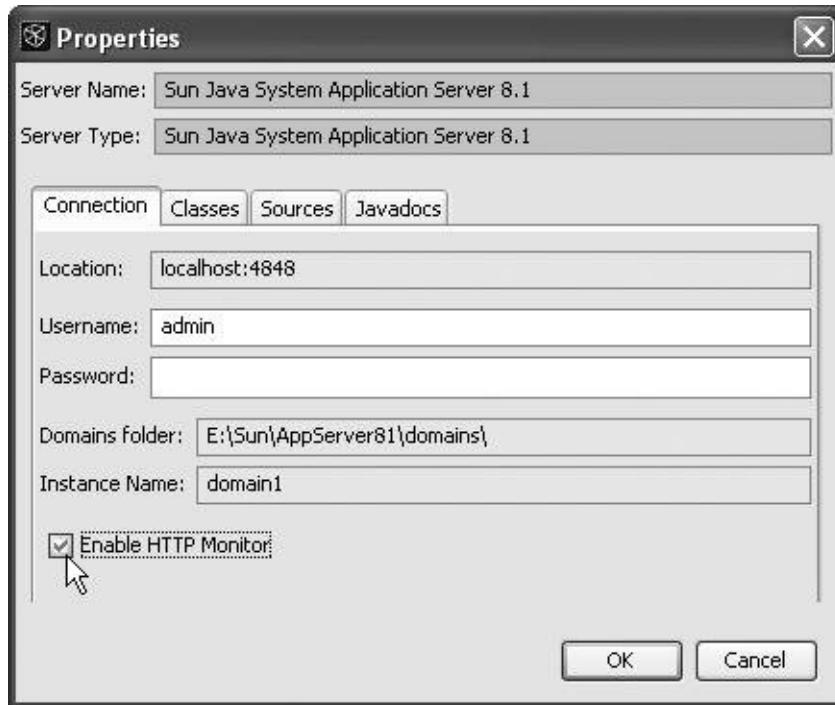
1. Expand the Runtime window's Servers node to show your

registered web server. Then right-click the server's node and choose Properties.

2. Select the Enable HTTP Monitor checkbox.
3. If the registered web server is currently running, stop and restart it by right-clicking your registered web server under the Runtime window's Servers node and selecting Start / Stop Server.

[Figure 8-7](#) shows the Enable HTTP Monitor property in the Sun Java System Application Server 8 Properties dialog box.

Figure 8-7. Properties dialog box for the Sun Java System Application Server



Setting up the HTTP Monitor for Servers Started Outside of the IDE

You can also use the HTTP Monitor on web servers started outside the IDE. To do so, execute the following tasks:

1. Go to the directory where the IDE is installed, and copy the `modules/org-netbeans-modules-schema2beans.jar` and `modules/org-netbeans-modules-web-httppmonitor.jar` files to your web module's `WEB-INF/lib` directory.
2. Add a filter declaration that is appropriate for your servlet's version to the top of your web module's `WEB-INF/web.xml` file.

Filters and filter mapping entries must be specified at the beginning of a deployment descriptor. See the examples below for filters for the servlets corresponding to the 2.3 and 2.4 versions of the Servlet specification.

A Servlet 2.4 filter declaration might look like the following:

```
<filter>
  <filter-name>HTTPMonitorFilter</filter-name>
  <filter-class>
    org.netbeans.modules.web.monitor.server.Monitor
  </filter-class>
  <init-param>
    <param-name>
      netbeans.monitor.ide
    </param-name>
    <param-value>
      name-of-host-running NetBeans IDE:http-server
    </param-value>
  </init-param>
</filter>
```

```
<filter-mapping>
<filter-name>
    HTTPMonitorFilter
</filter-name>
<url-pattern>
    /*
</url-pattern>
<dispatcher>
    REQUEST
</dispatcher>
<dispatcher>
    FORWARD
</dispatcher>
<dispatcher>
    INCLUDE
</dispatcher>
<dispatcher>
    ERROR
</dispatcher>
</filter-mapping>
```

A Servlet 2.3 filter declaration might look like the following:

```
<filter>
    <filter-name>HTTPMonitorFilter</filter-name>
    <filter-class>
        org.netbeans.modules.web.monitor.server.MonitorFilter
    </filter-class>
    <init-param>
        <param-name>
            netbeans.monitor.ide
        </param-name>
        <param-value>
            name-of-host-running NetBeans IDE:http-server
        </param-value>
```

```
</init-param>
</filter>
<filter-mapping>
  <filter-name>
    HTTPMonitorFilter
  </filter-name>
  <url-pattern>
    /*
  </url-pattern>
</filter-mapping>
```

A web application can be monitored with the IDE HTTP Monitor from multiple running NetBeans IDEs by adding more `init-param` entries to the servlet filter declaration in the web deployment descriptor. For instance, you would add an `init-param` entry such as the one shown below:

```
<init-param>
  <param-name>
    netbeans.monitor.ide
  </param-name>
  <param-value>
    name-of-2nd-host-running NetBeans IDE:http-serv
  </param-value>
</init-param>
```



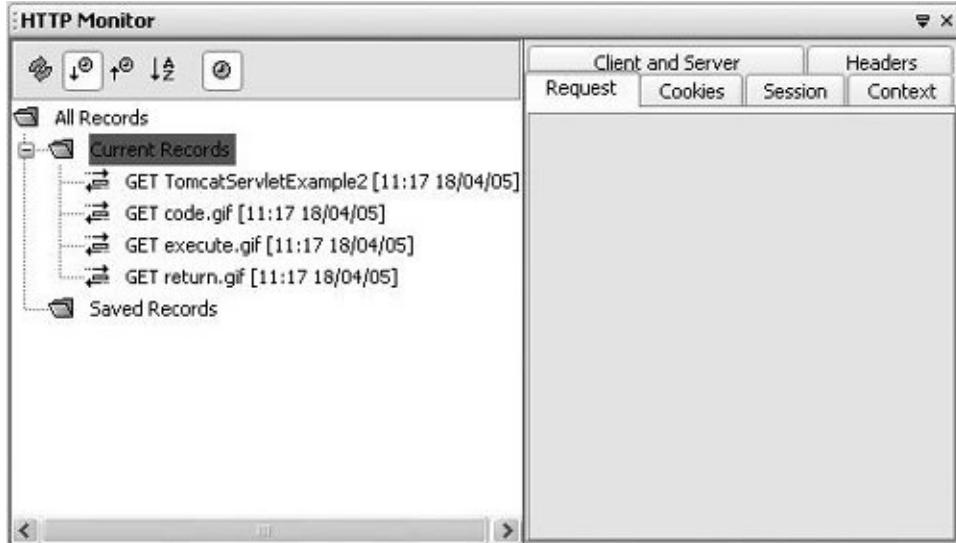
When you deploy the web module you have been monitoring with the HTTP Monitor to a production server, remember to remove the Servlet filter and filter mapping declarations from the web module's deployment descriptor. Otherwise, the web module will be open for HTTP monitoring from those NetBeans IDEs specified in the `init-param` section(s) of the servlet filter in the web module's deployment descriptor.

Analyzing the Collected Data

After you have set up the HTTP Monitor, you can use the HTTP Monitor to debug your web application by observing data flow from your JSP page and servlet execution on the web server. The HTTP Monitor records data about each incoming request. The HTTP Monitor is automatically displayed at the bottom of the IDE. A snapshot of the HTTP Monitor is shown in [Figure 8-8](#).

Figure 8-8. HTTP Monitor

[[View full size image](#)]



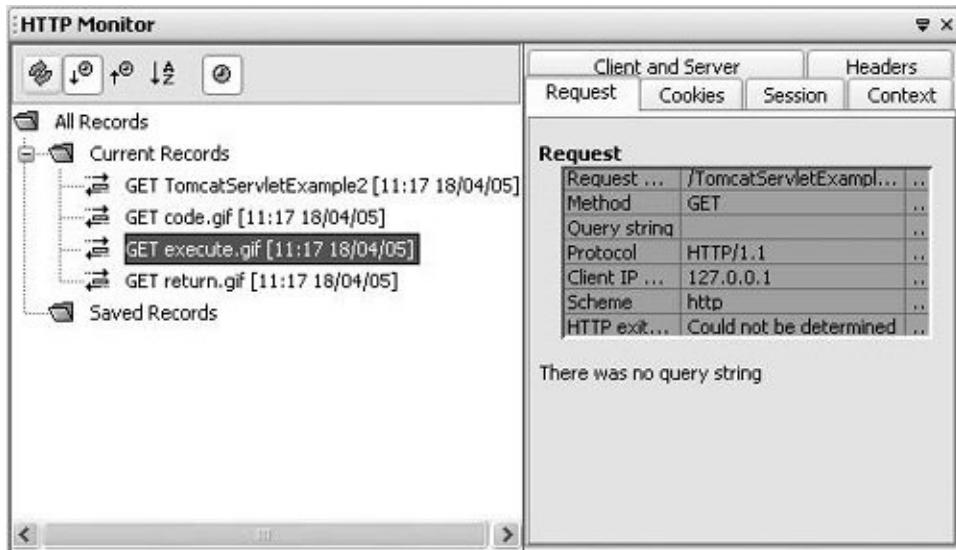
The HTTP Monitor consists of two panels. On the left is a tree view of HTTP request records. Every HTTP request made to the HTTP server is recorded in the HTTP Monitor. Requests resulting from internal dispatches are reflected by nested nodes under

those web containers that support it. In addition, forwarded or included requests are nested under the node corresponding to the main request.

Displayed in the right panel of the HTTP Monitor is additional data for a selected HTTP request record in the left panel. When you select an HTTP request record on the left, session data corresponding to the selected record is displayed in the right panel. The additional session information available in the right panel includes detailed request information; cookie name/value pairs; session data; servlet context attributes; initialization parameters; and client/ server information, such as client protocol, client IP address, server platform, and server hostname, along with additional HTTP request header information. The right panel allows you to view specific data in each of these categories by selecting a tab corresponding to the information you would like to see. [Figure 8-9](#) shows the additional session information for a selected HTTP request record.

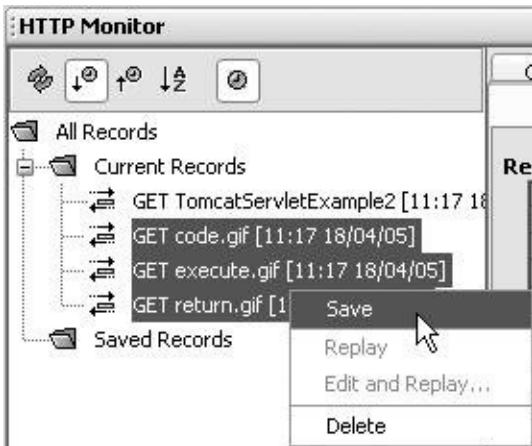
Figure 8-9. HTTP Monitor with a record selected and its request data displayed

[\[View full size image\]](#)



In the tree view (left panel of the HTTP Monitor) are two categories of records that you can view: Current Records and Saved Records. The Current Records category represents HTTP request records collected since the IDE has been started. Current records persist across restarts of the web server but not across restarts of the IDE. To persist current records across IDE restarts, you must save those records. Individual current records may be saved by selecting an HTTP request record, right-clicking the selected current record, and choosing Save from the contextual menu. You can select multiple HTTP request records by pressing the Shift key or Ctrl key. [Figure 8-10](#) illustrates the selecting and saving of multiple HTTP requests.

Figure 8-10. Saving multiple records in the HTTP Monitor



Notice that when you save HTTP requests, the selected records are moved to the Saved Records category. The selected records are not copied. Keep this in mind should you want to replay a sequence of HTTP requests.

The tree view (left panel of the HTTP Monitor) also provides several options for viewing the HTTP requests in the upper-left portion of the panel, in the form of five buttons. For example, you can reload all the HTTP request records, sort the HTTP request records by timestamp in descending or ascending order, sort the records alphabetically, and show or hide the timestamp for each HTTP request record. [Table 8-4](#) summarizes the action of each button.

Table 8-4. HTTP Monitor Toolbar Buttons

Request Record View Button	Action
reload	Reloads all the HTTP request records currently stored
descending sort	Sorts HTTP request records by timestamp in descending order
ascending sort	Sorts HTTP request records by timestamp in ascending order

alphabetically sort	Sorts the HTTP request records alphabetically
time stamp	Hides or displays the timestamps in the list of HTTP requests

In addition to being saved, HTTP records may be deleted. The IDE provides much flexibility for deleting HTTP records. For example, individual and multiple current records or all current records may be deleted. Individual saved records, multiple saved records, or all saved records can also be deleted.

To remove a record, right-click the record to be deleted and choose the Delete option from the contextual menu. To remove multiple records, select additional records, using the Shift or Ctrl key; then right-click the selected records and choose the Delete option from the contextual menu. To remove all current records, right-click the current records folder and choose the Delete option. To remove all Saved Records, right-click the Saved Records folder and choose the Delete option from the contextual menu.

Replaying HTTP Requests

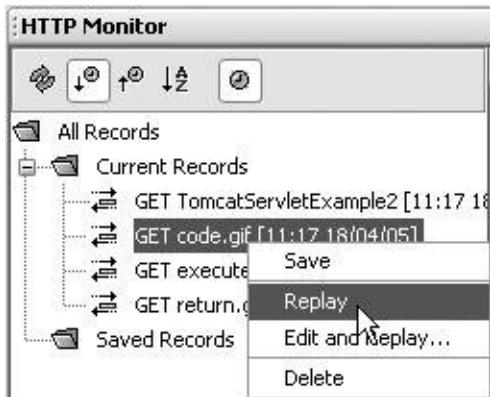
The most powerful feature of the HTTP Monitor is the editing and replaying of HTTP requests. By editing and replaying HTTP requests, you can quickly and easily trace and diagnose problems with the flow of data from JSP pages and servlet execution on the web server. When you replay an HTTP request, the response appears in your web browser. Thus you can track the result of a given HTTP request by having it replayed in your web browser.



The IDE opens your default web browser when you replay HTTP requests. If the IDE cannot find the operating system's default web browser, you can configure the web browser manually. See [Changing the IDE's Default Web Browser](#) earlier in this chapter.

Both current records and saved records may be replayed. To replay an HTTP request, select the HTTP request to replay in the left panel (tree view) and choose the Replay option from the contextual menu. [Figure 8-11](#) shows a Current Record being selected for replay.

Figure 8-11. Selecting a record to be replayed in the HTTP Monitor



Notice that the selected record is replayed in your browser after you have chosen Replay from the contextual menu.

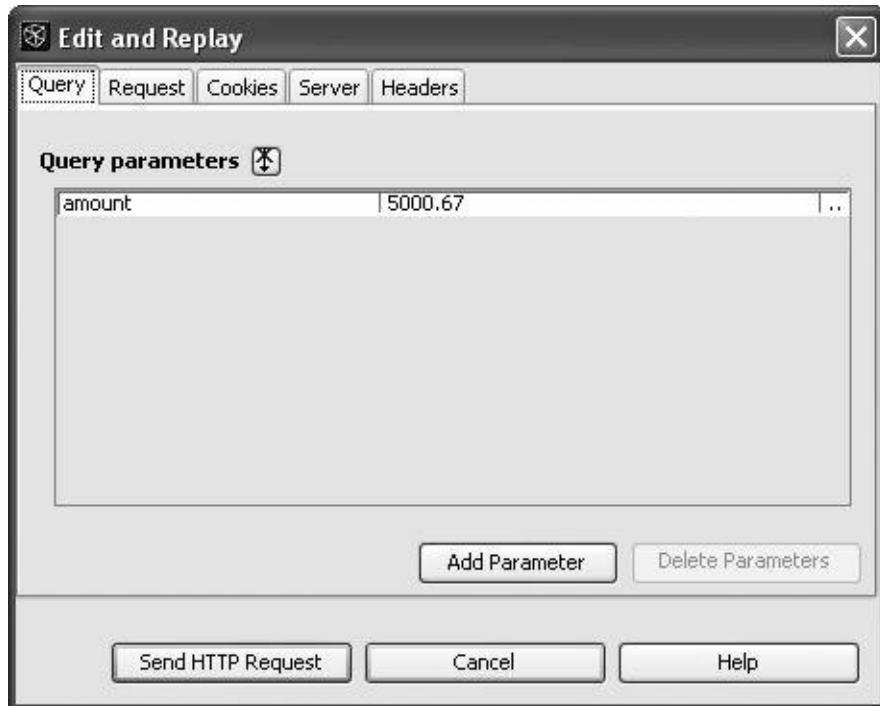
In addition to replaying HTTP requests, you can edit an HTTP request before replaying it. To do so, select an HTTP request in the left panel (tree view) either a Current Record or Saved

Record right-click the record, and choose the Edit and Replay option from the contextual menu. After you choose the Edit and Replay option, a dialog box is displayed, where you can make various modifications.

The supported modifications in the edit and replay of an HTTP request include options to edit a parameter to query, modify request URI parameters, modify cookies, modify server execution properties, and modify HTTP request parameters.

On the Query tab (see [Figure 8-12](#)), you can add a query parameter or delete a query parameter and modify URI request parameters. On the Request tab, you can modify the request URI by selecting the ellipsis (...) button next to the request parameter value. You can change the request method from a GET to a POST or PUT by selecting from the combo box to the right of the Request Method type in the left column. If you click the ellipsis button in the request protocol's far-right column, the Request Edit Property dialog box appears, which enables you to modify the request protocol. On the Cookies tab, you can add, modify or delete cookies associated with the HTTP request. On the Server tab, you can modify server execution values, such as the hostname and port where the HTTP request should be executed. You can add, modify, and delete HTTP headers in the Headers tab.

Figure 8-12. HTTP Monitor Edit and Replay dialog box



After making your desired edits to the HTTP request, you can replay the modified request by clicking the Send HTTP Request button. The resulting HTTP request and response is sent to your web browser, where the results are displayed.

Chapter 9. Creating Web Applications on the JSF and Struts Frameworks

- [JSF Overview](#)
- [Struts Overview](#)

WEB FRAMEWORKS WERE INTRODUCED IN NETBEANS IDE 5.0 to encapsulate best practices and thus reduce the overall development effort by providing additional declarative behavior for common idioms such as navigation. NetBeans IDE supports the ubiquity of Struts and JavaServer Faces (JSF) technologies by incorporating configuration file editing, assistance in setting up a web application project with the framework required libraries, and configuration files, and wizards for generating Java artifacts. This chapter focuses on the framework-specific features of web application projects; for general information on web application projects, see [Chapter 8](#).

JSF Overview

JavaServer Faces technology (JSF) was created under the auspices of JSR 127, which was finalized in 2004. JSF includes a set of APIs for representing UI components and their state, event handling, input validation, navigation, internationalization, and accessibility. A custom tag library provides the capability to define a JSF interface within a JavaServer Page (JSP). The component model and the standardization of the technology have led to increasing adoption.

A typical interaction begins with a form submission being routed through the JSF servlet to a method on a managed bean (JavaBeans component). The managed bean performs the appropriate action typically using a business delegate. After invoking the appropriate delegate method, control is returned to the JSF controller. The controller then forwards to the appropriate view using a set of navigation rules specified in the configuration file.

Following is an list of some important concepts in JSF technology:

- **Managed Bean.** A managed bean is a component associated with a UI component. The managed bean defines JavaBeans properties that are bound to a component value (typically a mapping between a form element and a value) or an instance. A managed bean can also be used to perform functions such as validation and event handling.
- **Configuration File.** The XML configuration file provides the capability to specify behavior, such as that of page navigation and managed beans, declaratively.

- **Tag Libraries.** The JavaServer Faces tag libraries provide access to the UI components and the capability to reference managed beans.
- **UI Components.** The JavaServer Faces component model provides the capability to create reusable components that are independent of their rendering. The UI components can be extended by component developers to create new components.
- **Navigation.** The page flow is specified in the navigation section of the configuration file. The page flow allows the next page to be specified in terms of a logical outcome. This allows the actual flow to change without affecting the application code.

This section is intended to serve as a brief overview to JSF technology; additional information can be found at (<http://java.sun.com/j2ee/javaserverfaces/index.jsp>).

Creating a Web Project with JSF Support

JSF support can be added during project creation when using the New Web Application wizard.

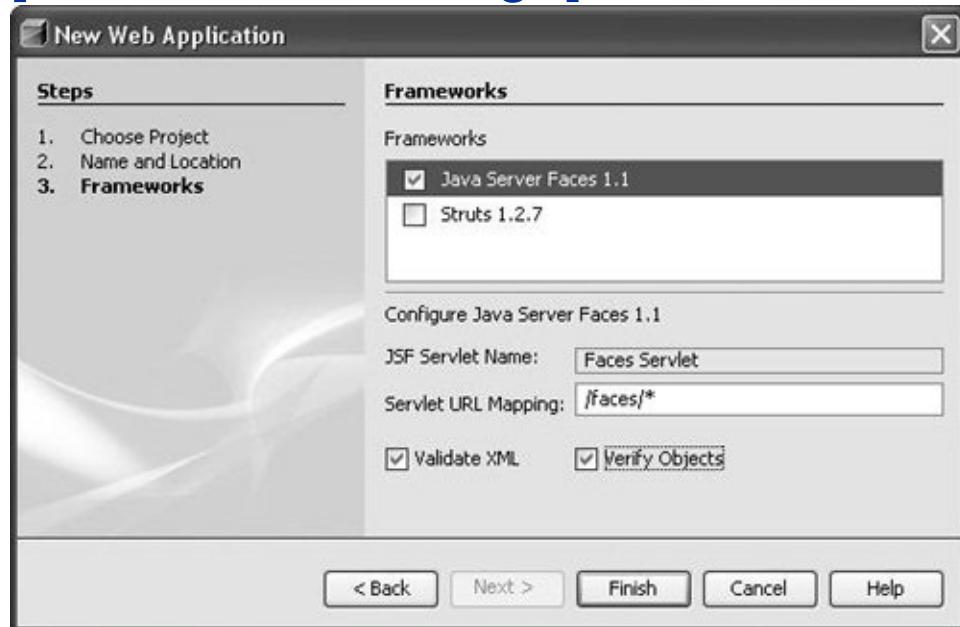
JSF support is enabled by selecting the JSF checkbox in the wizard (see [Figure 9-1](#)). This allows additional JSF-specific information to be specified:

- **JSF Servlet Name.** This text field contains the name used in the servlet entry that will be generated for the `FacesServlet` class. The servlet entry will also contain a reference to the generated configuration file.

- **Servlet URL Mapping.** This text field provides the capability to configure the set of URLs that are mapped to the Faces Servlet.
- **Validate XML.** If this value is checked, the configuration file will be validated when parsed by the `FacesServlet` class.
- **Verify Objects.** If this value is checked, all objects referenced from the configuration file will be checked to ensure successful creation. The above two properties provide early fail fast behavior and are useful to turn on at development time. Both of these properties are context parameters for the `FacesServlet` class, so they can easily be changed later.

Figure 9-1. New Web Application wizard with JSF framework support

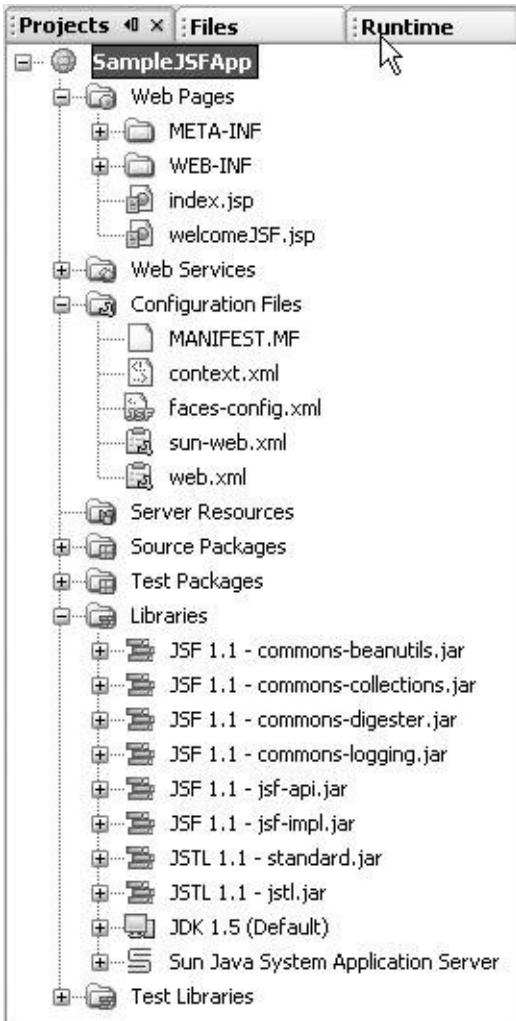
[[View full size image](#)]



After you complete the wizard, the standard web application project will be augmented to support JSF development as follows (see [Figure 9-2](#) for the JSF enabled web application project):

- `faces-config.xml` is generated in the configuration files area. The `facesconfig.xml` has the appropriate document type specified as well as the root element.
- The default `index.jsp` is augmented to refer to the `welcomeJSF.jsp`. The `href` tag is prefixed with the URL pattern specified in the wizard so the request will be intercepted by the `FacesServlet` class.
- The JSF libraries are added to the libraries directory. The libraries will be included by default when packaging the WAR file in `WEB-INF/lib` directory. If the libraries have already been deployed to the server or will instead be deployed as part of a Java EE application, the project's Properties dialog box can be used to set which libraries will be packaged with the WAR file.
- The `web.xml` file is updated with a servlet entry referencing the `FacesServlet` class. The context parameters such as the location for the `facesconfig.xml` file (and the appropriate context parameters as specified in the wizard for verifying objects and validating the configuration file) are also generated. A servlet mapping referencing the `FacesServlet` class will also be generated with the URL pattern specified in the wizard.

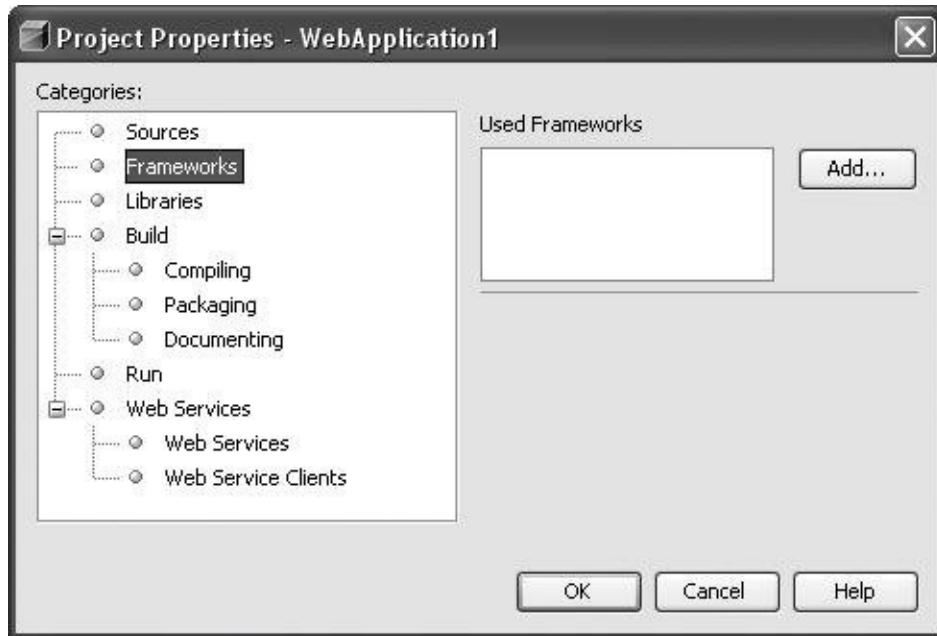
Figure 9-2. Web Application Project using the JSF framework



After you complete the wizard, code completion will be enabled for JSF tag libraries, JSF Java classes, and the `faces-config.xml` file.

Support for JSF can also be added to an existing web project using the Project Properties Framework tab (see [Figure 9-3](#)).

Figure 9-3. Project Properties dialog box, Framework tab



Managed Bean Support

A managed bean provides the capability to perform data binding and other functions such as validation and serving as a controller. The IDE provides a wizard to automate creation of managed bean classes.

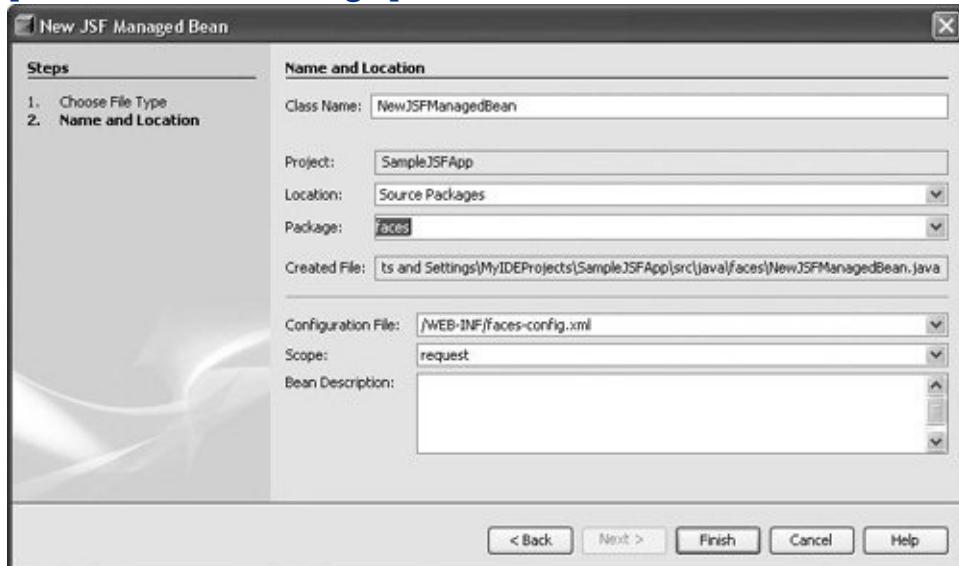
To create a managed bean class:

1. Open the New JSF Managed Bean wizard by right-clicking the node of a JSF-enabled web application project, choosing New | File/Folder, selecting the Web category, and selecting JSF Managed Bean Action.
2. In the Name and Location section of the wizard (shown in [Figure 9-4](#)), specify the name and package of the generated managed bean class. The JSF configuration file must also be specified; the default is typically correct. The scope of the managed bean must be specified. An optional description of

the managed bean can be specified in the description field.

Figure 9-4. New JSF Managed Bean wizard

[[View full size image](#)]



3. When the Finish button is clicked, the wizard will generate the managed bean classes along with a managed bean entry in the JSF configuration file.



The node for the managed bean class provides a wizard to help generate properties. A quick way to navigate to this node is to press Ctrl-Shift-1 from the Source Editor. The Bean Patterns node is a subnode that provides the capability to add various JavaBeans properties.

JSF Configuration File Support

In addition to code completion, the JSF configuration file provides several wizards launched from the context menu of the Source Editor. The wizards provided include:

- **Add Navigation Rule.** Encapsulates the possible transitions from the specified page using the JSF navigation-rule element.
- **Add Navigation Case.** Adds a single transition to an existing navigation rule or generates a navigation-rule if one does not exist. The wizard allows both action and outcome guards to be specified.
- **Add Managed Bean.** Automates generation of a managed bean entry. This wizard does not generate the managed bean Java class, so this is useful when referencing existing managed beans.



When you open the JSF configuration file in the Source Editor, you can navigate directly to externally referenced artifacts (such as Java and JSP files). Jumping to the source code is achieved by holding down the Ctrl key, moving the mouse over the identifier until it is underlined in blue, and then clicking it. For example, holding down the Ctrl key and moving the mouse over a managed bean class element displays the class name underlined in blue (See [Figure 9-5](#)), and clicking the identifier opens the Java file in the Source Editor.

Figure 9-5. Hyperlinking in faces-config.xml file

```
<managed-bean>
    <managed-bean-name>GuessBean</managed-bean-name>
    <managed-bean-class>faces.GuessBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>guess</property-name>
        <value>32</value>
    </managed-property>
</managed-bean>
```

Struts Overview

Struts was donated to the Apache Foundation in 2000 and provides a framework incorporating Java EE best practices. Struts is widely adopted due to its support for internationalization, powerful custom tag library (the first releases were prior to JSTL and contained similar tag capabilities), declarative configuration, support for a variety of presentation technologies, form validation, and its maturity.

Struts provides a front controller servlet and relies on other technology to provide the model (typically data access technologies such as Enterprise JavaBeans) and view (typically JSP) of the MVC design pattern. The Struts controller uses a configuration file (`struts-config.xml`) to map incoming requests to an action. An action receives form data as an `ActionForm` instance, and uses this data to execute the appropriate business logic. The configuration file allows logical names to be used to represent a view; thus, an action can forward control to the appropriate view using a logical name. Many web applications use JSP for the view, so Struts provides custom tag libraries that facilitate interaction with HTML forms.

The following are key elements of the Struts framework:

- **Action.** Struts incorporates the service to worker design pattern, where an action commonly invokes methods on a business delegate using the data supplied by an `ActionForm` bean and uses the Struts controller to forward control to the appropriate view. A business delegate provides a reusable web independent facade for interacting with business services such as relational databases, messaging systems, and web services.
- **ActionForm Bean.** An `ActionForm` bean represents data

shared between the view and the action. An ActionForm bean is available both for populating the view and providing input to an action. An ActionForm instance also has a `validate` method to allow input mapped from the view to be verified.

- **Configuration File (`struts-config.xml`).** The `struts-config.xml` file supports declaratively specifying the behavior of the Struts controller. Navigation (forwards) and behavior (actions) are specified in the Struts configuration file.
- **Tag Libraries.** The Struts tag libraries support internationalized applications and provide additional support for input forms interacting with the Struts framework. Struts incorporates the internationalization support in Java, where user-visible strings are specified in a language-specific resource bundle. Struts tag libraries such as form elements can specify a resource bundle key to allow the language-specific text to be displayed.

This section is intended to serve as an overview of the Struts capabilities. Additional details can be found on the Apache Struts project page (<http://struts.apache.org>).

Creating a Web Project with Struts Support

Struts support can be added during project creation when using the New Web Application wizard.

Struts support is enabled by selecting the Struts 1.2.7 checkbox in the wizard (see [Figure 9-6](#)). This allows additional Struts specific information to be specified:

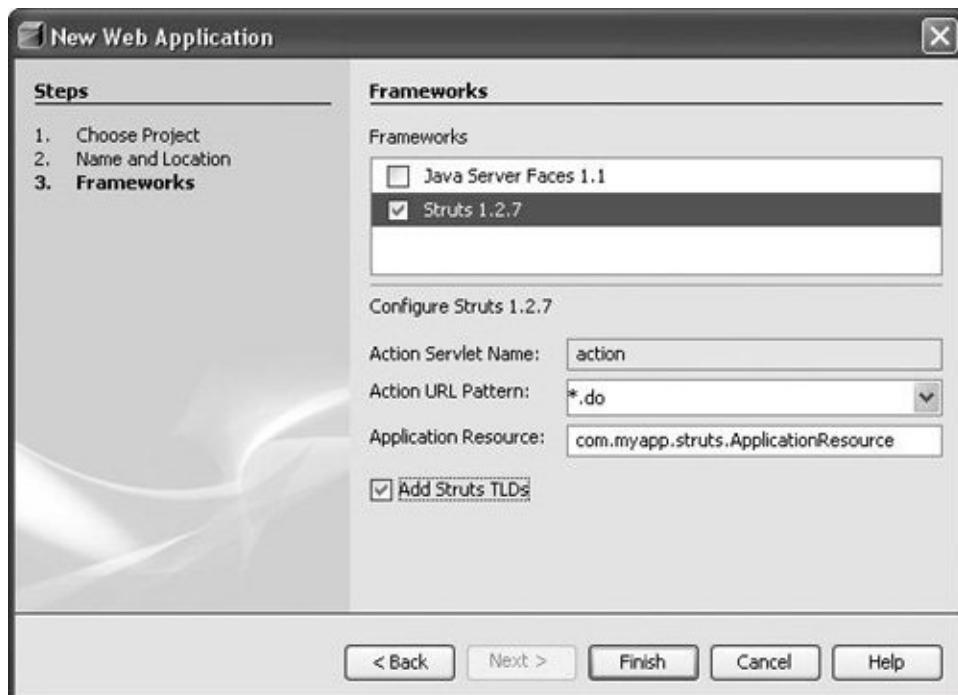
- The Action Servlet Name text field specifies the name of the servlet entry for the Struts action servlet to be specified.

The `web.xml` deployment descriptor contains a servlet entry for the action servlet, specifying the appropriate Struts specific parameters, such as the name of the servlet class and the path to the `struts-config.xml` configuration file.

- The Action URI Pattern combo box allows the appropriate patterns that should be mapped to the Struts action controller to be specified. This generates a corresponding `web.xml` servlet mapping entry to map the specified URI pattern to the action servlet.
- The Application Resource text field provides the capability to specify the resource bundle that will be used in the `struts-config.xml` file for localizing messages.
- The Add Struts TLD checkbox provides the capability to generate tag library descriptors for the Struts provided tag libraries. A tag library descriptor is an XML document that contains additional information about the entire tag library and each individual tag.

Figure 9-6. Struts support in the New Web Application wizard

[\[View full size image\]](#)



After you complete the wizard, the standard web application project is augmented to support Struts development as follows (see [Figure 9-7](#) for the Struts enabled web application project):

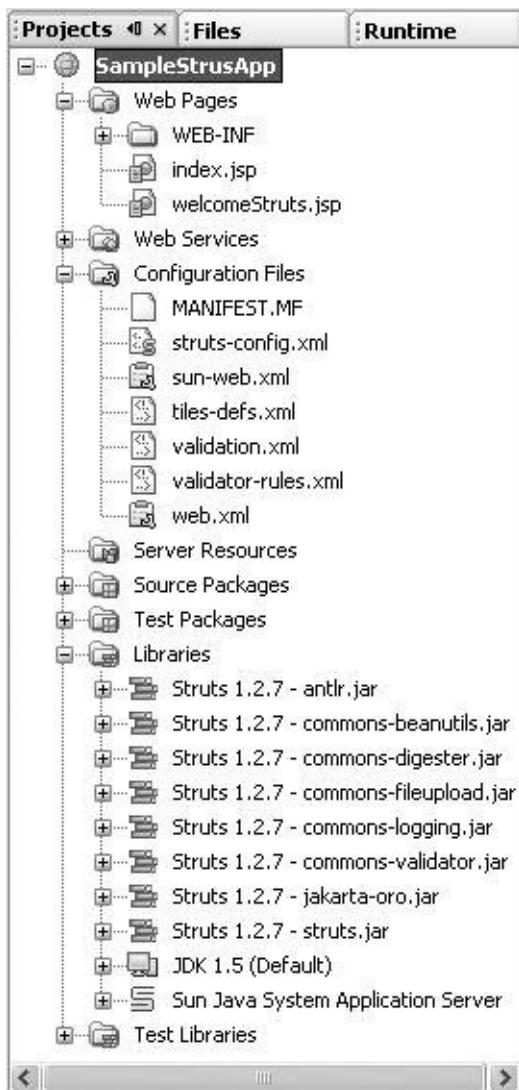
- `struts-config.xml`, `tiles-defs.xml`, `validation.xml`, and `validatorrules.xml` files are generated in the configuration files area. The `strutsconfig.xml` file is populated with the resource bundle specified in the wizard, a single global forward that references a `welcomeStruts.jsp` page, a controller entry for the tiles configuration file (`tiles` is a templating library), and a single action mapping (which provides the mapping to the generated struts welcome JSP). The other configuration files provide template files that can be used to start using additional Struts capabilities.
- The default `index.jsp` is augmented to refer to the `welcomeStruts.jsp` page using the appropriate action.

- The Struts libraries are added to the project and can be viewed in the Projects window from the project's Libraries node. The libraries are included by default when packaging the WAR file in the `WEB-INF/lib` directory.

If the libraries have already been deployed to the server or will instead be deployed as part of a Java EE application, the Project Properties dialog box can be used to control which libraries will be packaged with the WAR file.

- The `web.xml` file references the Struts action servlet.
- The resource bundle specified in the wizard is generated and populated with some default messages.
- If tag library descriptor generation was specified as part of the wizard, the tag library descriptors are generated.

Figure 9-7. Projects window with a web application project using the JSF framework

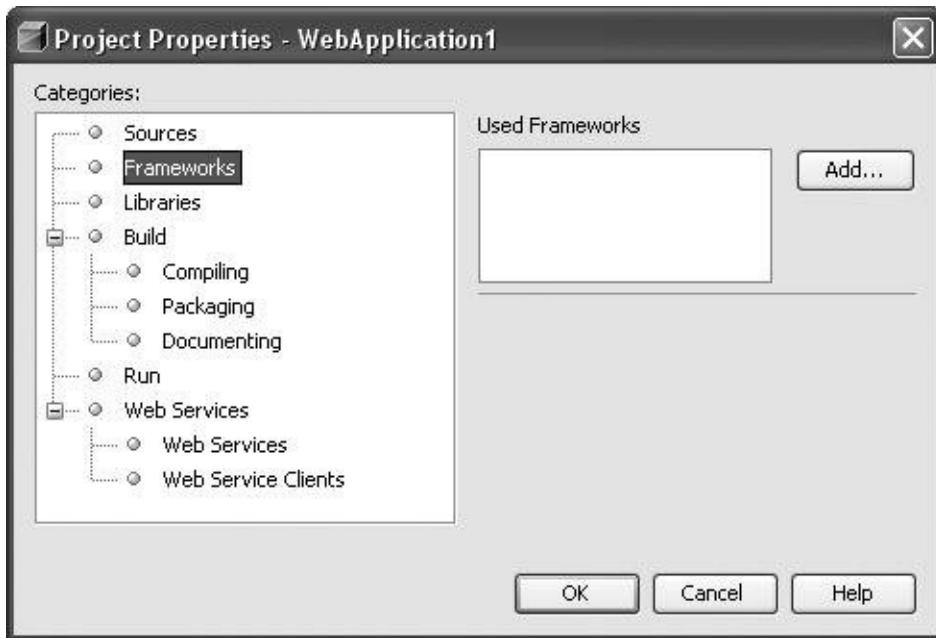


After you complete the wizard, code completion is enabled for Struts tag libraries, Struts Java classes, and Struts-related configuration files.

Adding Struts support can also be done after a project has been created, through the Project Properties dialog box (see [Figure 9-8](#)), which you can open by rightclicking the project's main node and choosing Properties.

Figure 9-8. Project Properties dialog box,

Frameworks tab



The Add button is used to select the desired web application support. The behavior after the initial launch is similar to what is done for a new project.

Action Support

An action class receives a request (as an `ActionFormBean` class), typically invokes business tier logic, and finally forwards control to the appropriate view object. The IDE provides a wizard to automate creation of action classes.

To create an action class:

1. Open the New Struts Action wizard by right-clicking the node of a Struts-enabled web application project and choosing New | File/Folder, selecting the Web category, and

selecting the Struts Action template.

2. In the Name and Location section of the wizard (shown in [Figure 9-9](#)), specify the name and package of the generated action class. The Struts configuration file and the action path must also be specified. The superclass for the generated action class can be selected from the combo box. The choices are:

org.apache.struts.action.Action. An action provides the capability to map incoming requests to the appropriate business logic. The Struts controller will select an appropriate action for each request. This class serves as the superclass for the other classes and the controller logic is implemented in the execute method.

org.apache.struts.actions.DispatchAction. A dispatch action uses the method specified in the request parameter named in the parameter property of the ActionMapping class. This allows a single action class to contain multiple methods for dispatching. The parameter property (shown in [Figure 9-10](#)) specifies the request parameter, which must match a method in the class and have the same signature as the execute method.

Figure 9-9. Name and Location panel of the New Struts Action wizard

[[View full size image](#)]

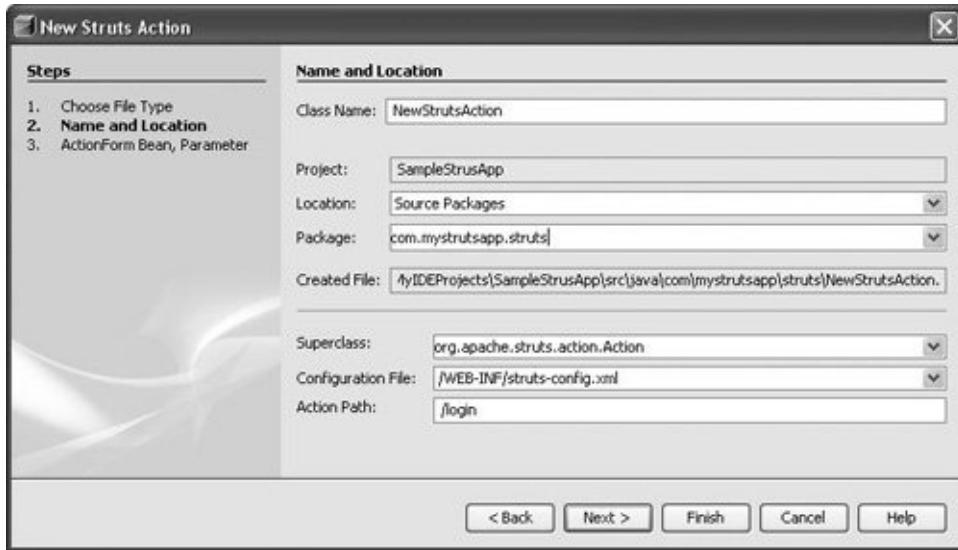
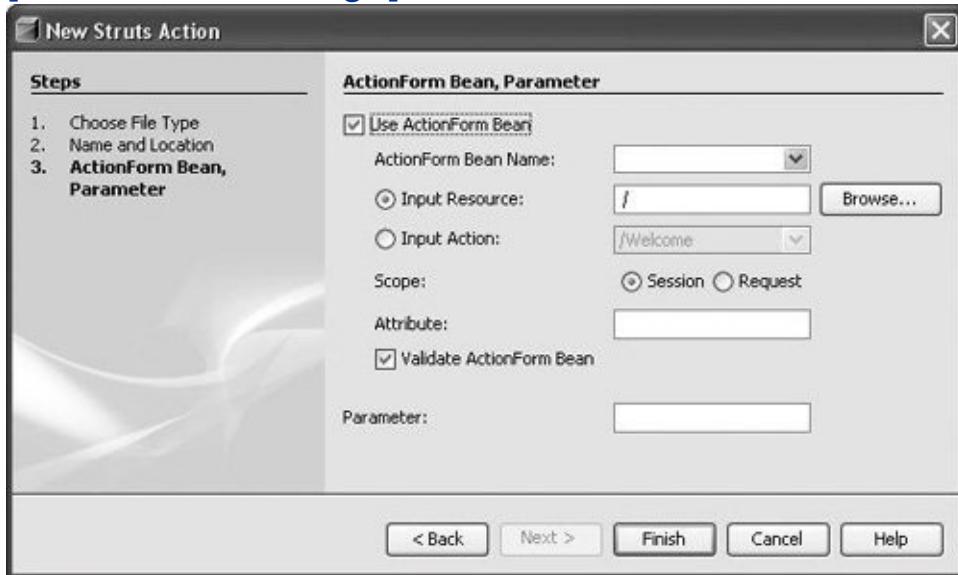


Figure 9-10. New Struts Action Wizard, ActionForm Bean and Parameter page

[View full size image]



org.apache.struts.actions.MappingDispatchAction. A mapping dispatch action is similar to the dispatch action in that a different method can be used to dispatch the request. The difference is that the mapping dispatch action specifies the name of the method in the parameter property

instead of the request parameter. This approach does not expose this mapping technique through application visible request parameters. The parameter property (shown in [Figure 9-10](#)) specifies the method name for dispatch.

org.apache.struts.actions.LookupDispatchAction. A lookup dispatch action supports multiple submit buttons with the same name. The parameter mapping property specifies the name of the button. The value of the request parameter with the same button name is used to locate the key in the application resource bundle. The action class must provide a map between the button key and the method name in the class to dispatch.

3. The last and optional step in the wizard is the ActionForm Bean and parameter step (see [Figure 9-10](#)), which allows you to specify the action form bean corresponding to the action. This step collects the following additional information used during the generation of the action mapping sections of the Struts configuration file.

The ActionForm Bean name specifies the [ActionBean](#) that should be used as input to the action. This list is populated from the action bean section of the Struts configuration file.

The Input Resource and Input Action radio buttons determine where control is returned if a validation error occurs.

The Scope radio buttons determine whether the ActionForm bean should be added using session or the request scope.

The Attribute text field specifies the name of an attribute (in the scope defined above) where the ActionForm bean should be stored.

The Validate checkbox specifies whether the validate method should be called prior to calling the action object.

The Parameter text field allows an action specific context to be specified. The appropriate context for each action type is discussed in step 2.

ActionForm Support

An ActionForm bean provides a Java representation of an HTML form. The Struts framework provides the binding from request parameters to property methods. The ActionForm binding is performed using JavaBeans conventions to determine the appropriate method for each request parameter. The IDE provides a wizard to automate creation of an ActionForm bean.

To create an ActionForm bean:

1. Open the New Struts ActionForm wizard by right-clicking the node of a Struts-enabled web application project and choosing New | File/Folder, selecting the Web category, and selecting Struts Action Struts ActionForm Bean.
2. In the Name and Location section of the wizard (shown in [Figure 9-11](#)), specify the name and package of the generated `ActionForm` class. The Struts configuration file must also be specified. The superclass for the generated `ActionForm` Form class can be selected from the combo box. The choices are:

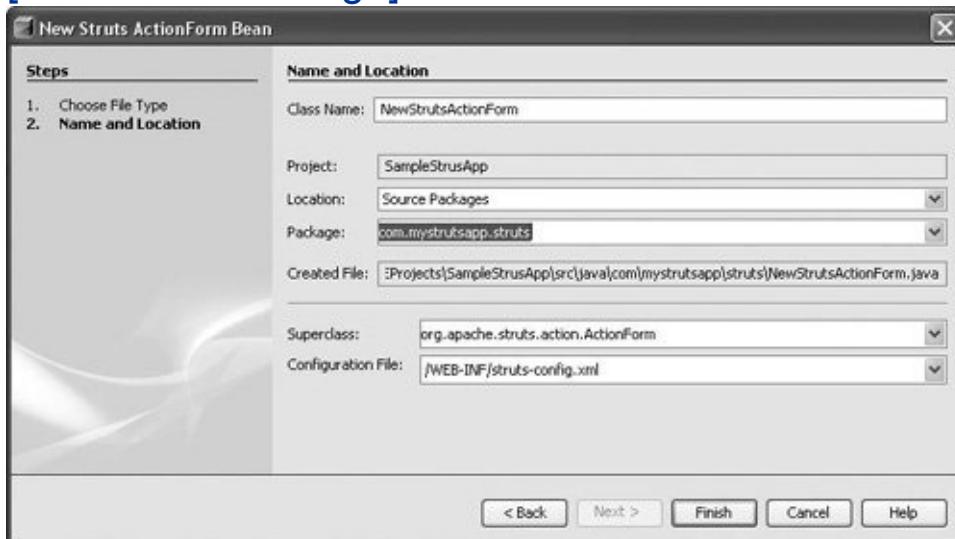
`org.apache.struts.action.ActionForm`. An ActionForm class may be associated with ActionMapping classes and represents the form data. An ActionForm class has several integration points into the Struts framework, notably the capability to reset the form data back to the default values and provide a way to validate user input. The ActionForm class is subclassed and must adhere to the JavaBeans convention for the mapping from the form data.

org.apache.struts.validator.ValidatorForm. The validator form provides the capability to validate properties based on the `validate.xml` file. The `validate.xml` file uses the name of the action to match the form name in the `validate.xml` file. Adding Struts support using the wizards generates the appropriate entries into the `struts-config.xml` file to enable validation form support, which includes the resource bundle keys required by the `validate.xml` file.

org.apache.struts.validator.ValidatorActionForm. The validator action form provides the capability to validate properties based on the `validate.xml` file. Similar to `ValidatorForm`, `ValidatorActionForm` uses the action path as the key into the `validate.xml` file form element's name. Using the specified Struts wizard adds the appropriate entries to enable the validator capabilities. This includes the resource bundle keys required by the `validate.xml` file.

Figure 9-11. New Struts ActionForm wizard

[[View full size image](#)]



3. Click Finish. The wizard will generate the ActionForm classes along with an entry in the form beans section of the Struts

configuration file. The generated file contains several TODO comments, which can be displayed using the To Do window.



The Java node for the ActionForm class provides a wizard to help generate properties. A quick way to navigate to this node is to press Ctrl-Shift-1 from the Source Editor. The Bean Patterns node is a subnode that provides the capability to add various JavaBeans properties.

Struts Configuration File Support

In addition to code completion, the Struts configuration file provides several wizards launched from the context menu of the Source Editor. The wizards provided include:

- **Add Action.** The Add Action wizard collects the information necessary to generate an action element in the action mapping section. This wizard is used in conjunction with an existing action class and supports binding of an ActionForm bean.
- **Add Forward/Include Action.** These actions provide the capability to redirect to another action, possibly using a resource bundle. This feature can be used to change application behavior without modifying source code.
- **Add Forward.** A forward provides a way to associate a logical name with an existing action. The Forward wizard supports both global and local forwards and the capability to determine the forward through a resource bundle.

- **Add Exception.** An exception allows a mapping between a specific exception and an action to be specified. The exception can be specified either globally or locally and can be directed through a resource bundle.
- **Add ActionForm Bean.** Form Beans provide a mapping between user view forms and Java code and serve as input to actions. This wizard supports both strongly typed beans as well as dynamic beans (which support `java.util.Map` style access to view parameters).
- **Add ActionForm Bean Property.** The `DynaActionForm` class support allows property-based configuration of these classes using properties. This wizard supports both indexed and scalar values.



When you open the Struts configuration file in the Source Editor, you can navigate to externally referenced artifacts (such as Java and JSP files) and internal references (such as form beans). Jumping to the source code is achieved by holding down the Ctrl key, moving the mouse over the identifier until it is underlined in blue, and then clicking it. For example, holding down the Ctrl key and moving the mouse over an action's input JSP name causes the JSP name to be underlined in blue (see [Figure 9-12](#)). Clicking the underlined identifier opens the JSP file in the Source Editor.

Figure 9-12. Hyperlinking in `struts-config.xml` file

```
|   <action-mappings>
|     <action path="/login" type="com.mystrutsapp.struts.New">
|       <action path="/Welcome" forward="/welcomeStruts.jsp"/>
|     </action-mappings>
```

Chapter 10. Introduction to Java EE Development in NetBeans IDE

- [Configuring the IDE for Java EE Development](#)
- [Java EE Server Support](#)
- [Getting the Most from the Java BluePrints Solutions Catalog](#)

THE JAVA PLATFORM, ENTERPRISE EDITION (JAVA EE) DEFINES the standard for developing multi-tier enterprise applications. The Java EE platform simplifies enterprise applications by basing them on standardized, modular components; by providing a complete set of services to those components; and by handling many details of application behavior automatically, without complex programming. The Java EE platform is targeted for developers who want to write distributed transactional applications for the enterprise and leverage the speed, security, and reliability of server-side technology.

NetBeans IDE has comprehensive support for the Java EE developer. There are advanced wizards to create entire Java EE applications and individual components, such as web applications, servlets, JavaServer Faces applications, Enterprise JavaBeans modules (EJB modules), and Enterprise JavaBeans components (enterprise beans), and web services.

In addition, the IDE provides a complete runtime environment based on the Sun Java System Application Server. NetBeans IDE 5.0 supports versions 8.1 and 8.2 of this application server, which provide the reference implementation of the J2EE 1.4 SDK. This application server, which is available separately or as part of a technology bundle with NetBeans IDE, is a J2EE 1.4-compliant application server that is free for development,

deployment, and redistribution. It offers the ideal companion to the IDE for all the developers who need an integrated environment in which complete Java EE applications can be developed, built, assembled, deployed, and debugged.

NetBeans IDE 5.0 also provides out-of-the-box support for BEA WebLogic Application Server 9.0 and JBoss Application Server 4.0.3.

In NetBeans IDE 5.0, the support is targeted mainly toward the J2EE 1.4 SDK. In subsequent NetBeans IDE releases, support will be added for Java EE 5 SDK.

Configuring the IDE for Java EE Development

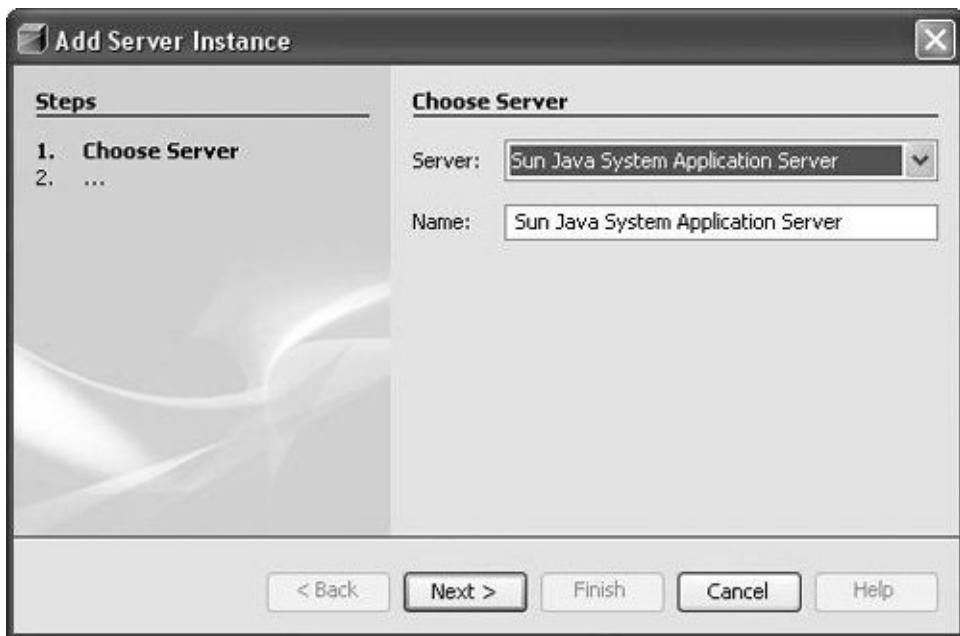
To explore all the capabilities of NetBeans IDE related to Java EE application and web services development, you need to make sure your environment is correctly configured. If you have downloaded the bundle containing both NetBeans IDE and Sun Java System Application Server, you have a preconfigured development environment, and you can skip the following steps. If you have downloaded a standalone NetBeans IDE, you will also need to install Sun Java System Application Server. Version 8.2 is available from <http://java.sun.com/j2ee/1.4/download.html>. This application server is the core of the J2EE 1.4 SDK and is free for development, deployment, and redistribution.

Before you can deploy an enterprise application, web application, JSP page, servlet, or EJB module, the server to which you are going to deploy needs to be registered with the IDE. By default, only the bundled Tomcat web server is registered with the IDE and this is not a complete Java EE server.

To register a Sun Java System Application Server instance:

1. In the IDE, choose Tools | Server Manager.
2. In the Server Manager, click the Add Server button. The Add Server Instance wizard (shown in [Figure 10-1](#)) appears and displays the types of servers that are compatible with the IDE.

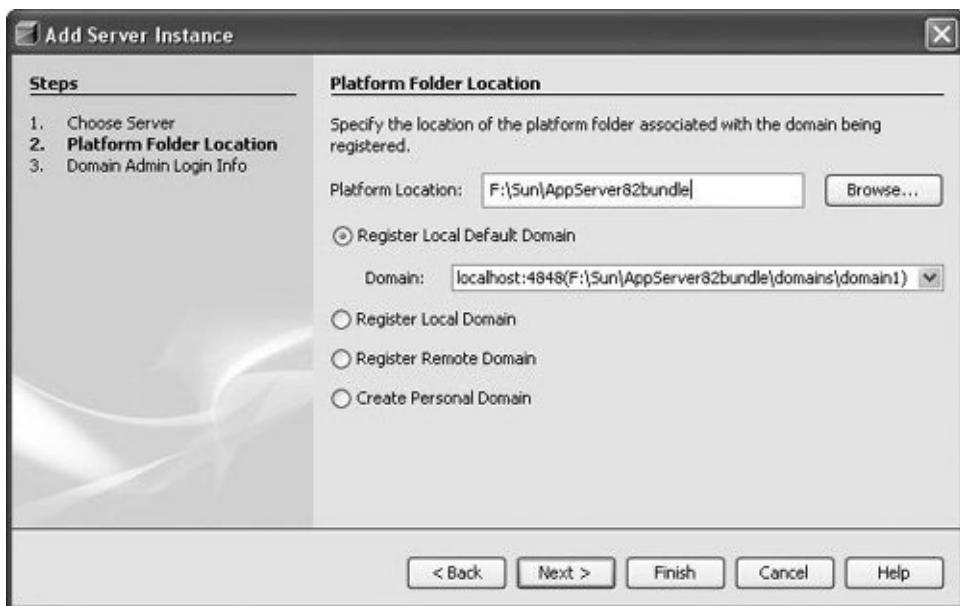
Figure 10-1. Add Server Instance wizard, Choose Server page



3. On the Choose Server page of the wizard, select the type of server you want to register (the type called "Sun Java System Application Server" works for versions 8.1, 8.2, and version 9.0 aka GlassFish Server) and click Next.
4. On the Platform Folder Location page (shown in [Figure 10-2](#)), specify the local installation of the server and keep the default configuration for registering a local default application server.

**Figure 10-2. Add Server Instance wizard,
Platform Folder Location page**

[[View full size image](#)]



5. On the Domain Admin Login Info page of the Add Server Instance wizard, specify server-specific information in the panels that follow and click Finish. (Remember that the Admin username is **admin** and the default password is **adminadmin**, unless you picked a different one at installation time.)

When you register a server in the IDE, you make its libraries available for production, deployment, or both. If you deploy your applications to a remote instance of the Sun Java System Application Server, its libraries are available at runtime. However, during development, you might need a local instance of this server. You can have multiple instances of the Sun Java System Application Server registered with the IDE. Once you have registered your local instance, you can open the wizard again to register remote instances (by entering a remote machine name and its port number in the Platform Location field of the Platform Folder Location page of the wizard).

When a server is registered in the IDE, you can see its node in the Runtime window under the Servers node. When you create a project in the New Project wizard, you select the server to which you want to deploy your application. After you create the

application, you can change the server by right-clicking the project, choosing Properties, clicking the Run node, and selecting a different server.

Now you are ready to create web services, enterprise beans, and Java EE applications.

Java EE Server Support

NetBeans IDE 5.0 works out-of-the-box with Sun Java System Application Server 8.1 and 8.2 (and pre-releases of version 9.0), and BEA WebLogic Application Server 9.0 and JBoss Application Server 4.0.3. [Table 10-1](#) illustrates the different features for each target server.

Table 10-1. Java EE Server Features Supported Directly in NetBeans IDE 5.0

Description	Sun AS 8.1, 8.2	BEA WebLogic 9	JBoss 4.03
Start and stop the server (local)	yes	yes	yes
Deployment and redeployment	yes	yes	yes
Remote deployment	yes	no	no
Undeployment	yes	no	no
Remote undeployment	yes	no	no
Fast directory-based deployment (Web)	yes	yes	yes
Java debugging	yes	yes	yes
JSP source level debugging	yes	no	yes
Java EE Profiling (NetBeans Profiler must be installed)	yes	yes	yes
View console output (for local server)	yes	no	no
View log files	yes	yes	yes

Edit server's conf file (AS8.1: via favorites, and xml editor)	yes	no	no
List deployed applications	yes	no	no
List sub-elements in deployed applications, and display properties	yes	no	no
List/Edit registered server resources	yes	no	no
Admin UI	yes	yes	yes
HTTP monitoring	yes	no	no
Generate server-specific CMP and JNDI data (no manual steps needed)	yes	no	no
Visual editing of server-specific data	yes	no	no
View servlet generated from JSP files	yes	no	no
Complete Web Services support	yes	no	no
Complete EJB CMP Mapping tool	yes	no	no
Zero config support (auto creation/registration of resources)	yes	no	no
Out-of-the-box Blueprints solutions working	yes	no	no
Single bundle, easy to install with NetBeans	yes	no	no
Server JVM options configuration	yes	no	no

Server specific DD XML code completion/ validation	yes	no	no
Java EE Verification (AS8.x is required)	yes	no	no
Server-specific resource creation wizards and registration	yes	no	no
Secure Server (HTTPS) admin access and certificate validation	yes	no	no
JSR 88 graphical configBean class implementation	yes	no	no
Complete support for server-specific Ant tasks	yes	no	no
Out-of-box JSF support (Java Server Faces xml code completion/validation, lib registration from app server area, etc.)	yes	no	no
Pointbase or Derby DB integration (start/ stop menu, driver, samples) (only when AS8.x is installed and registered)	yes	no	no
Pointbase or Derby driver pre-configure for Server runtime	yes	no	no
AVK (Application Verification Kit) ready (via extra AVK plug-in module from NetBeans Update Center)	yes	no	no

The registration of these servers is similar to the mechanism used for the Sun Java System Application Server. To register a

server:

- 1.** Choose Tools | Server Manager and click the Add Server button.
- 2.** In the Server drop-down list, select either BEA WebLogic Application Server 9.0 or JBoss Application Server 4.0.3.
- 3.** In the Server Location page of the wizard, specify where the server is installed.

After you click Finish, the server is registered in the IDE.

As a preview feature in NetBeans IDE 5.0, it is also possible to register a build of the Glassfish Java EE 5 server (using the Sun Java System Application Server choice). A post-5.0 version of NetBeans IDE will provide comprehensive support for the Java EE 5 specification.

Also, by the time you read this, it might be possible to register an instance of the WebSphere 6 application server in the IDE. If there is not an option for the WebSphere server in the Server Manager, connect to the IDE's Update Center (Tools | Update Center) to see if there is a plug-in module available that supports working with WebSphere.

Getting the Most from the Java BluePrints Solutions Catalog

NetBeans IDE provides a unique capability for learning and understanding best practices for Java application development with its integration of the Java BluePrints Solutions Catalog. The Java BluePrints Solutions Catalog has long been accepted as the source of Java application best practices and Java suggested guidelines. The Java BluePrints Solutions Catalog also illustrates these best practices and guidelines through various example applications. It provides a huge repository of example applications from which you can literally cut and paste source code or tailor code for your own specific application.

In NetBeans IDE, you can directly access the catalog and install example Java BluePrints Solutions directly into the IDE as a new project. This feature provides you a unique opportunity to learn and understand quickly various Java BluePrints best practices and recommended guidelines.



The Java BluePrints Solutions Catalog is updated on an ongoing basis with new updates available for the IDE through the IDE's Update Center (choose Tools | Update Center).

In the NetBeans IDE 5.0 release, the following Java BluePrints Solutions are available, grouped into three categories:

- AJAX

- Using JSF with AJAX
- Auto-Completion
- Auto-Completion using a JSF component
- Progress Bar
- Progress Bar using JSF
- Realtime Form Validation
- Web Tier Design
 - Making Web Applications Accessible
 - Handling Command Submissions
 - Creating Tabbed View
 - Storing Session State on the Client
 - Server-side Validation
 - Client-side Validation
- Web Services Design
 - Accessing Web Services From J2EE Components
 - Accessing Web Services From a Stand-alone Java Client
 - Designing Document Oriented Services

- Using xsd:any to Represent XML Documents in a Service Interface
- Using xsd:anyType to Represent XML Documents in a Service Interface
- Using Attachments to Represent XML Documents in a Service Interface
- Using Schema-defined Types to Represent XML Documents in a Service Interface
- Using Strings to Represent XML Documents in a Service Interface
- ServiceLocator for Web Services Clients

Accessing the Java BluePrints Solutions Catalog

When you install NetBeans IDE, the Java BluePrints Solutions Catalog is available without any additional installation steps. You can access the catalog by choosing Help | Java BluePrints Solutions Catalog. The catalog is displayed in the IDE's main document area (where the Source Editor also appears) as shown in [Figure 10-3](#).

Figure 10-3. Java BluePrints Solutions Catalog in NetBeans IDE main window



Navigating the Java BluePrints Solutions Catalog

Because the Java BluePrints Solutions Catalog displayed in the NetBeans IDE main window follows the browser paradigm, it is easy to navigate. You will notice a drop-down list from which you can select different Java BluePrints solutions (see [Figure 10-4](#)).

Figure 10-4. Java BluePrints Solutions Catalog with the combo box listing the different solutions

available

[View full size image]



The solutions are grouped into categories, within which you can navigate by clicking the Back or Forward buttons. You can also use the drop-down list to select a specific solution within a category. When a specific Java BluePrints Solution is selected, there are three tabs to choose among, as shown in [Figure 10-5](#).

Figure 10-5. A blueprint with the Solution tab selected

[View full size image]

The screenshot shows a web browser window titled "Java BluePrints Solutions Catalog" under the "Sun Microsystems" logo. The page is labeled "Early Access". The main content area is titled "Accessing Web Services from J2EE Components". It includes author information ("Sean Brydon, Smitha Kangath") and a status indicator ("Status: Early Access"). A section titled "Problem Description" contains text about J2EE components accessing web services using JAX-RPC technology, mentioning three modes: stubs, dynamic proxies, and DII. Below this, a list of considerations includes "Portability" and "Handling exceptions".

The Solution tab displays a description of the issue the Java BluePrints Solution is trying to solve. The Design tab describes the design of the solution so you can understand the implementation decisions made and the design best practices used in the solution. The Example tab allows you to install an example implementation of the Java BluePrints Solution in NetBeans IDE as a NetBeans project.

To view the solution, design or install an example as a NetBeans project for a given Java BluePrints Solution, simply select the appropriate tab.

Creating an example project is a very useful capability, because it allows you to see a running, working example implementation of the Java BluePrints Solution by being able to run or even debug the solution. In addition, it is very easy to pull source

code from a working implementation into your own specific application or project.

Creating a NetBeans Project from a Java BluePrints Solution

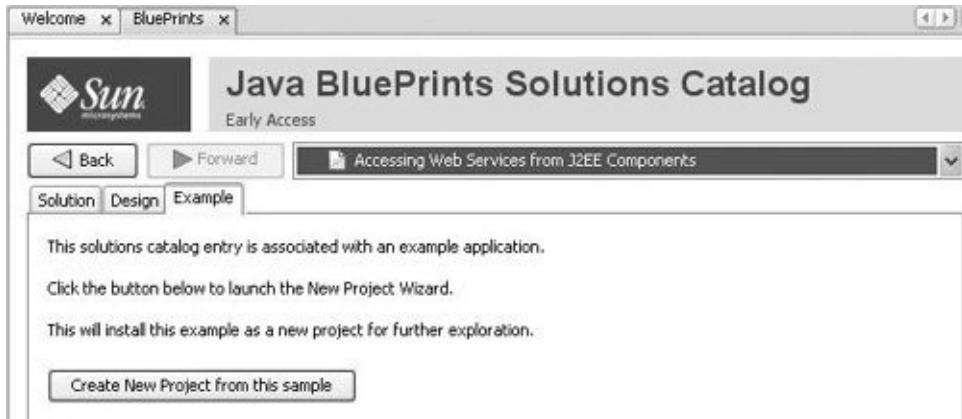
This section describes the steps for installing a Java BluePrints Solutions example in NetBeans IDE as a NetBeans project. In addition, the section describes how to run the example once it is installed as a NetBeans project.

To install a Java BluePrints Solution from the Java BluePrints Solutions Catalog:

1. Choose Help | BluePrints Solutions Catalog.
2. Once the catalog is displayed, select the solution you want to work with from the drop-down list in the display.
3. In the solution you have selected, click the Example tab. You will see a screen that looks similar to [Figure 10-6](#).

Figure 10-6. A solution with its Example tab selected

[\[View full size image\]](#)

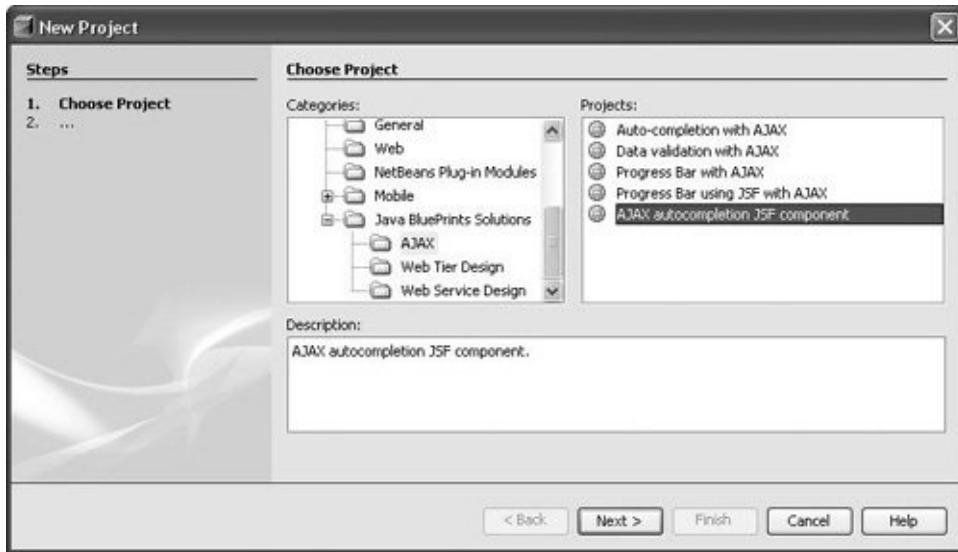


4. On the Example screen for the solution, click the Install Example button.

This will begin the installation and setup of a new NetBeans IDE project for the selected Java BluePrints Solution. The IDE's New Project wizard is launched with the Java BluePrints Solution you have chosen to install selected as the project. For example, the Auto-Completion using a JSF component project template would be chosen (as shown in [Figure 10-7](#)) if you clicked Install Example in the screen for the AJAX Auto-Completion using a JSF component solution.

Figure 10-7. New Project wizard with a Java BluePrints Solution selected

[\[View full size image\]](#)



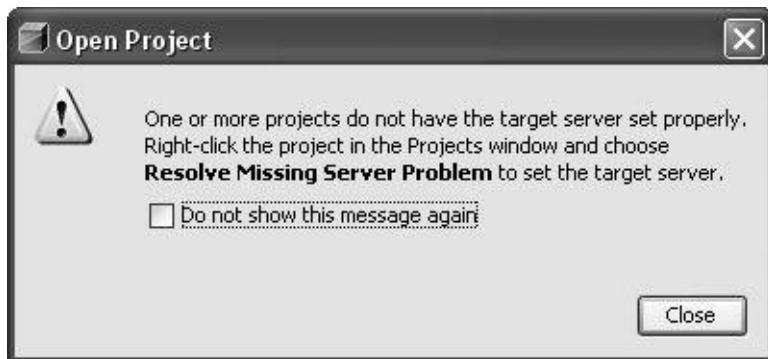
5. Click the Next button to continue the installation and setup of the Java BluePrints Solution.
6. The next screen of the wizard asks you to supply a project name (or to accept the default name) and a project location (or to accept the default) and asks whether to set this as the main project in NetBeans IDE.

In almost all cases, you should accept the defaults shown in the wizard. Only if you want to change the name or location of the project or set it as the main project should you change the default settings displayed in this wizard.

7. Click the Finish button to complete the installation and setup of the Java BluePrints Solution as a NetBeans IDE project.

After completing the wizard, you might see the warning dialog box shown in [Figure 10-8](#). This generally occurs if you do not have the Sun Java System Application Server registered with the IDE. It is possible to set the project to work with a different server (such as Tomcat), but some of the features in the application might not work if the server is not fully J2EE 1.4 compliant.

Figure 10-8. Warning dialog box that appears if the IDE does not detect an appropriate server for the project



If you see the warning dialog box, follow the instructions it gives. After clicking OK to close the dialog and open the project, right-click the newly created project in the Projects window and choose Resolve Missing Server Problem. The Resolve Missing Server Problem dialog box (shown in [Figure 10-9](#)) appears.

Figure 10-9. Resolve Missing Server Problem dialog box



8. In the Resolve Missing Server Problem dialog box, select a target server. If you have a Sun Java System Application Server, JBoss Application Server, or BEA WebLogic Application Server installed with NetBeans IDE, you will see options for selecting Sun Java System Application Server, JBoss Application Server, WebLogic Application Server, and Tomcat. Choose the desired server for your project and click OK.



If you would like to use the Sun Java System Application Server, JBoss Application Server, or WebLogic Application Server with the Java BluePrints Solution you are installing, but the server does not appear in the list of servers in the Resolve Missing Server Problem dialog box, click Cancel to exit the dialog box.

If necessary, download and install the application server. Then register the application server in the IDE's Server Manager (available through the Tools menu). After that, you can go back to the Resolve Missing Server Problem dialog box and select the application server you wish to use as the target server.

Once you have the Java BluePrints Solution created as a NetBeans IDE project, you can perform operations such as building, deploying, and debugging.



You can also install the available Java BluePrints Solutions as IDE projects straight from the New Project wizard. In the New Project wizard, you can expand the Sample folder category to show BluePrints Solutions folder. The BluePrints Solutions folder contains the same Java BluePrints Solutions as the ones in the catalog that have an Example tab.

Running a Java BluePrints Solutions Project

To run a NetBeans IDE project that has been created from the Java BluePrints Solutions Catalog, you perform the same operations as you would when running other Java EE applications in the IDEthat is, you open the Projects window, right-click the newly created project for your Java BluePrints Solution, and choose Run Project. NetBeans IDE will build the newly created Java BluePrints Solutions project, deploy it to the target server, and load the application's home page in your default web browser automatically.

Once you have the Java BluePrints Solution created as a project in NetBeans IDE, you can perform a large number of operations. For example, you can deploy the application to your target server, as you have already seen; you can run the application in a debugger; you can use the HTTP Monitor to analyze the HTTP requests that are passed between your

browser and the deployed application; and you can make changes to the source files by editing the project source.

In fact, using the Java BluePrints Solutions Catalog and one of its applications is an excellent way to learn some of the Java EE technologies and best practices on how to use the technologies. In addition, these solutions are an excellent source from which you can cut and paste code for an application you are developing.

Chapter 11. Extending Web Applications with Business Logic: Introducing Enterprise Beans

- [EJB Project Template Wizards](#)
- [EJB Module Structure](#)
- [Adding Enterprise Beans, Files, and Libraries to Your EJB Module](#)
- [Adding Business Logic to an Enterprise Bean](#)
- [Adding a Simple Business Method](#)
- [Enterprise Bean Deployment Descriptors](#)

FOR MANY NETBEANS IDE USERS, as well as web application developers, Enterprise JavaBeans (EJB) technology might be new or apparently complex. However, NetBeans IDE provides wizards and other features to make it easy to create enterprise beans and add business methods to them. Once these business methods are implemented (in Java code), they can be called either from other enterprise beans or from a web application's servlets or utility classes.

The benefits of encapsulating application code within EJB business methods are numerous:

- Enterprise beans support *transactions*, the mechanisms that manage the concurrent access of shared objects. Transaction settings are declarative, via the deployment

descriptor files.

- Enterprise beans can be used by many clients, across machines or not (remote and/or local access).
- Enterprise bean business methods can be secured declaratively, without source code modification.
- Enterprise beans access external resources such as databases, message queues, mail sessions, and web services declaratively via Java Naming and Directory Interface (JNDI) naming. The JNDI naming service enables components to locate other components and resources. To locate a Java Data-base Connectivity (JDBC) resource, for example, an enterprise bean invokes the JNDI `lookup` method. The JNDI naming service maintains a set of bindings that connects names to objects. The `lookup` method passes a JNDI name parameter and returns the related object.

See [Table 11-1](#) for a list of all of the enterprise bean types.

Table 11-1. Types of Enterprise Beans

Enterprise Bean Type	Description
Session	Performs a task for a client or implements a web service. A session bean can be stateful for conversation handling between the client (the user of the business logic) and the server, or stateless.
Entity	Represents a business entity object that exists in persistent storage, typically SQL databases (and possibly others).
Message-Driven	Acts as a listener for the Java Message Service API, processing messages asynchronously.

EJB Project Template Wizards

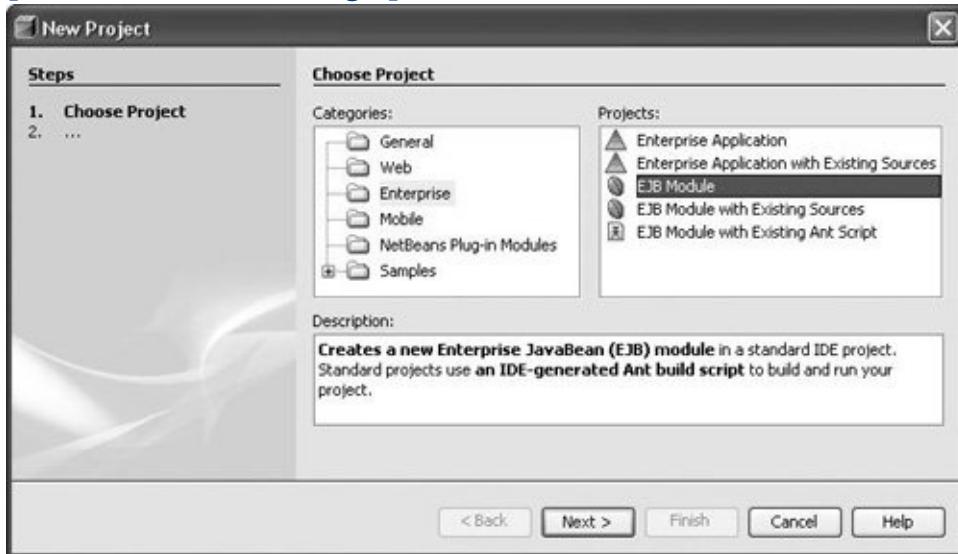
The first thing to do in developing enterprise beans is to create an EJB Module project that can contain one or more enterprise beans. Whereas a web application is a deployable Java EE component containing a collection of servlets, web pages, and JSP files, an EJB module is a deployable Java EE component that contains a collection of enterprise beans.

To create an EJB Module project:

1. Choose the EJB Module project template, under the Enterprise category (as shown in [Figure 11-1](#)).

Figure 11-1. New Project wizard with EJB Module project type selected

[[View full size image](#)]



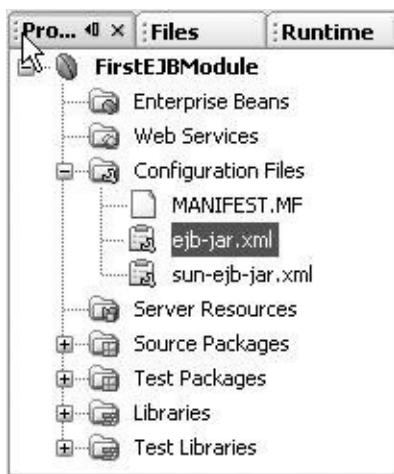
2. On the Name and Location page of the wizard, specify the location of the project, its name, and whether you want to add this EJB module to an existing enterprise application

(EAR) project.

You can add the module to an enterprise application project later, such as when you create the enterprise application project.

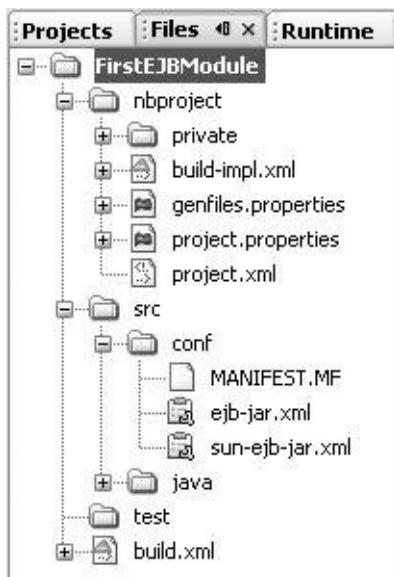
Once you complete the wizard, your project is created and visible in the Projects window, as shown in [Figure 11-2](#). For now, it contains no enterprise beans. The section Adding Enterprise Beans, Files, and Libraries to Your EJB Module later in this chapter explains how to populate the module.

Figure 11-2. Projects window with EJB Module project showing



You can open the Files window (shown in [Figure 11-3](#)) to see which directories and files have been created on disk. See the EJB Module Structure later in this chapter for a description of the conventions used for this structure.

Figure 11-3. Files window with EJB Module project showing



EJB Module Structure

The Java BluePrints Solutions Catalog provides guidelines on how to structure your enterprise applications and EJB modules to ensure that they work properly with different application servers. When you create a project in the IDE, this Java BluePrints convention structure is respected.

Following is a quick description of the structural elements of the built EJB module:

- The `src/java` folder, which contains all the Java source files in the application.
- The `src/conf` folder, which contains the enterprise deployment descriptors and the application server's specific deployment descriptors.
- The `setup` directory, which contains server-specific resource files. This folder does not appear until you have added any of these resources (through the Server Resources node in the Projects window or via the server's administration features).

You can find additional information on these conventions at
<https://conventions.dev.java.net>

See [Table 11-2](#) for information on how the various source elements of an EJB module map to their representation in the IDE and where they end up in the deployed component.

Table 11-2. Matrix of EJB Module Elements and Their Representation in the IDE

Content	Representation in the Projects Window	Representation in the Files Window	Location within the Built EJB JAR File (located in the dist folder)
Enterprise beans	Enterprise Beans node	<code>src/java</code> folder	Root of the file
Java source files,	Source	<code>src/java</code> folder	Package

helperclasses, enterprise bean Java files, etc.	Packagesnode		structure forthe JAR file
Unit tests	Test Packagesnode <code>test</code> folder	N/A	
Deployment descriptor(<code>ejb-jar.xml</code> , <code>webservices.xml</code>)	ConfigurationFiles <code>src/conf</code> folder node		META-INF folder
Application server's deployment descriptors (<code>sun-ejb- jar.xml</code> , <code>sun-cmp- mapping.xml</code> , others for JBoss or WebLogic servers)	ConfigurationFiles <code>src/conf</code> folder node		META-INF folder
Application server specific resources or scripts (such as SQL)	Server Resourcesnode (visible when some enterprise resources exist there)	<code>setup</code> folder	N/A. The resources in this folder are registered automatically at deployment time for the Sun Application Server target.
Web services	Web Services node <code>src</code> area (Java code)		Package structure for the JAR file
Libraries	Libraries node	Location of the <code>libraries</code> folder	JAR libraries included in the EJB module JAR file, at the top location.
Test classpath entries	Test Librariesnode <code>test</code> folder		N/A
Project metadata, including build script	Project Propertiesdialog box, which you can open by right- clicking the project's node and	<code>build.xml</code> file, <code>nbproject</code> folder	N/A

choosing
Properties.

EJB module build
area(*.class files)

Not shown

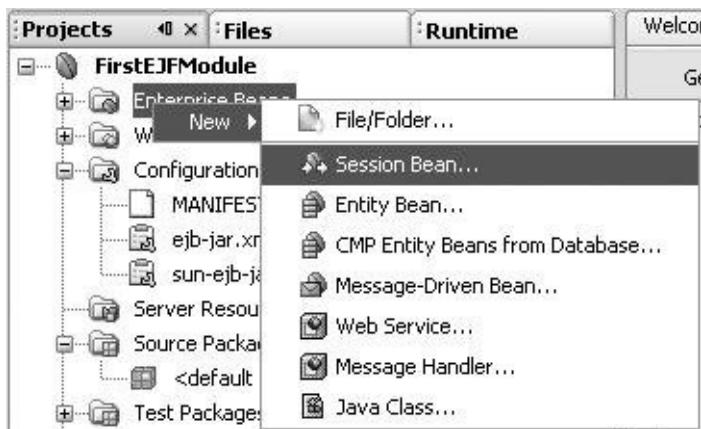
build and
build/generated

Main content for
the EJBmodule
archive JAR file.

Adding Enterprise Beans, Files, and Libraries to Your EJB Module

Once you have created an EJB Module project by using the New Project wizard, you can start populating it with new enterprise beans and helper Java classes. The most straightforward way to create files is to open the Projects window, right-click the node where you want to place the file, and choose New and then select a template from the submenu. A short wizard appears for the template, enabling you to set the name and other characteristics of the file. For example, choose the Session Bean template, as shown in [Figure 11-4](#).

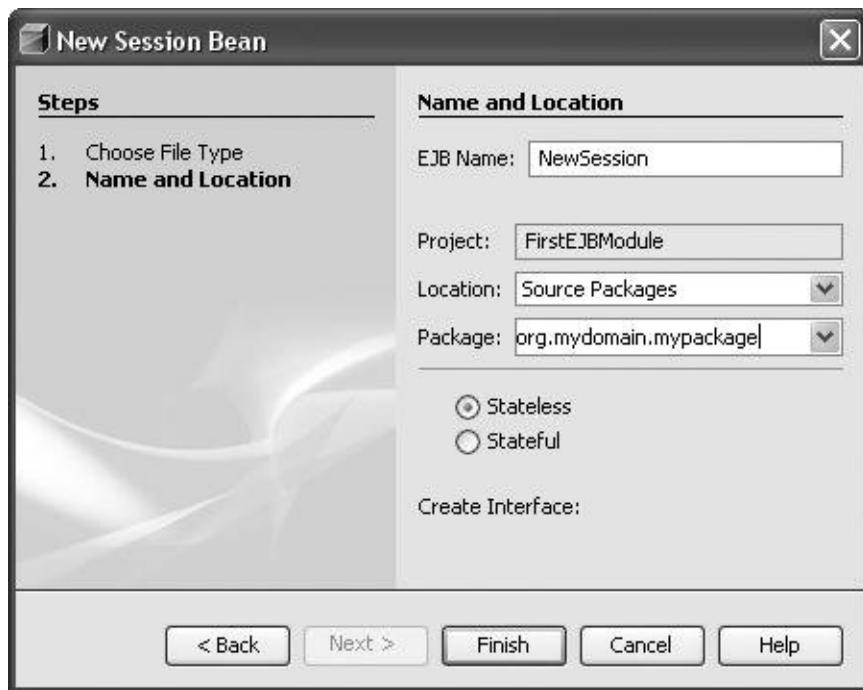
Figure 11-4. Adding a file to an EJB module



The wizard will create a session bean and add it to the EJB module. You can specify the bean name and package (make sure you select or create a Java package). You can specify whether this session bean will have a local and/or a remote interface (local is the default) and whether the bean is stateful or stateless. Accept the default values to create a local stateless

session bean, as shown in [Figure 11-5](#).

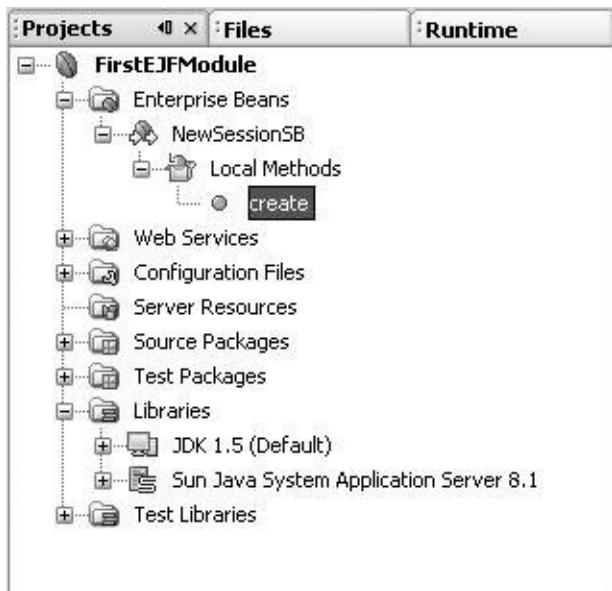
Figure 11-5. New Session Bean page of the New File wizard



Notice the new logical representation of the enterprise bean in the Projects window under the Enterprise Beans node (shown in [Figure 11-6](#)). This enterprise bean is a set of Java files (four for a simple session bean):

- The local interface (`BeanNameLocal.java`)
- The local home interface (`BeanNameLocalHome.java`)
- The bean implementation itself (`BeanNameBean.java`)
- The business interface (`BeanNameLocalBusiness.java`)

Figure 11-6. Projects window showing a new enterprise bean



The Projects window's logical view hides the complexity of this enterprise bean by showing only a single node that exposes some important methods for the bean, like the local methods.



"Where are my Java files?" When you work with an enterprise bean's node in the Projects window, you do not see all of the Java files and deployment descriptors that are part of the bean. NetBeans IDE keeps all of these files synchronized automatically when you work with the bean's node. If you want to see the individual Java files, you can browse them within the Source Packages node.

Adding Business Logic to an Enterprise Bean

In this section, you learn about the different method types you can add to an enterprise bean. Remember that NetBeans IDE provides wizards that greatly simplify the work of coding business logic within enterprise beans.

An enterprise application gets its work done through methods that the client calls on the bean. The following method types are either automatically generated via the wizard with the default implementation or can be added via popup menu actions in the Projects window or the Source Editor window for an enterprise bean implementation class. The IDE takes care of necessary extra generation (such as updating the local or remote interfaces with the correct signature entry).

- **Business methods.** A client calls business methods on a bean through the bean's remote interface (or local interface, as applicable).

You have to add business methods to the bean yourself; the IDE doesn't generate any default business method declarations. However, when you specify a business method, the IDE places matching method declarations in the bean class and in the remote, local, or remote and local interfaces.



Business methods are the most important methods for an enterprise bean. These are the ones called by other enterprise beans or web tier components, like JSP files or servlets. A special IDE wizard is available to simplify the coding of calling an EJB business method. From a servlet or enterprise bean file in the Source Editor, right-click to activate the popup menu, and choose Enterprise Resources | Call Enterprise Bean.

- **Life-cycle methods.** The container calls several methods to manage the life cycle of an enterprise bean. Depending on the type of bean, the container works through the methods in slightly different ways. You have the option of specifying parameters for some of these methods.

The IDE automatically generates the appropriate life-cycle method declarations for each type of bean and places them in the bean class.

- **Finder methods.** The client goes through the home interface to find an entity bean instance by its primary key. You can also add other finder methods.

The IDE automatically generates a `findByPrimaryKey` method declaration in the local home interface of every entity bean (and in the bean's home interface, if it has one). The IDE also places a corresponding `ejbFindByPrimaryKey` method declaration in the bean class of every entity bean that manages its own persistence (that is, a *bean-managed persistent entity bean*, or BMP entity bean). If you add another finder method, the IDE automatically places the corresponding method declarations in the local home (and home) interface and, for BMP entity beans, in the bean class.

An entity bean that delegates its persistence to the container is called a *container-managed persistent entity bean*, or CMP entity bean. Finder methods that are added to CMP entity beans include EJB Query Language (EJB QL) statements, which are converted automatically to the kind of SQL code the server needs.

- **Create methods.** The container initializes the enterprise bean instance, using the create method's arguments.

- **Home methods.** An entity bean can use a home method for a lightweight operation that doesn't require access to any particular instance of the bean. (By contrast, a business method does require access to a particular instance.) It is up to you to add the home method explicitly; the IDE generates the corresponding method declaration in the bean class and the bean's local home or home interface. An entity bean can have any number of home methods.
- **Select methods.** A CMP entity bean can use a select method. Like a finder method, a select method can query the database and return a local or remote interface or a collection. In addition, a select method can query a related entity bean within the same EJB module and return values from its persistent fields. Select methods aren't exposed in remote-type interfaces and can't be invoked by a client.
- **onMessage methods.** A client sends a message through a Java Message Service (JMS) destination to call an `onMessage` method on a message-driven bean.

Adding a Simple Business Method

To add a business method:

1. Choose the Add | Business Method popup menu item from an enterprise bean's node (as shown in [Figure 11-7](#)), or choose EJB Methods | Add Business Method by right-clicking in the Source Editor for a bean implementation class (as shown in [Figure 11-8](#)).

Figure 11-7. Projects window and adding a method to an enterprise bean

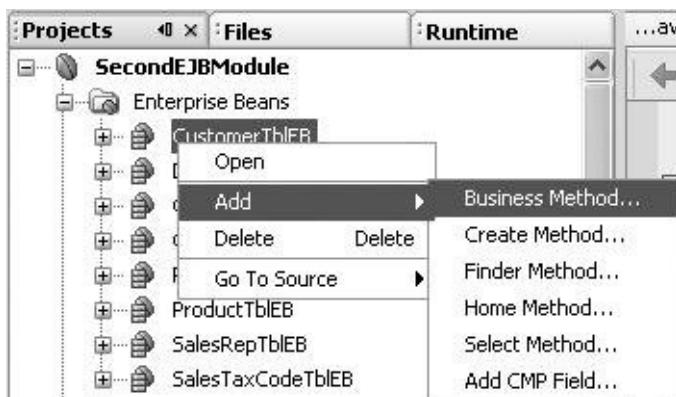
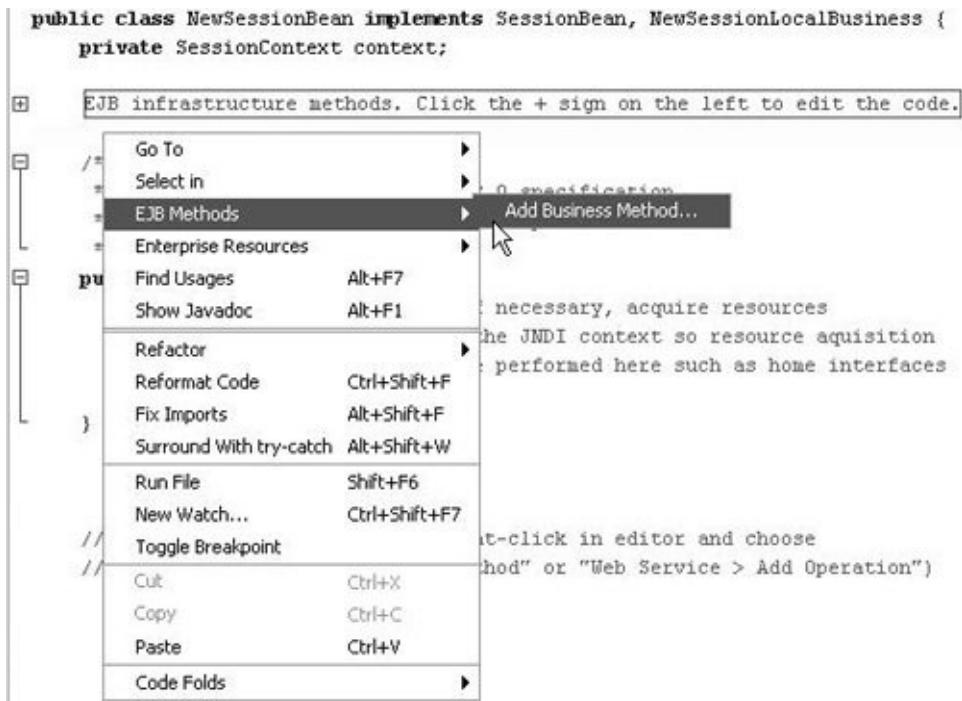


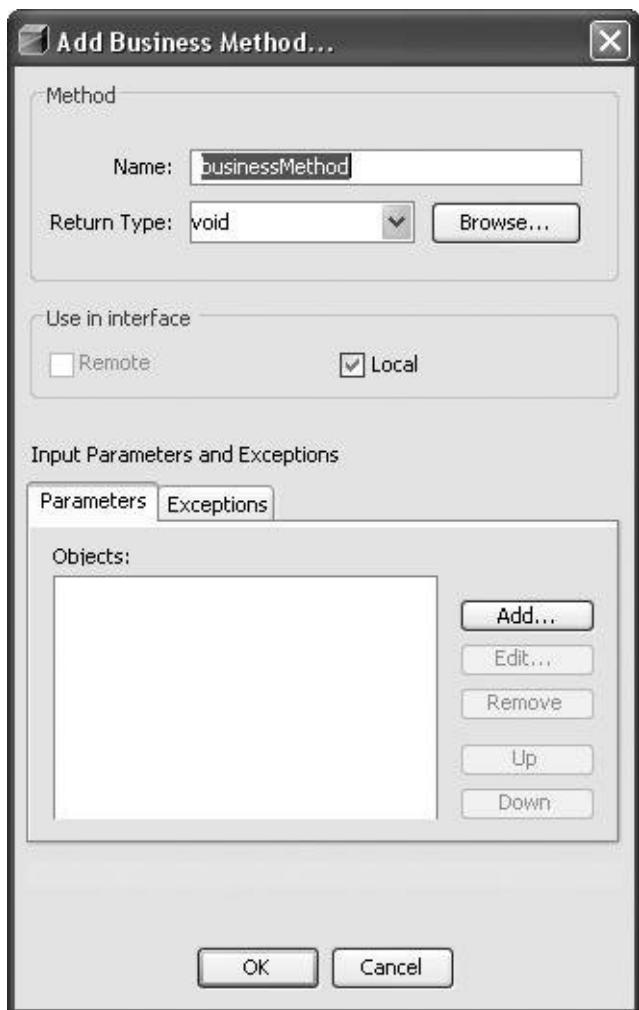
Figure 11-8. Source Editor and adding a method to an enterprise bean

[[View full size image](#)]



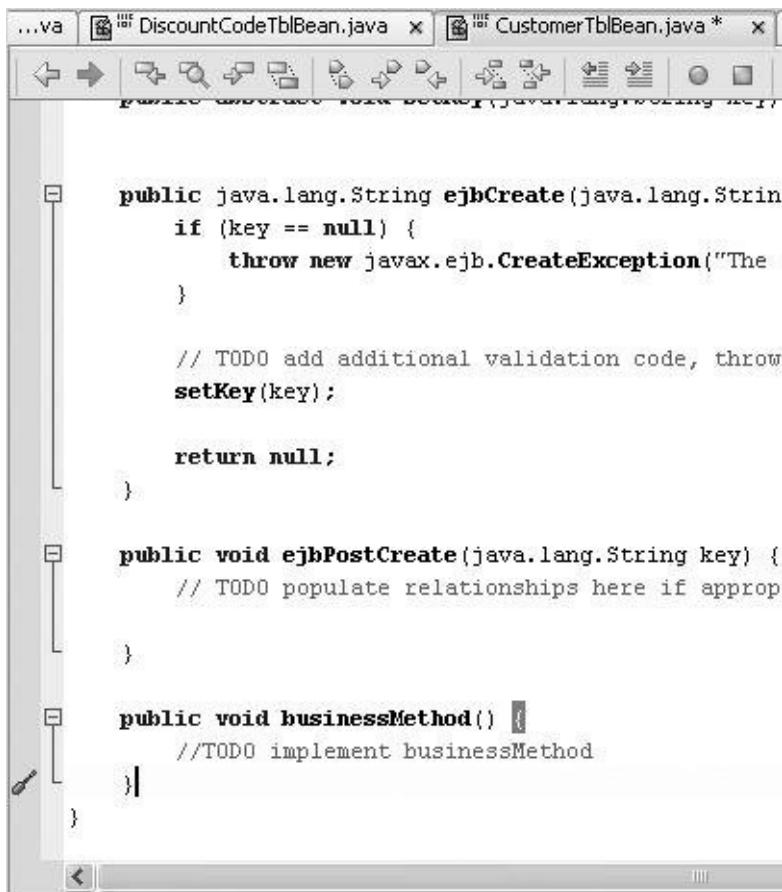
2. In the Add Business Method dialog box (shown in [Figure 11-9](#)), enter a method name, a list of parameters and their type, and possible exceptions that will be thrown by this business method.

Figure 11-9. Add Business Method dialog box



Notice the change inside the bean implementation Java file (as shown in [Figure 11-10](#)). It now contains the new business method body, and now you can use the capabilities of the IDE's Source Editor to finish the implementation of the method.

Figure 11-10. Source Editor with new business method added



The screenshot shows a Java code editor within an IDE. The code is for an EJB bean named `DiscountCodeTblBean.java`. It contains three methods: `ejbCreate`, `ejbPostCreate`, and `businessMethod`. The `ejbCreate` method includes validation logic for a key parameter. The `ejbPostCreate` method has a TODO comment. The `businessMethod` method is currently empty. The code editor has a toolbar at the top and a status bar at the bottom.

```
...va DiscountCodeTblBean.java * CustomerTblBean.java *
[File] [Edit] [View] [Search] [Tools] [Help]
public java.lang.String ejbCreate(java.lang.String key) {
    if (key == null) {
        throw new javax.ejb.CreateException("The :");
    }

    // TODO add additional validation code, throw
    setKey(key);

    return null;
}

public void ejbPostCreate(java.lang.String key) {
    // TODO populate relationships here if appropriate
}

public void businessMethod() {
    //TODO implement businessMethod
}
```

3. When you are done coding the method, right-click your project's node in the Projects window and choose Build Project to trigger the compilation of the Java files and the creation of the EJB archive (JAR) file in the `dist` directory.

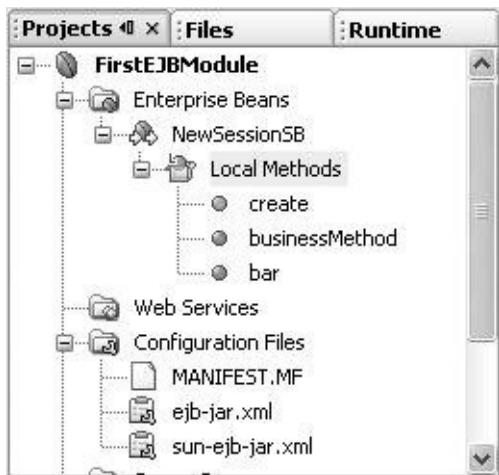
It is also possible to use the IDE's editor hints mechanism to synchronize business method implementation and definition in the interface. If you type a new business method in the Source Editor, and then select the method name in the Source Editor, the IDE displays hints. One of them is Add to Local Interface, as seen in [Figure 11-11](#).

Figure 11-11. Editor Hints for EJB Business Methods

```
public String bar() { ... }  
└─ Add To Local Interface
```

After selecting this hint, you can verify that the definition was added in the interface and that the EJB has a new Business Method (as shown in [Figure 11-12](#)).

Figure 11-12. EJB new Business Methods



Enterprise Bean Deployment Descriptors

One of the goals of NetBeans IDE is to keep you from having to deal with deployment descriptors as much as possible. This is achieved via the zero-configuration concept implemented in the IDE. When you use the provided commandssuch as Call Enterprise Bean, Use Database, Call Message, Send Email from your application, or Call Web Service Operationthe IDE performs the following tasks:

- Generates the Java code snippet for the JNDI lookup code in the caller Java file.
- Makes sure to update the Java EE deployment descriptor file by adding the corresponding `EJB-REF` or `RESOURCE-REF` elements.
- Modifies the enterprise application project to add the necessary project dependencies if the call goes to a class or resource in another IDE project. The resulting packaging has to use the Enterprise Application Project type to make sure all of the EJB modules are correctly assembled into an enterprise application Archive (EAR) file before deployment.

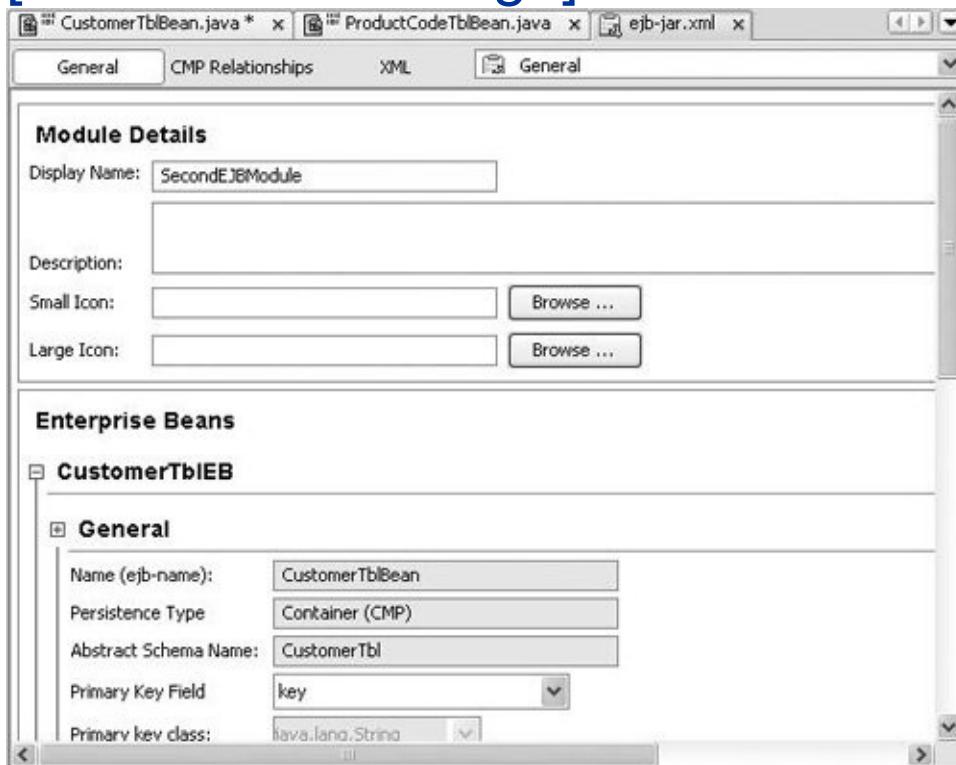
Of course, you are still allowed to edit the deployment descriptors. Here also, NetBeans IDE offers significant ease-of-use features, such as two editing modes (direct XML editing with code completion and online validation features, and visual editing).

The visual editor for a deployment descriptor file (shown in [Figure 11-13](#)) is opened by double-clicking the deployment descriptor's node in the Projects window. A set of views is available. For an EJB module, for example, General and XML views are available. Also, a combo box to the right of the

buttons for the views allows you to jump directly to a particular section in the deployment descriptor file.

Figure 11-13. Visual deployment descriptor editor

[View full size image]

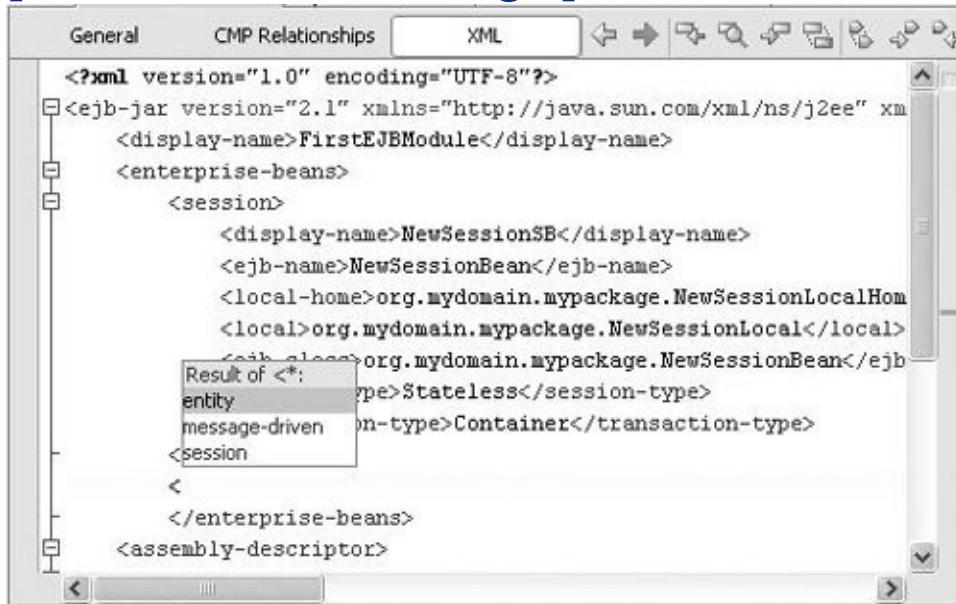


If you want to edit the file's XML code directly, you can switch to the XML view of the file by clicking the XML toggle button located at the top of the visual editor.

When you open an XML file in the Source Editor (shown in [Figure 11-14](#)), you are able to use code completion, which makes hand coding faster and more accurate. See Generating Code Snippets without Leaving the Keyboard in [Chapter 5](#) for a general discussion of how code completion works.

Figure 11-14. XML deployment descriptor editor with code completion

[View full size image]



Also note the toolbar at the top of the Source Editor. The last icon activates the XML validation action so that you can verify the conformance of the deployment descriptor file with the DTD or schema. See [Figure 11-15](#) for an example of the output after running the XML Validation command.

Figure 11-15. Output of XML validation

[View full size image]



Chapter 12. Extending Java EE Applications with Web Services

- [What Is a Web Service?](#)
- [Consuming Existing Web Services](#)
- [IDE and Server Proxy Settings](#)
- [Creating a WSDL File](#)
- [Web Service Types](#)
- [Implementing a Web Service in a Web Application](#)
- [Implementing Web Services within an EJB Module](#)
- [Testing Web Services](#)
- [Adding Message Handlers to a Web Service](#)

IN THE FOLLOWING SECTIONS, you will learn how easy it is to create and add web services to Java enterprise applications (both web applications and EJB modules) and how to publish them so that they can be used by other applications and tested from within the IDE.

What Is a Web Service?

The W3C organization defines web services as follows: "A Web service is a software system identified by a URI whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols."

The implementation for a web service can be done in any language, and a web service can be accessed by many different platforms, because the messages are XML-based. The J2EE 1.4 platform has a specification (JSR 109) for web services creation for web applications (Java EE web tier-based) and EJB Modules.

NetBeans IDE supports the creation of such web services in Java enterprise applications, as well as the consumption of published web services within Java enterprise applications and Java SE applications.

Web services allow applications to expose business operations to other applications, regardless of their implementation. This is possible via the use of the following standards:

- **XML, the common markup language for communication.** Service providers, which make services available, and service requestors, which use services, communicate via XML messages.
- **SOAP, the common message format for exchanging information.** These XML messages follow a well-defined format. Simple Object Access Protocol (SOAP) provides a common message format for web services.
- **WSDL, the common service specification format.** In addition to common message format and markup language, there must be a common format that all service providers can use to specify service details, such as the service type, the service parameters, and how to access the service. Web Services Description Language (WSDL) provides web services with such a common specification format.

Consuming Existing Web Services

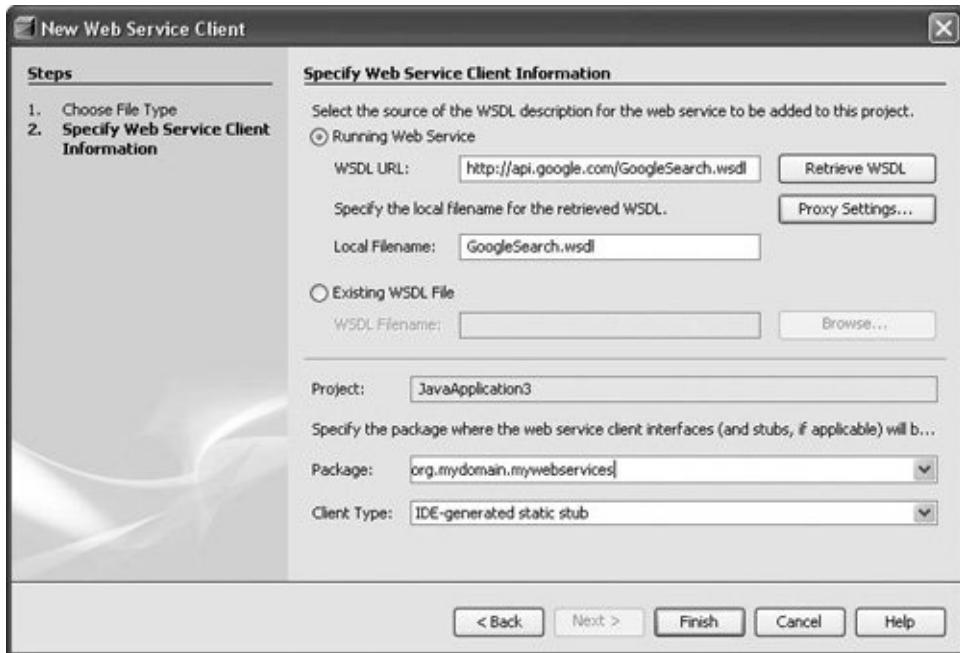
For a web service to be usable by other applications, a WSDL file must be published for the web service. The application uses this WSDL file to construct the necessary artifacts. The NetBeans Web Service Client wizard automates the creation process and updates the deployment descriptor files with the appropriate `<service-ref>` elements for the web application project. You can also consume web services from general Java projects.

To create a WSDL file:

1. Open the Web Service Client wizard by right-clicking the node of a web application project or a Java project and choose New | Web Service Client.
2. In the wizard (shown in [Figure 12-1](#)), pick a WSDL file from the URL of the running service or from a local directory on your system.

Figure 12-1. New File wizard: create web service client

[\[View full size image\]](#)



Make sure you pick a package name for the generated interfaces that your user code will use to access and interact with this web service. Then choose a client type from the Client Type combo box. Two types of web service clients can be generated:

- **J2EE Container-generated static stub (JSR-109 Enterprise web services).** Defines the packaging of web services into standard Java EE modules, including a new deployment descriptor, and defines web services that are implemented as session beans or servlets. This is the recommended and portable (via the J2EE 1.4 specification) way.
- **IDE-Generated static stub (JSR-101 JAX-RPC).** Defines the mapping of WSDL to Java classes and vice versa. It also defines a client API to invoke a remote web service and a runtime environment on the server to host a web service.

3. Click Finish.

In the Projects window, you should see a new logical node

under the Web Services References node (see [Figure 12-2](#)). Explore the children of this node; for each web service operation defined in the WSDL file, the IDE shows a node that has the operation's name and a popup menu that allows you to test this operation directly within the IDE without writing a single line of code.

Figure 12-2. Projects window with populated Web Service References node



The necessary Web Service References entry in the `web.xml` file for this web application is updated, so you can use this web service from either a servlet or a utility Java class within your web application. The entry in the `web.xml` file should look something like the following code block:

```
<service-ref>
  <service-ref-name>service/GoogleSearchService</serv
```

```
<service-interface>org.mydomain.mywebservices.GoogleSearchPort</service-interface>
<wsdl-file>WEB-INF/wsdl/GoogleSearch.wsdl</wsdl-file>
<jaxrpc-mapping-file>WEB-INF/wsdl/GoogleSearch-mapping.xml</jaxrpc-mapping-file>
<port-component-ref>
  <service-endpoint-interface>
    org.mydomain.mywebservices.GoogleSearchPort</service-endpoint-interface>
  </port-component-ref>
</service-ref>
```



Publishing of the WSDL file is often done via a Universal Discovery, Description, and Integration (UDDI) registry. NetBeans IDE does not provide a user interface for publishing web services to a UDDI registry. However, this does not prevent you from using WSDL files that come from a UDDI registry or other source. Although most of the time, the IDE selects the correct `wscompile` tool options for generating client code from the WSDL file, you can refine these options.

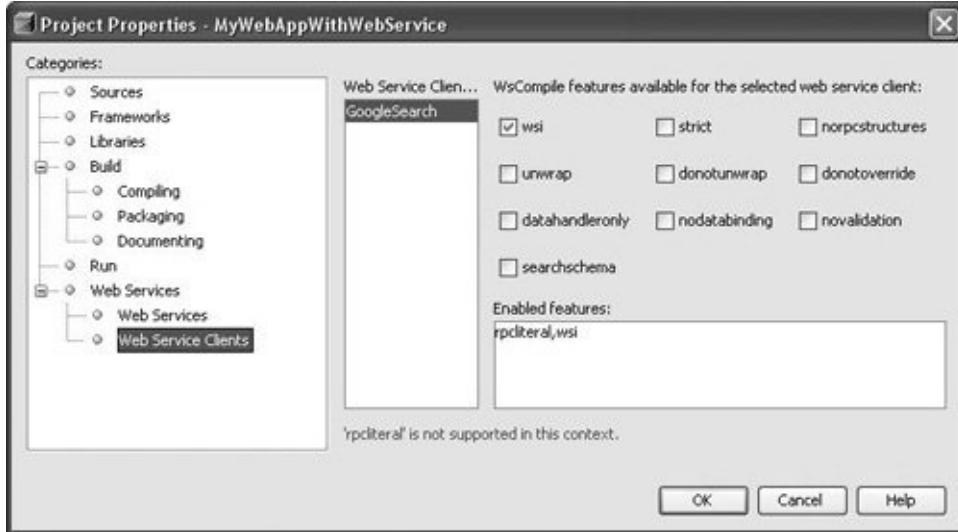
To modify the `wscompile` tool options for a project (this is for advanced developers and is not a common task):

1. Right-click the project's main node and choose Properties to open the Project Properties dialog box.
2. Select the Web Services | Web Services Clients node (see [Figure 12-3](#)).

Figure 12-3. Project Properties dialog box with

Web Service Clients panel displayed

[View full size image]



For example, you can use checkboxes to set or disable the `strict`, `unwrap` and `novalidation` flags and to enter which `wscompile` options to use.

For now, keep the default values used by the IDE.

For advanced configuration, see the key to the `wscompile` tool options in [Table 12-1](#).

Table 12-1. `wscompile` Tool Options

Option	Effect on Web Service Client
<code>datahandleronly</code>	Maps attachments to the <code>DataHandler</code> type.
<code>donotoverride</code>	No regeneration of classes that already exist on the classpath.
<code>donounwrap</code>	Disables unwrapping of document/literal wrapper elements in WSI mode(default).

explicitcontext	Turns on explicit service context mapping.
jaxbenumtype	Maps anonymous enumeration to its base type.
nodatabinding	Turns off data binding for literal encoding.
noencodedtypes	Turns off encoding type information.
nomultirefs	Turns off support for multiple references.
norpcstructures	No generation of RPC structures (-import only).
novalidation	Turns off full validation of imported WSDL documents.
resolveidref	Resolve xsd:IDREF.
searchschema	Searches schema aggressively for types.
serializeinterfaces	Turns on direct serialization of interface types.
strict	Generates code strictly compliant with JAXRPC Specification.
unwrap	Enables unwrapping of document/literal wrapper elements in WSI mode.
wsi	Checks for compliance with the WSI-Basic Profile, which is a specification for improved interoperability. For example, the WS-I Basic Profile prohibits the use of rpc/encoded. Therefore, if you set the wsi feature, a warning will be generated when you build a

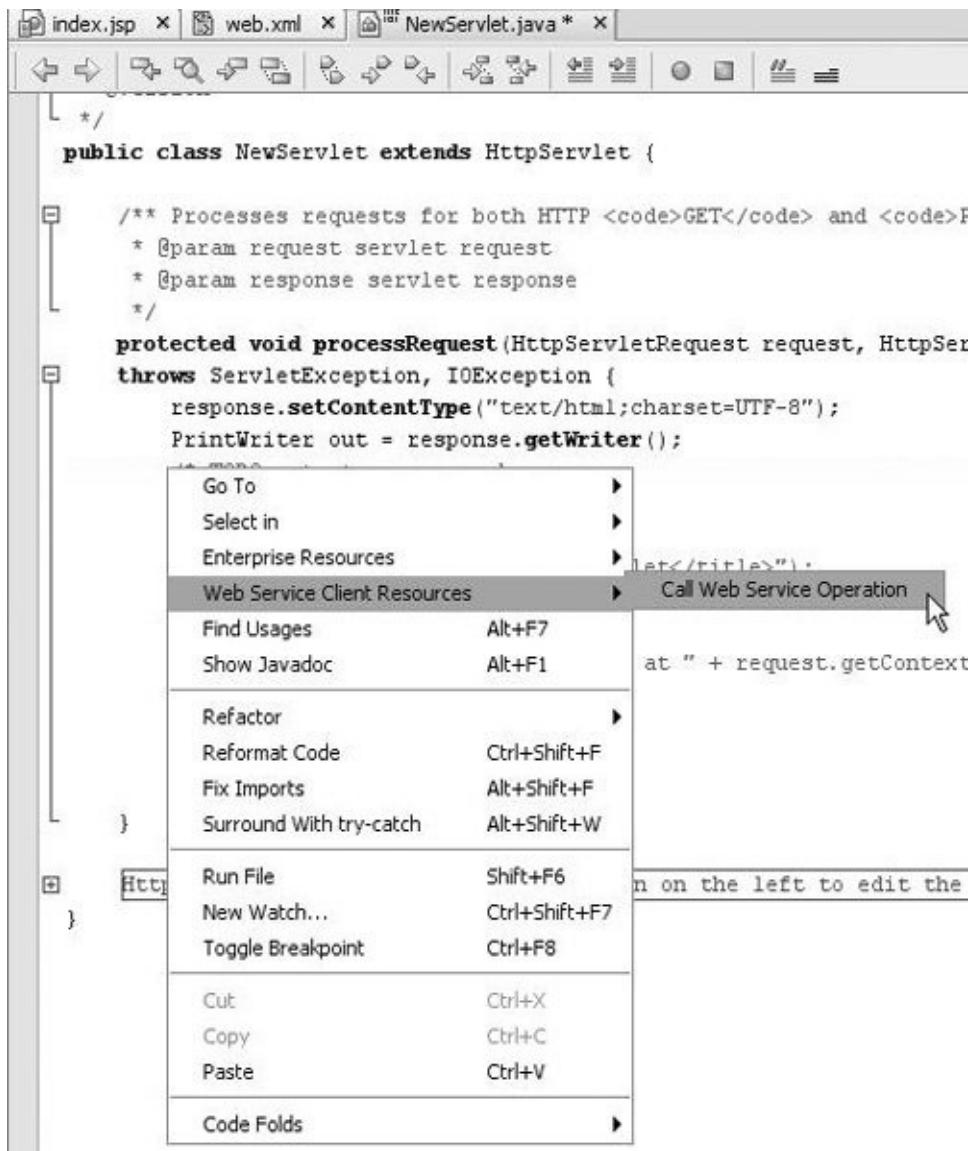
web service client that uses a WSDL file that uses rpc/encoded.

Now you will want to call an operation for this web service within your Java code or even in your JSP page. To call an operation:

1. Right-click the location in your Java source code or your JSP page where you want to insert some code, and choose Web Service Client Resources | Call Web Service Operation (see [Figure 12-4](#)).

Figure 12-4. Generating code in the Source Editor for calling a web service operation

[\[View full size image\]](#)



2. Select an operation from the Select Operation to Invoke dialog box.

After calling the operation, you can see the new code that the IDE has added within your Java file.

You should see a try/catch block that looks something like the following:

```
try {  
    getGoogleSearchPort().doSpellingSuggestion  
    //TODO enter operation arguments */;
```

```
    } catch(java.rmi.RemoteException ex) {
        // TODO handle remote exception
    }
```

Make sure you replace the `//TODO` comment with the list of correct parameters for this operation. In our example, the `doSpellingSuggestion` method takes two arguments. We also want to display the output parameter. So, the updated code looks like the code below:

```
try {
    getGoogleSearchPort().doSpellingSuggestion(
        out.println(
            getGoogleSearchPort().doSpellingSuggest(
                "cx5Str9QFHJu96fb3CY920ie3aAiAVr6
                "NetBeanss") //the term I need a
            );
} catch(java.rmi.RemoteException ex) {
    out.println(ex.toString());
}
```

("`cx5Str9QFHJu96fb3CY920ie3aAiAVr6`" is a private key to access the Google search via APIs. To get yours, go to <http://www.google.com/apis/> and register).

You should also see some generated private methods that do the web service reference lookup from the initial context and the RPC Port accessor for the selected operation, as follows:

```
private org.mydomain.mywebservices.GoogleSearchService
getGoogleSearchService() {
    org.mydomain.mywebservices.GoogleSearchService
    googleSearchService = null;
```

```

try {
    javax.naming.InitialContext ic = new
        javax.naming.InitialContext();
    googleSearchService =
        (org.mydomain.mywebservices.GoogleSearch
        ic.lookup("java:comp/env/service/GoogleS
} catch(javax.naming.NamingException ex) {
    // TODO handle JNDI naming exception
}
return googleSearchService;
}

private org.mydomain.mywebservices.GoogleSearchPor
getGoogleSearchPort() {
    org.mydomain.mywebservices.GoogleSearchPort
        googleSearchPort = null;
    try {
        googleSearchPort =
            getGoogleSearchService().getGoogleSearchPort()
    } catch(javax.xml.rpc.ServiceException ex) {
        // TODO handle service exception
    }
    return googleSearchPort;
}

```



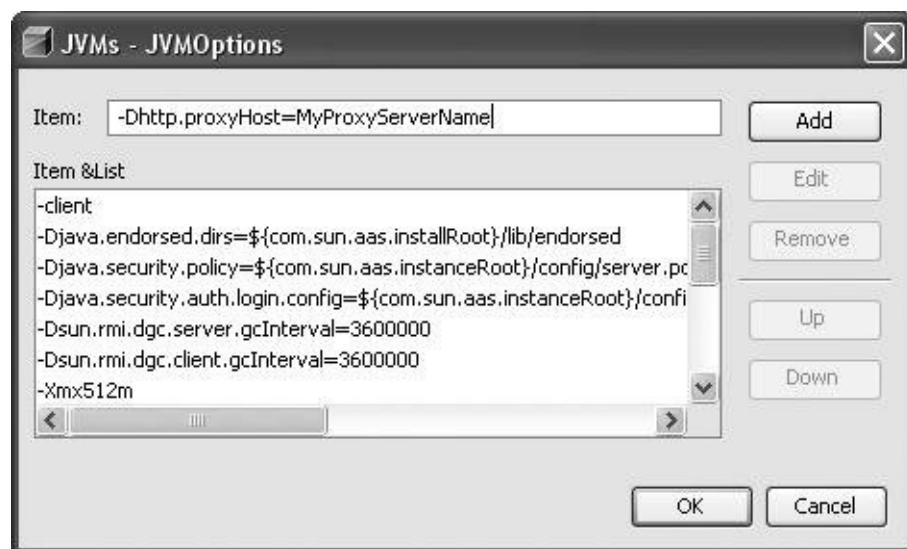
Notice the `// TODO` statements in the added code. The IDE's Java EE wizards always add some TODO statements so that you can quickly find in the Java source code the areas you need to fill in, such as business logic that you need to complete. You can view all outstanding TODO statements in a single list by choosing Window | To Do.

IDE and Server Proxy Settings

It is possible that getting information for a web service does not work for you. In this case, you should check if your network access requires a proxy and you'll need to specify a proxy host and proxy port by first clicking the Proxy Settings button. Then the IDE will be able to look up the WSDL.

Do not forget to set up the Sun Java System Application Server runtime to also access the Internet via this proxy server. Using the IDE's Runtime window, navigate to the JVMs node of a running Sun Java System Application Server instance. When you double-click this node, the JVMsJVMOptions dialog appears (see [Figure 12-5](#)), where you can add the necessary JVMOptions properties: `http.proxyHost` and `http.proxyPort`. The server will have to restart for the new JVM options to be applied.

Figure 12-5. Setting the proxy information for a Sun Application Server



Creating a WSDL File

To create a new WSDL file with embedded or imported XML schema:

1. Right-click a web application or Java application, choose New | File/Folder, and select WSDL file from the Web Services category.
2. In the Name and Location page, set the following properties:
 - **File Name.** Specifies the WSDL filename.
 - **Folder.** Specifies a folder to house the WSDL file.
 - **Target Namespace.** Specifies the namespace of the elements (such as message, portType, binding) defined in the WSDL file.
 - **Import XML Schema File(s).** Specifies the XML schemas needed by the WSDL file, if any.

The IDE creates a WSDL file for you, with default code that you can modify according to your business needs.

Implementing a Web Service in a Web Application

In the IDE, you can create a web service by implementing an existing WSDL file, exposing existing code, or creating one from scratch. The IDE generates deployment information in the deployment descriptors, the necessary Java code that describes this web service (the default implementation; the Service Endpoint Interface, also called *SEI*; and a compilation target in the Ant build script).

A simple way to create a web service is to start from scratch.

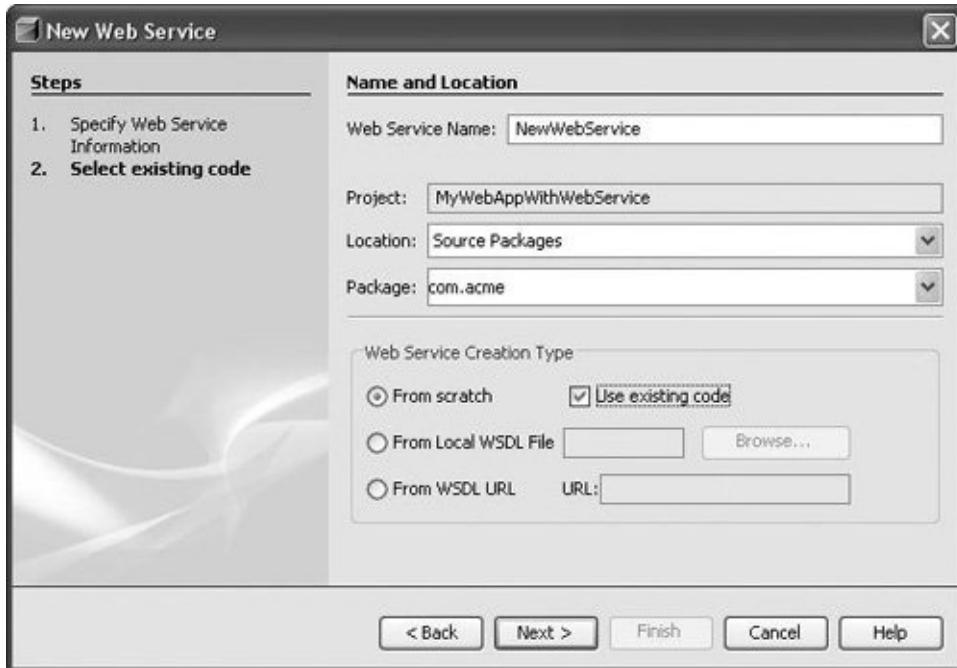
Creating a Web Service

To create a new web service:

1. Right-click a web application's project node and choose New | Web Service.
2. In the New Web Service wizard (see [Figure 12-6](#)), specify a name and a Java package, select the From Scratch radio button, and click Finish.

Figure 12-6. New Web Service wizard

[\[View full size image\]](#)



It is strongly recommended that you do *not* use the default package (which would occur if you left the Package field blank).

When you complete the wizard, a new web service with no operations is added to your project. You can see a logical node representing this new component in the Web Services node in the Projects window. A new servlet entry is added to the `web.xml` file, and a `webservice-description` entry is added in the `webservices.xml` description file (a sibling of the `web.xml` file). Most of the time, you don't need to worry about these entries, as they are automatically updated by the IDE when necessary.

As a developer, you will use the Projects window most (as opposed to the Files window). It synthesizes the implementation details of a web service (such as its implementation class and its SEI) and allows you to:

- Explore the existing operations.
- Add new operations for the web service.

- Register the web service to the runtime registry that will allow you to test it from within the IDE.
- Configure any web service message handler classes that might be associated with it.

Adding an Operation

Choose Add Operation (either by right-clicking the web service's node in the Projects window, as shown in [Figure 12-7](#), or by right-clicking within the web service in the Source Editor). In the Add Operation dialog box (shown in [Figure 12-8](#)), you can configure the list of parameters for this operation, as well as the return type and any exceptions.

Figure 12-7. Adding an operation to a web service

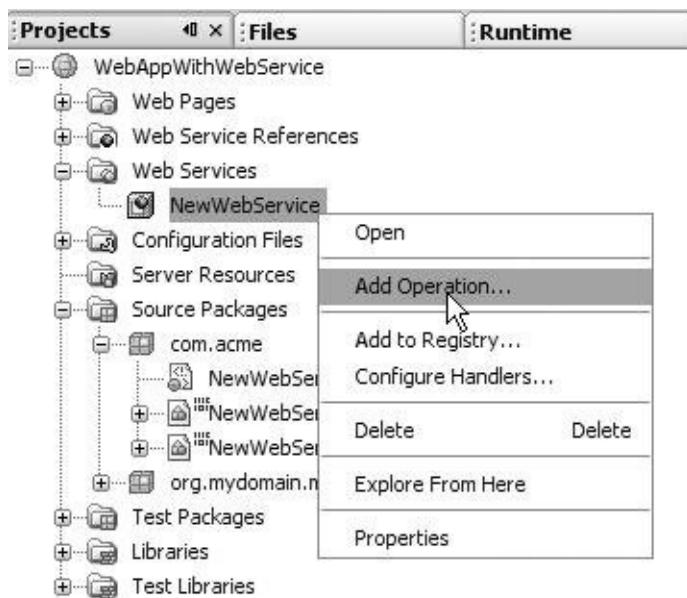


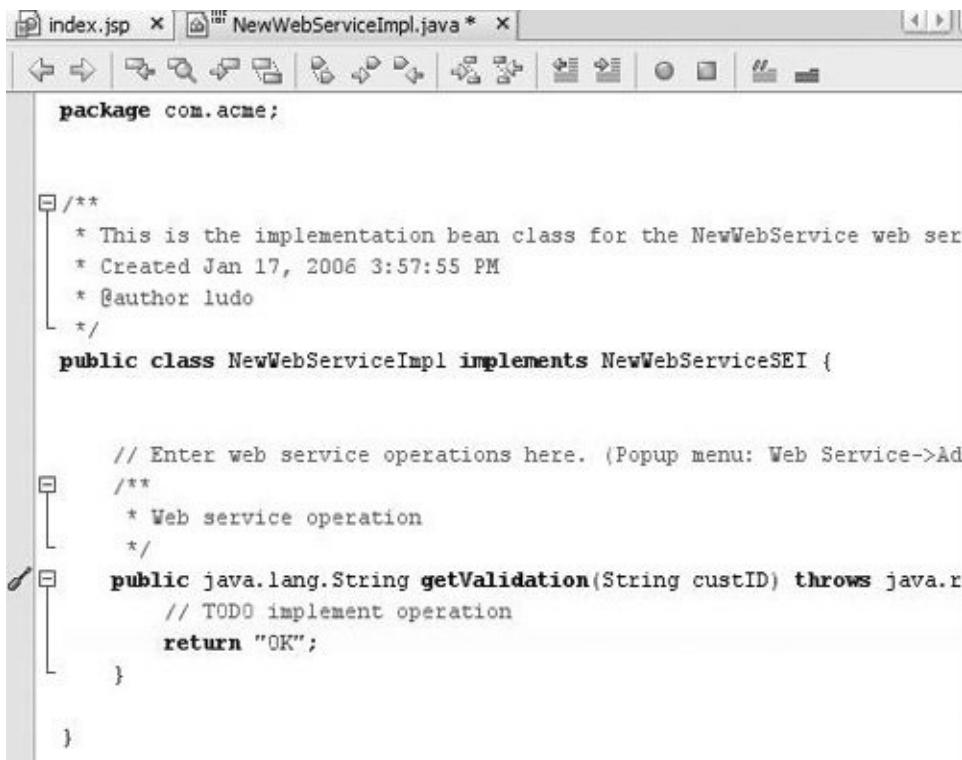
Figure 12-8. Add Operation dialog box



The operation is added to the Java class implementing the web service (see [Figure 12-9](#)). The IDE automatically synchronizes the SEI, so you need to concentrate only on developing the operation method body.

Figure 12-9. Adding an operation to a web service

[\[View full size image\]](#)



The screenshot shows a Java code editor window with two tabs: "index.jsp" and "NewWebServiceImpl.java *". The "NewWebServiceImpl.java" tab is active. The code is as follows:

```
package com.acme;

/**
 * This is the implementation bean class for the NewWebService web service.
 * Created Jan 17, 2006 3:57:55 PM
 * @author ludo
 */
public class NewWebServiceImpl implements NewWebServiceSEI {

    // Enter web service operations here. (Popup menu: Web Service->Add)
    /**
     * Web service operation
     */
    public java.lang.String getValidation(String custID) throws java.rmi.RemoteException {
        // TODO implement operation
        return "OK";
    }
}
```

Compiling the Web Service

Now that the web service has been added to your web application project, you just need to call the Build Project or Deploy Project command (available by right-clicking the project's node) to trigger an Ant build process that will invoke the `wscompile` tool with the correct parameters. The web application will be packaged as a WAR file and deployed to the target server for this project. Following is an example of what the Ant output might look like (keeping in mind that by default the lines do not wrap they way they do in this excerpt):

```
init:  
deps-module-jar:  
deps-ear-jar:  
deps-jar:
```

```
wscompile-init:  
Created dir: C:\ludo\WebAppWithWebService\build\web\WEB-INF\classes  
Created dir: C:\ludo\WebAppWithWebService\build\web\WEB-INF\lib  
Created dir: C:\ludo\WebAppWithWebService\build\generated  
Created dir: C:\ludo\WebAppWithWebService\build\generated\wsclient  
Created dir: C:\ludo\WebAppWithWebService\build\generated\wsclient\src  
GoogleSearch-client-wscompile:  
Copying 1 file to  
C:\ludo\WebAppWithWebService\build\generated\wsclient\src  
warning: Processing WS-I non conforming operation "doGet" with  
RPC-Style and SOAP-encoded  
warning: Processing WS-I non conforming operation "doSearch" with  
RPC-Style and SOAP-encoded  
warning: Processing WS-I non conforming operation "doGet" with  
RPC-Style and SOAP-encoded  
web-service-client-generate:  
Copying 1 file to C:\ludo\WebAppWithWebService\build\web\WEB-INF\classes  
Created dir: C:\ludo\WebAppWithWebService\build\web\WEB-INF\lib  
Copying 1 file to C:\ludo\WebAppWithWebService\build\web\WEB-INF\lib  
Copying 5 files to C:\ludo\WebAppWithWebService\build\web\WEB-INF\lib  
Copying 2 files to C:\ludo\WebAppWithWebService\build\web\WEB-INF\lib  
library-inclusion-in-archive:  
library-inclusion-in-manifest:  
web-service-client-compile:  
Compiling 12 source files to C:\ludo\WebAppWithWebService\build\web\WEB-INF\classes  
Compiling 3 source files to C:\ludo\WebAppWithWebService\build\web\WEB-INF\classes  
Copying 1 file to C:\ludo\WebAppWithWebService\build\web\WEB-INF\classes  
NewWebService_wscompile:  
command line: wscompile "C:\Program Files\Java\jdk1.6.0\jre\bin\java" -cp "C:\Program  
Files\Java\jdk1.6.0\lib\tools.jar;C:\Sun\AppServer82\lib\appserver.jar;  
\AppServer82\lib\saaj-api.jar;C:\Sun\AppServer82\lib\saaj-impl.jar;  
C:\Sun\AppServer82\lib\jaxrpc-api.jar;C:\Sun\AppServer82\lib\jaxrpc-  
impl.jar;C:\ludo\WebAppWithWebService\build\web\WEB-INF\classes"  
wscompile "C:\Program Files\Java\jdk1.6.0\jre\bin\java" -cp "C:\Program  
Files\Java\jdk1.6.0\lib\tools.jar;C:\Sun\AppServer82\lib\appserver.jar;  
\AppServer82\lib\saaj-api.jar;C:\Sun\AppServer82\lib\saaj-impl.jar;  
C:\Sun\AppServer82\lib\jaxrpc-api.jar;C:\Sun\AppServer82\lib\jaxrpc-  
impl.jar;C:\ludo\WebAppWithWebService\build\web\WEB-INF\classes"
```

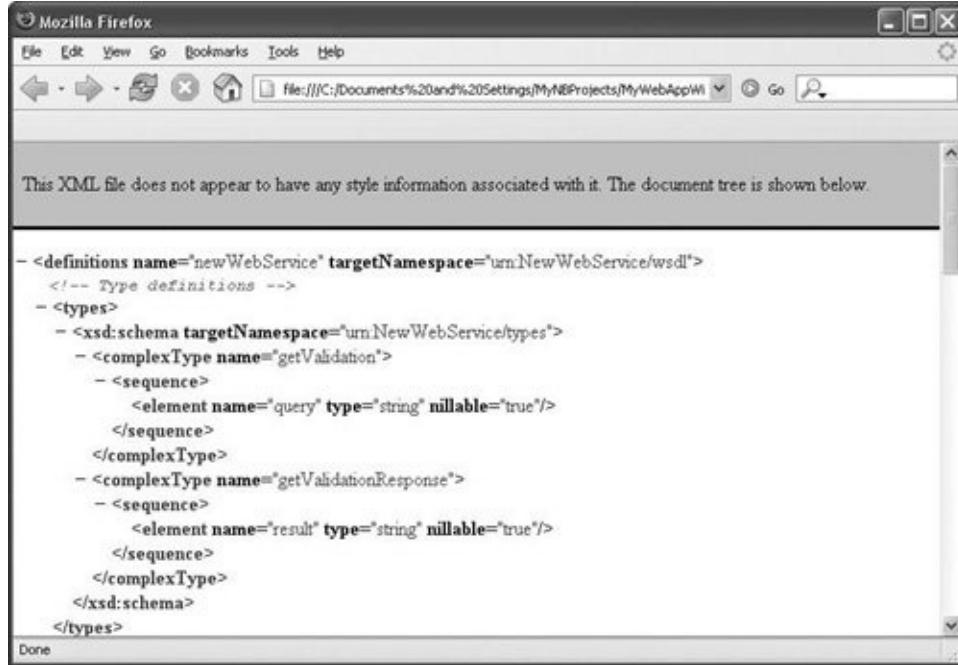
```
com.sun.xml.rpc.tools.wscompile.Main -d
"C:\ludo\WebAppWithWebService\build\generated\wsbinary
features:documentliteral,strict,useonewayoperations -ke
"C:\ludo\WebAppWithWebService\build\web\WEB-INF\NewWeb
mapping.xml" -nd "C:\ludo\WebAppWithWebService\build\w
"C:\ludo\WebAppWithWebService\build\generated\wsservic
Xprintstacktrace
"C:\ludo\WebAppWithWebService\src\java\com\acme\NewWeb
config.xml"
[creating model: NewWebService]
[creating service: NewWebService]
[creating port: com.acme.NewWebServiceSEI]
[creating operation: getValidation]
compile:
compile-jsps:
Created dir: C:\ludo\WebAppWithWebService\dist
Building jar: C:\ludo\WebAppWithWebService\dist\WebApp
do-dist:
dist:
run-deploy:
Incrementally deploying WebAppWithWebService_localhost
Completed incremental distribution of WebAppWithWebSer
Trying to create reference for application in target server
successfully
Trying to start application in target server completed
Deployment of application WebAppWithWebService completed
run-display-browser:
Browsing: http://localhost:8080/WebAppWithWebService/
run:
BUILD SUCCESSFUL (total time: 10 seconds)
```

You can use a web browser (as shown in [Figure 12-10](#)) to query the deployed and running web application for the published WSDL file for this web service. In this case, the file is

<http://localhost:8080/WebAppWithWebService/NewWebService?WSDL>.

Figure 12-10. Web browser displaying the WSDL file for the web service

[\[View full size image\]](#)



The screenshot shows a Mozilla Firefox browser window displaying an XML document. The title bar says "Mozilla Firefox". The address bar shows "file:///C:/Documents%20and%20Settings/MyNBProjects/MyWebApp/WSDL.wsdl". The main content area contains the following text:

```
This XML file does not appear to have any style information associated with it. The document tree is shown below.

<definitions name="newWebService" targetNamespace="urn:NewWebService/wsdl">
  <!-- Type definitions -->
  <types>
    <xsd:schema targetNamespace="urn:NewWebService/types">
      <complexType name="getValidation">
        <sequence>
          <element name="query" type="string" nillable="true"/>
        </sequence>
      </complexType>
      <complexType name="getValidationResponse">
        <sequence>
          <element name="result" type="string" nillable="true"/>
        </sequence>
      </complexType>
    </xsd:schema>
  </types>
</definitions>
```

Your web service is now published and available to anyone so that other applications (Java enterprise applications, .NET applications, or Java ME applications) can interoperate with its operations.

Creating Web Services from WSDL

You can also create a web service from a WSDL document. A typical scenario when this is necessary is when business partners formulate the way they will communicate in web

services. The "contract" between them would be the WSDL, in which they would agree on the data and messages that will be exchanged, as well as on how these messages will be sent and received. Then this WSDL is used to implement the web service.

The elements of a WSDL document can be categorized into *abstract* and *concrete* parts. The `types`, `message`, and `portType` elements describe the data that form the messages sent and received by web services and clients, as well as the operations that will use these messages. These sections constitute the abstract portion of the WSDL. The `binding` and `service` elements describe the protocols and transport mechanisms used to send and receive the messages, as well as the actual address of the endpoint. This is considered to be the concrete portion of the WSDL file.

When you are creating a web service from a WSDL in the IDE, a new WSDL is created and packaged with the web service. The abstract portion of the original WSDL is copied to the new one. The concrete portion of the original WSDL is normalized for SOAP binding. Because the JAX-RPC runtime that is used in the NetBeans IDE only supports SOAP over HTTP binding, the WSDL is searched for the first occurrence of this binding. If found, it is copied into the new WSDL. If not, a SOAP/HTTP binding is created for the first `portType` defined in the original WSDL. Thus, the web service created from WSDL in the NetBeans IDE will always have exactly one SOAP binding and one service port corresponding to that binding. The service element that is added will be named according to the web service name specified in the wizard, replacing the service element in the original WSDL.

To create a web service from a WSDL file, click the From Local WSDL File radio button or the From WSDL URL radio button in the wizard, depending on the source of the WSDL document. When the web service is created, classes for the service endpoint interface and implementation bean will be created. These classes will contain all the operations described in the

WSDL. The implementation bean class will be displayed in the Source Editor, and you may then enter code to implement these operations. If the WSDL file describes operations that use complex types, classes for these types (known as *value types*) are also generated so that you may use them in your implementation code.

Because the WSDL document governs the interface to the web service, you may not add new operations to web services that are created from a WSDL, because these operations will not be reflected back in the WSDL.

Note that WSDLs that import other WSDLs are not supported by this facility.

Web Service Types

By default, NetBeans IDE creates *document/literal* Web services. This refers to the way the SOAP message is sent over the wire and is expressed in the SOAP binding part of the WSDL. The document/literal nomenclature comes from the way SOAP messages are described in the SOAP binding of a WSDL documentnamely, its `style` and `use` attributes.

The `style` attribute refers to the formatting of the SOAP message. This basically refers to what the SOAP body will contain when it is sent over the wire.

There are two ways to format a SOAP message: RPC and document. When the style is RPC, the contents of the SOAP body are dictated by the rules of the SOAP specificationthat is, the first child element is named after the operation, and its children are interpreted as the parameters of the method call. The endpoint will interpret this as an XML representation of a method callthat is, a remote procedure call. On the other hand, if the style attribute is `document`, the SOAP body consists of arbitrary XML, not constrained by any rules and able to contain whatever is agreed upon by the sender and receiver.

The `use` attribute describes how data is converted between XML and software objectsthat is, how it is serialized to XML.

If the `use` attribute is `encoded`, the rules to encode/decode the data are dictated by some rules for encoding, the most common of which is the SOAP encoding specified in the SOAP specification. Section 5 of the SOAP specification defines how data should be serialized to XML. In this case, web services or clients see data in terms of objects.

If the `use` attribute is `literal`, the rules for encoding the data are dictated by an XML schema. There are no encoding rules, and the web service or client sees the data in terms of XML. Here, the developer does the work of parsing the XML to search for needed data.

Thus, document/literal web services typically are used to exchange business documents, whereas RPC/encoded web services typically are used to invoke remote objects. For this reason, document/literal is preferred over RPC/encoded because in document/literal, you have full control of the messages that are being exchanged. The WS-I Basic Profile, which is a specification for web services interoperability, does not support RPC/encoded web services.

Following are the advantages of document/literal over RPC/encoded web services:

- Document/literal formatting is more interoperable than RPC/encoded formatting because RPC formatting tends to bind the messages to programming language structures.
- Document/literal Web services scale better than RPC/encoded because of

the overhead involved in marshalling and unmarshalling RPC data.

- Document-centric Web services lend themselves to validation of the documents being exchanged. This is cumbersome to do with RPC-style services.

Implementing Web Services within an EJB Module

To implement a web service in an EJB Module, you will use a similar web service wizard described for the web application. Most of the artifacts that are created and the deployment descriptor entries that are added for the module are similar to those in a web application. You will follow the same procedure for adding SOAP message handlers and configuring them in web services.

One significant difference is the implementation bean of a web service in an EJB module. JSR 109 requires that web services be implemented in the EJB module as stateless session beans. Thus, the implementation of the web service operations in an EJB module is contained in the session bean class. The module's deployment descriptor will have a stateless session bean entry, but this will declare an endpoint interface instead of a local or remote interface. Also, a web service in an EJB module does not have a home interface.

Once you have created a web service within an EJB module, you will see the Web service logical node in the Projects window (shown in [Figure 12-11](#)). The source code you will manipulate is the stateless session bean implementation class. The developer experience is similar to the development of a web service with a web application.

Figure 12-11. Web service within an EJB Module

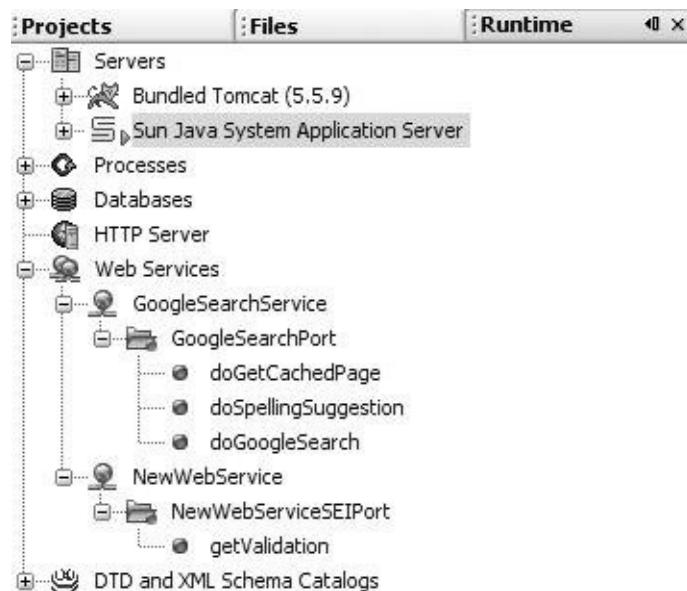


In the Source Packages node, you can see the service bean class and the SEI and the service XML config file.

Testing Web Services

NetBeans IDE has a built-in test environment for publishing web services, either for those created by you and deployed within a web application or an enterprise application, or those published externally. All you need is access to the WSDL file for this web service. You can use the Web Services Registry tool from the Web Services node in the IDE's Runtime window (shown in [Figure 12-12](#)) to register the web services in the IDE.

Figure 12-12. Web Services registry in the Runtime window



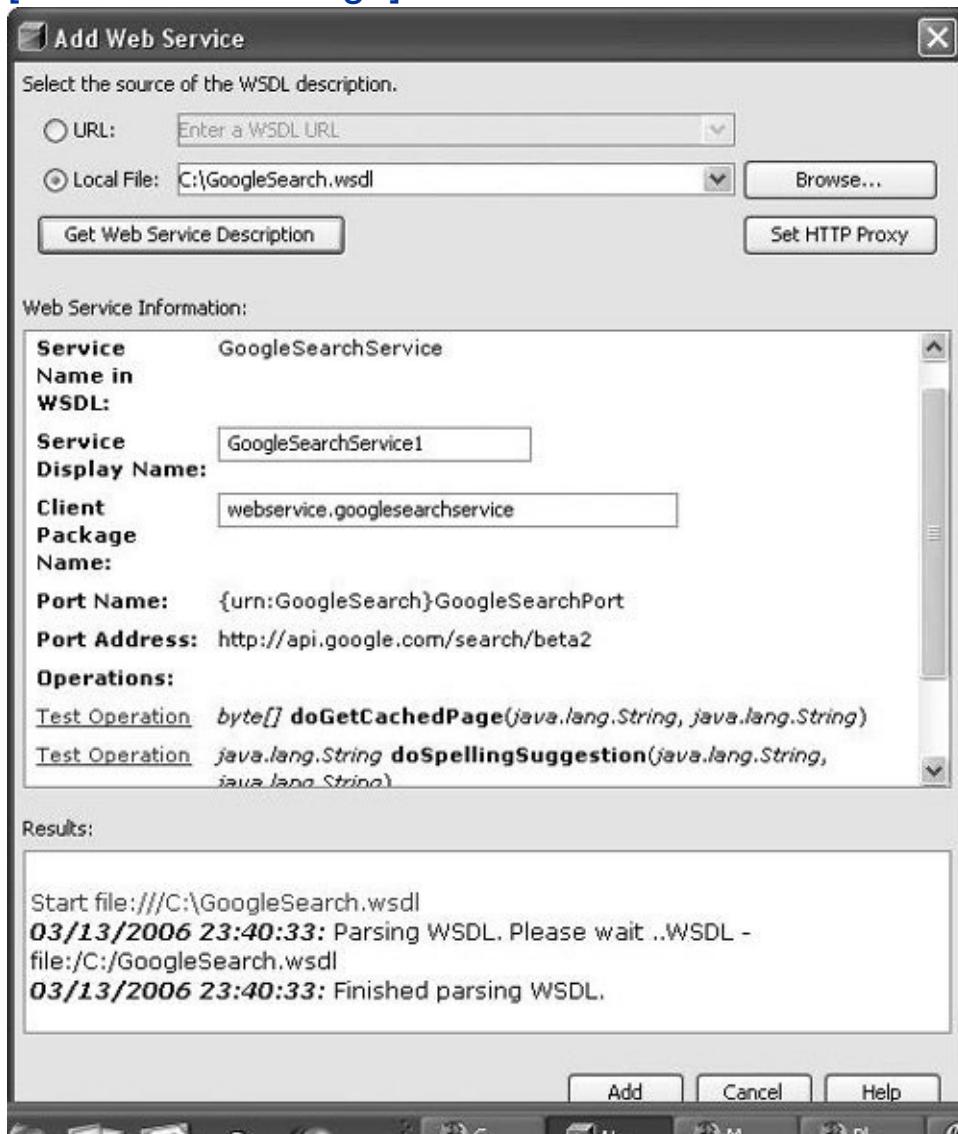
To add a web service to the registry:

1. Right-click the Web Services node in the Runtime window and choose Add Web Service to activate the wizard.

2. In the wizard (see [Figure 12-13](#)), enter the WSDL file (either as a URL or local file) and click the Add button.

Figure 12-13. Add Web Service wizard

[[View full size image](#)]



Once you specify the file and click the Add button, the service's operations are available as nodes in the registry, and you can use the Test Operation command.

To test the Web service operation:

1. In the Runtime window, expand the Web Services node and navigate to the node for the operation you want to test (this node should have no subnodes); right-click that node and choose Test Operation (see [Figure 12-14](#)).

Figure 12-14. Choosing the Test Operation command on a web service in the Runtime window



2. In the wizard that appears (see [Figure 12-15](#)), enter any input parameters for this operation and click Submit.

Figure 12-15. The Test Web Service Operation dialog box



The web service operation is called, and the output parameters are displayed in the Results area.

Adding Message Handlers to a Web Service

NetBeans IDE makes it easy to develop Java EE web services and clients because it shields application developers from the underlying SOAP messages. Instead of writing code to build and parse SOAP messages, you merely implement the service methods and invoke them from remote clients.

However, there might be times when you want to add functionality to web service applications without having to change the web service or client code. For example, you might want to encrypt remote calls at the SOAP message level. SOAP message handlers provide the mechanism for adding this functionality without having to change the business logic. Handlers accomplish this by intercepting the SOAP message as it makes its way between the client and service.

A SOAP message handler is a stateless instance that accesses SOAP messages representing RPC requests, responses, or faults. Tied to service endpoints, handlers enable you to process SOAP messages and to extend the functionality of the service. For a given service endpoint, one or more handlers may reside on the server and client.

A SOAP request is handled as follows:

- The client handler is invoked before the SOAP request is sent to the server.
- The service handler is invoked before the SOAP request is dispatched to the service endpoint.

A SOAP response is processed in this order:

1. The service handler is invoked before the SOAP response is

sent back to the client.

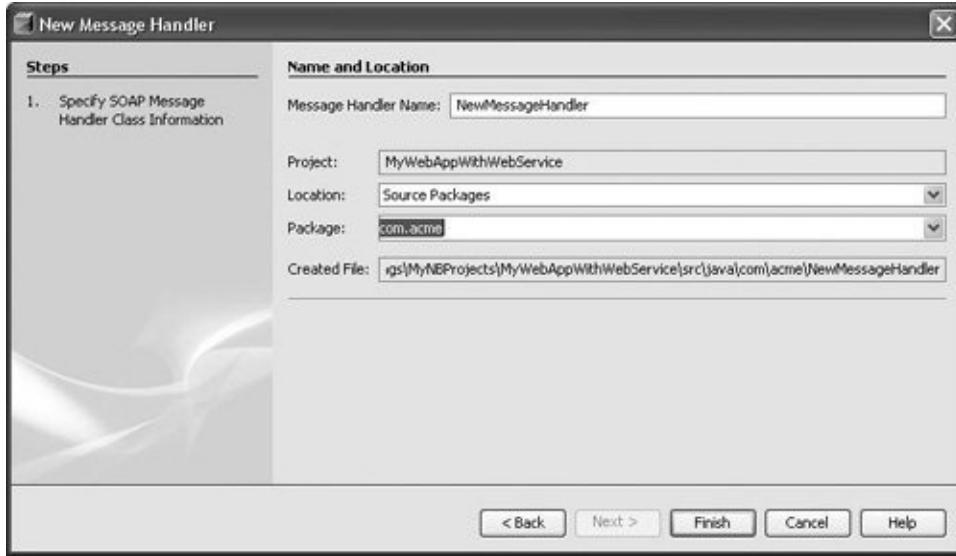
2. The client handler is invoked before the SOAP response is transformed into a Java method return and passed back to the client program.

To create a message handler in a web application in the IDE:

1. Right-click the web application's node in the Projects window and choose New | Message Handler.
2. On the New Message Handler page of the New File wizard (shown in [Figure 12-16](#)), enter a name and package for the handler and click Finish.

Figure 12-16. New Message Handler page of the New File wizard

[[View full size image](#)]



The new Java file created contains the core code for the

message handler, as shown in the code sample below.

One interesting method here is `handleRequest(MessageContext context)`, which is called before the SOAP message is dispatched to the endpoint. The generated handler class provides a default implementation of this method (as an example), which prints out the contents of the SOAP body plus some date information. Note that the `MessageContext` parameter provides a context for obtaining the transmitted SOAP message. You may then use the SAAJ API (SOAP with Attachments API for Java) to access and manipulate the SOAP message.

Another method, `handleResponse(MessageContext context)`, is called before the response message is sent back to the caller. This method, together with `handleFault`, provides only the default implementation; it is left to you to provide your own implementation.

```
package com.acme;

import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPBody;
import java.util.Date;

public class NewMessageHandler extends
    javax.xml.rpc.handler.GenericHandler {
    // TODO Change and enhance the handle methods to s
    needs.

    private QName[] headers;
```

```
public void init(HandlerInfo config) {
    headers = config.getHeaders();
}

public javax.xml.namespace.QName[] getHeaders() {
    return headers;
}

// Currently prints out the contents of the SOAP body
// information.
public boolean handleRequest(MessageContext context)
    try{
        SOAPMessageContext smc = (SOAPMessageContext)context;
        SOAPMessage msg = smc.getMessage();
        SOAPPart sp = msg.getSOAPPart();
        SOAPEnvelope se = sp.getEnvelope();
        SOAPHeader shd = se.getHeader();

        SOAPBody sb = se.getBody();
        java.util.Iterator childElems = sb.getChildElements();
        SOAPElement child;
        StringBuffer message = new StringBuffer();
        while (childElems.hasNext()) {
            child = (SOAPElement) childElems.next();
            message.append(new Date().toString() +
                           formLogMessage(child, message));
        }

        System.out.println("Log message: " + message);
    } catch(Exception e){
        e.printStackTrace();
    }
    return true;
}

public boolean handleResponse(MessageContext context)
    try{
        SOAPMessageContext smc = (SOAPMessageContext)context;
        SOAPMessage msg = smc.getMessage();
        SOAPPart sp = msg.getSOAPPart();
        SOAPEnvelope se = sp.getEnvelope();
        SOAPHeader shd = se.getHeader();

        SOAPBody sb = se.getBody();
        java.util.Iterator childElems = sb.getChildElements();
        SOAPElement child;
        StringBuffer message = new StringBuffer();
        while (childElems.hasNext()) {
            child = (SOAPElement) childElems.next();
            message.append(new Date().toString() +
                           formLogMessage(child, message));
        }

        System.out.println("Log message: " + message);
    } catch(Exception e){
        e.printStackTrace();
    }
    return true;
}
```

```

        return true;
    }

    public boolean handleFault(MessageContext context)
        return true;
    }

    public void destroy() {
    }

    private void formLogMessage(SOAPElement child, String
{
        message.append(child.getElementName().getLocalName());
        message.append(child.getValue() != null ? ":" : " ")
+ " " : " ");

        try{
            java.util.Iterator childElems = child.getChildren();
            while (childElems.hasNext()) {
                Object c = childElems.next();
                if(c instanceof SOAPElement)
                    formLogMessage((SOAPElement)c, message);
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

Once this message handler is created, you need to associate it with your web service.

To associate a message handler with a web service:

1. Right-click the web service in the Projects window and choose Configure Handlers.
2. In the Configure SOAP Message Handlers dialog box (shown in [Figure 12-17](#)), click Add, and navigate to and select the message handler.

Figure 12-17. Configure SOAP Message Handlers dialog box



The IDE automatically updates the `webservices.xml` file under the `WEB-INF` directory of your web application by adding the `<handler>` element.

Following is an example of a `webservices.xml` file that the IDE has updated for you. (In general, you do not have to worry about this file at all. The IDE keeps it up to date for you.)

```
<?xml version='1.0' encoding='UTF-8' ?>
<webservices xmlns='http://java.sun.com/xml/ns/j2ee' xsi:namespaceURI='www.w3.org/2001/XMLSchema-instance' xsi:schemaLocation='http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/j2ee_web_services_1_1.xsd' version='1.1'>
```

```

<webservice-description>
  <webservice-description-name>
    NewWebService
  </webservice-description-name>
  <wsdl-file>WEB-INF/wsdl/NewWebService.wsdl</wsdl-file>
  <jaxrpc-mapping-file>
    WEB-INF/NewWebService-mapping.xml
  </jaxrpc-mapping-file>
  <port-component xmlns:wsdl-port_ns='urn:NewWebService'
    <port-component-name>NewWebService</port-component-name>
    <wsdl-port>wsdl-port_ns:NewWebServiceSEIPort</wsdl-port>
    <service-endpoint-interface>com.acme.NewWebService</service-endpoint-interface>
    <service-impl-bean>
      <servlet-link>WSServlet_NewWebService</servlet-link>
    </service-impl-bean>
    <handler>
      <handler-name>com.acme.NewMessageHandler</handler-name>
      <handler-class>com.acme.NewMessageHandler</handler-class>
    </handler>
  </port-component>
</webservice-description>

</webservices>

```

To see the effect of the message handler on the web service, perform the following steps:

- 1.** Run the web application by right-clicking its node in the Projects window and choosing Run Project.
- 2.** Add the web service to the IDE registry by right-clicking the web service's node in the Projects window and choosing Add to Registry.

- 3.** Switch to the Runtime window of the IDE; then navigate through the hierarchy of web service nodes, right-click an operation node, and choose Test Operation (as shown in [Figure 12-14](#)).
- 4.** In the Test Web Service Operation dialog box (shown in [Figure 12-15](#)), enter the input parameters and click the Submit button.

You can see the handler trace in the application server log file by opening the Runtime window, expanding the Servers node, right-clicking the Sun Java System Application Server node, and choosing View Server Log. A trace that looks something like the following should be displayed:

```
...
[#|2006-01-17T16:33:36.514-0800|INFO|sun-appserver-
pe8.2|javax.enterprise.resource.webservices.rpc.server
;|JAXRPCSERVLET14: JAX-RPC servlet initializing|#]

[#|2006-01-17T16:33:36.564-0800|INFO|sun-appserver-
pe8.2|javax.enterprise.system.tools.admin|_ThreadID=20
of dynamic reconfiguration event processing:[success]|:

[#|2006-01-17T16:33:46.909-0800|INFO|sun-appserver-
pe8.2|javax.enterprise.system.stream.out|_ThreadID=15;
Jan 17 16:33:46 PST 2006--getValidation String_1:SunCu
```

Chapter 13. Developing Full-Scale Java EE Applications

- [Entity Beans](#)
- [Creating Entity Beans with the Top-Down Approach](#)
- [Creating Entity Beans with the Bottom-Up Approach](#)
- [Assembling Enterprise Applications](#)
- [Importing Existing Enterprise Applications](#)
- [Consuming Java Enterprise Resources](#)
- [Java EE Platform and Security Management](#)
- [Understanding the Java EE Application Server Runtime Environment](#)
- [Ensuring Java EE Compliance](#)
- [Refactoring Enterprise Beans](#)
- [Database Support and Derby Integration](#)

THE PREVIOUS TWO CHAPTERS PROVIDED SOME STRATEGIES for extending web applications with Java enterprise-tier technology. This chapter continues the path of those chapters and handles topics such as entity beans, consuming resources, assembling applications from multiple code modules, and verifying the J2EE 1.4 compliance of enterprise applications.

Entity Beans

Beginning with the Enterprise JavaBeans 1.1 specification, entity beans have been a required part of the Java EE platform. Entity beans provide a Java *idiom* (method invocation) for accessing relational functionality in addition to the container benefits provided in the J2EE 1.4 specification, such as transaction support and security. An entity bean allows persistence to be handled by the container (*container-managed persistence*, or CMP) or by the developer (*bean-managed persistence*, or BMP). The difference between the two is that with CMP beans you need to define how the Java representation is mapped to the relational model (the container handles the code generation required to make this happen), whereas in BMP beans you must provide both the data representation and the implementation that reflects changes to the Java object model in the database. One common approach to bean-managed persistence is to use JDBC in the enterprise bean lifecycle methods.

A developer starting with an existing relational database must map the relational model currently in the database (if the database is in use by other applications, schema changes typically are not possible) to an object model. There are many possible complex mappings between an object model and the relational database. Relational database views are analogous; some of the same issues may be encountered during the mapping process (for example, a view can be created that cannot be updated, normally because of constraint violations).

Container-managed persistence provides some development advantages over bean-managed persistence. CMP beans allow you to work on an abstract schema, which is the combination of fields and relationships, independent of the underlying mapping to the relational database. The underlying mapping is provided by the application server, through the native object to relational mapping functionality. This capability extends to queries, which are written based on the abstract schema in the Enterprise JavaBeans Query Language (EJBQL). The query language is similar to the query capability provided in SQL.

This separation allows you to provide an object model and implementation that is separate from the mapping, thereby reducing the coupling. Thus, container-managed beans can be developed faster. Some proponents also claim that their runtime performance is better than that of BMP beans because the optimizations can be done in the server.

Bean-managed persistence requires that you implement the life-cycle methods to interact with the persistent storage mechanism. Typically, you would provide and execute SQL queries in the implementation of the life-cycle methods (`ejbStore` would perform an update, `ejbCreate` would perform an insert, and `ejbLoad` would perform a select). The disadvantage of doing JDBC directly in the entity bean is that you do not achieve productivity gains and may not achieve increased performance because of the impedance mismatch between the programming language and a relational database. (Programming languages operate on one object at a time, whereas a relational database can operate on sets of data.) The performance increase may not be realized, because one

common mapping is between a row and an object instance. In the entity model, set operations are performed by iterating over a collection of Java objects that cause SQL statements operating on a single row to be invoked. You may be able to incorporate existing tuned queries directly into this model or by using a different data representation to enable set-based operations.

NetBeans IDE provides both a top-down and a bottom-up approach for creating a related set of CMP entity beans. The top-down approach first creates the entity beans that describe the abstract schema (fields and relationships representing the data model). The Sun Java System Application Server provides a facility for automatically generating the required database tables. Or you can have the database schema created and use the application server mapping facility. The bottom-up approach generates the beans based on an existing database schema (the generation will produce CMP entity beans).

Creating Entity Beans with the Top-Down Approach

Creating an entity bean from the top down is comprised of the following general steps:

- 1.** Add a bean to your EJB module, using the Entity Bean template in the New File wizard.
- 2.** Select a primary key class for the bean, using the `ejb-jar.xml` file's visual editor.
- 3.** (For BMP beans) Implement the life-cycle methods.
- 4.** (For CMP beans) Add container-managed fields. You can generate such fields by right-clicking the entity bean's "logical" node (under the Enterprise Beans node) in the Projects window and choosing Add | CMP Field, or by right-clicking in the bean class in the Source Editor and choosing EJB Methods | AddCMP Field.
- 5.** (For CMP beans) Add container-managed relationships. You can do so in the `ejb-jar.xml` file's visual editor.
- 6.** Add finder methods. You can generate such methods by right-clicking a bean class in the Projects window and choosing Add | Finder Method, or by right-clicking the bean class in the Source Editor and choosing EJB Methods | Add Finder Method.
- 7.** Add business methods. You can generate such methods by right-clicking a bean class in the Projects window and choosing Add | Business Method, or by right-clicking the bean class in the Source Editor and choosing EJB Methods |

Add Business Method.

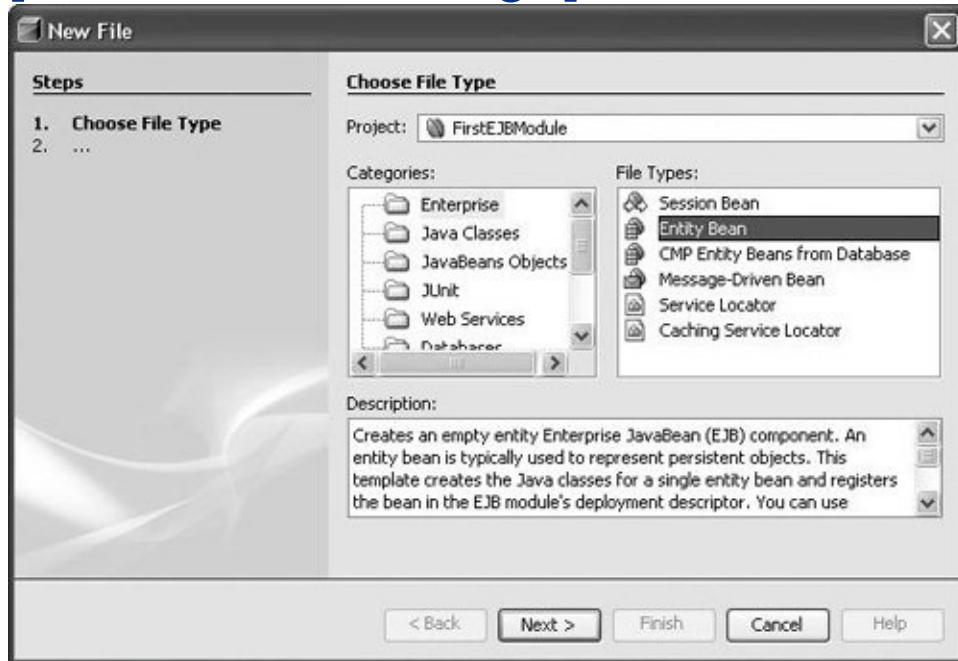
8. Set up the data source on the application server.
9. (CMP only) If the application server's CMP capability is being used, map the abstract schema specified in the entity bean to a persistence model.

Creating an Entity Bean

NetBeans IDE provides a wizard for creating entity beans in an EJB project (see [Figure 13-1](#)).

Figure 13-1. New File wizard with the Entity Bean template selected

[[View full size image](#)]

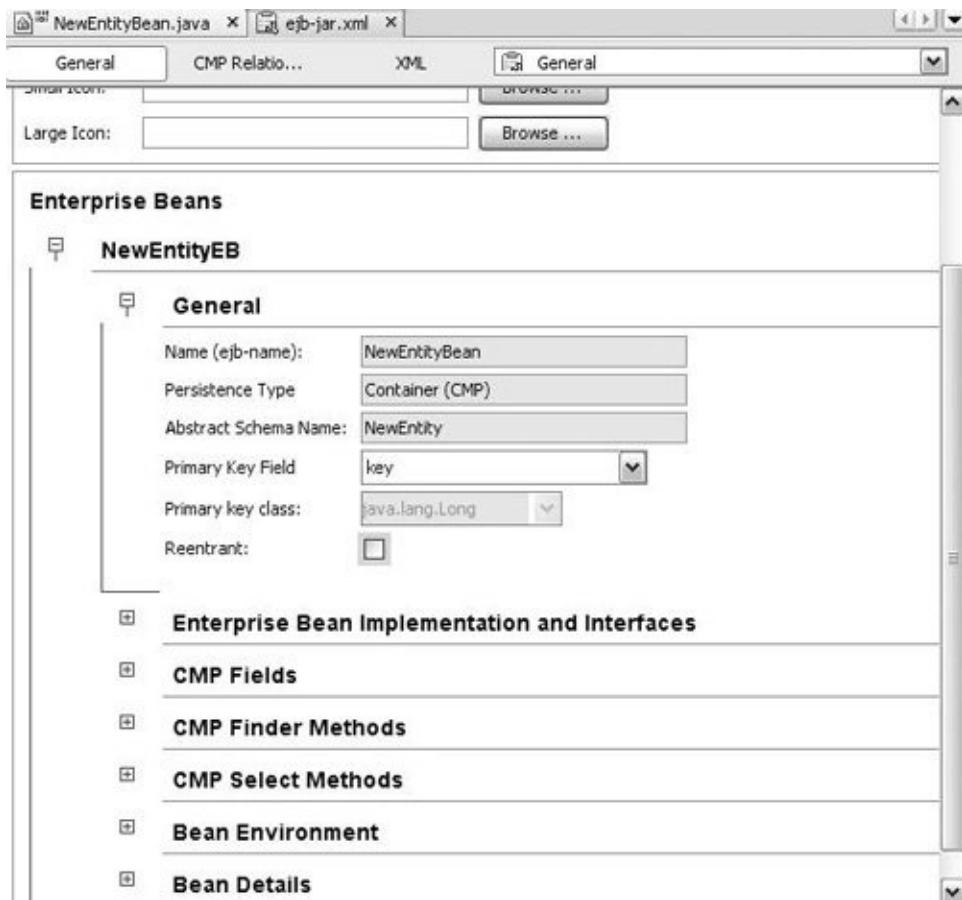


The wizard collects the minimum information necessary for creation of either a CMP or BMP entity bean. The wizard generates the necessary deployment descriptor entries along with a local home, local component interface (this provides the capability to perform compile-time checking on the bean class for the component interface methods), business interface, and a bean class.

The classes are named following the Java BluePrints recommendations. The component interface will initially be empty, and the home interface will contain the required `findByPrimaryKey` method. The primary key will default to `String`, which can be changed later using the visual editor (which is accessible by doubleclicking the `ejb-jar.xml` file, and can be found under the Configuration Files node in the Projects window). The visual editor for the deployment descriptor is shown in [Figure 13-2](#).

Figure 13-2. Visual editor for a CMP bean deployment descriptor

[[View full size image](#)]



The deployment descriptor entry includes a display name (using Java BluePrints recommended naming conventions) based on the EJB name. The assembly descriptor also is updated to require transactions for all methods.

Selecting a Primary Key Class

You can select a primary key class for the bean in the deployment descriptor (`ejb-jar.xml` file). A Java class representing the unique identifier (key) for each entity type must be selected. A Java class must represent the entire key. Compound keys and fields defined as primitive types require creation of wrapper classes. The wrapper class generally

overrides `equals` and `hashCode`. For CMP beans, this class must also provide public fields with names matching the CMP fields composing the key.

Implementing Life-Cycle Methods

For BMP beans, you need to implement the life-cycle methods. The required work will also be highlighted using TODO comments in the code generated by the bean template.

Typically, a relational database is used for persistence, so this may require configuration (see [Consuming Java Enterprise Resources](#) later in this chapter for more information). The `ejbStore` method must persist the data model from memory to the persistent storage (SQL update command). The `ejbLoad` method must retrieve data from the persistent model into the Java data model (SQL `select` statement). The `ejbRemove` method must remove data from the persistent model (SQL `delete` statement). Although not required, an `ejbCreate` statement can be used to add to persistent storage (SQL `create` statement). The `ejbFindByPrimaryKey` method is also required, which should determine whether the key is in persistent storage.

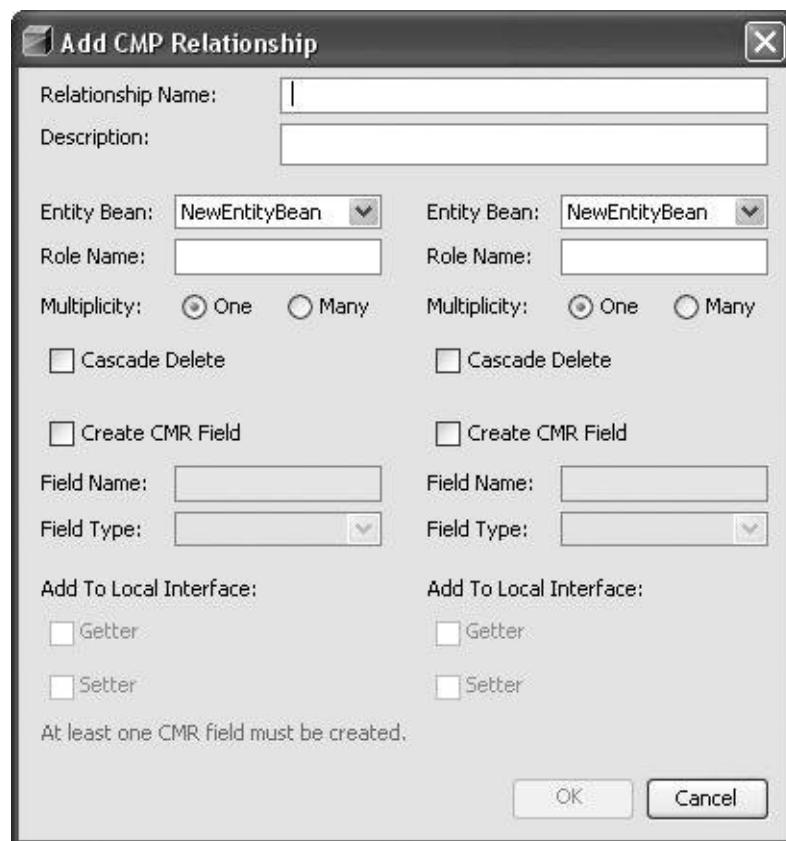
Adding Container-Managed Fields

For CMP beans, you need to add container-managed fields. Container-managed fields are the data attributed to this entity. Fields can be added via the EJB Methods menu in the Source Editor, the Add submenu of the contextual menu of the bean's node in the Projects window, or the visual editor for the `ejb-jar.xml` file. The wizards also provide checkboxes to expose the methods in the local interface.

Adding Container-Managed Relationships

(CMP only) To add container-managed relationships, select the CMP Relationships view in the `ejb-jar.xml` file's editor and then click the Add button to open the Add CMP Relationship dialog box (shown in [Figure 13-3](#)). A container-managed relationship represents an association between entity beans. The relationship is managed by the container, which means that adding or removing from one side of the relationship is reflected in the other side.

Figure 13-3. Add CMP Relationship dialog box



The Add CMP Relationship dialog box collects the information

necessary to generate the necessary deployment descriptor entries and the abstract get and set methods (if navigable).

In the Add CMP Relationship dialog box, the Entity Bean combo box represents the bean on each side of the relationship. The Cascade Delete checkbox enforces referential integrity by removing the dependent bean if the referenced bean is removed. A container-managed relationship (CMR) field (represented as an abstract getter and setter in the bean class) provides the capability to navigate to the other side of the relationship. These fields can be exposed using the Add to Local Interface checkbox.

Adding Finder Methods

Finder methods represent the capability to obtain a set of entity beans based on a set of search criteria. The Add Finder method dialog box can be accessed by rightclicking a bean class in the Projects window (and choosing Add | Finder Method) or the Source Editor (and choosing EJB Methods | Add Finder Method).

The EJB specification requires that a `findByPrimaryKey` method be present. CMP beans specify an EJB Query Language (EJBQL) query to define additional finder methods. The EJBQL query must return instances of the abstract schema for this EJB. The default abstract schema name is the enterprise bean name, so a typical bean query would be `SELECT OBJECT(o) FROM MyAbstractSchemaName AS o WHERE o.color = ?1`. BMP beans need to return the primary key (or keys) that meet the search criteria.

Adding Business Methods

Business methods provide the capability to expose services. Entity beans may add business methods to provide additional

logic or to expose data. Adding business methods may be done using the EJB Methods menu from the Source Editor contextual menu or from the Add submenu of the contextual menu for the enterprise bean's node in the Projects window.

Setting up a Data Source on the Application Server

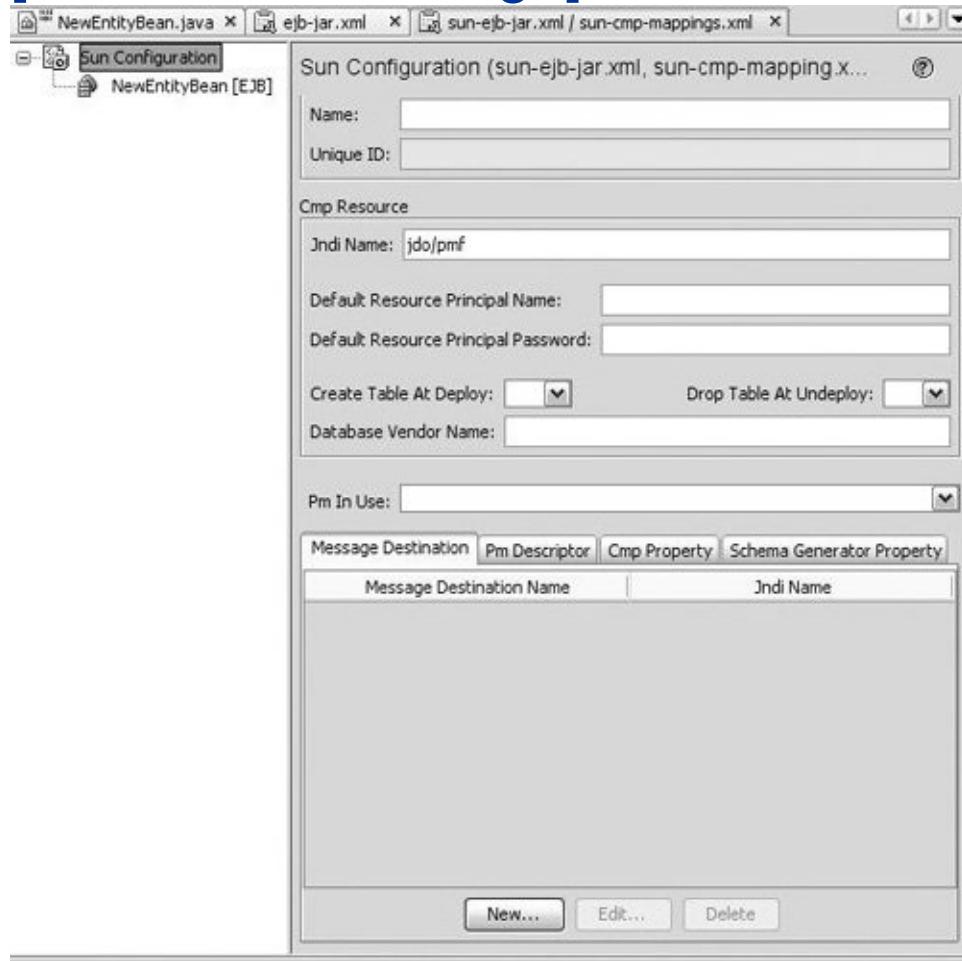
A data source must be set up on the application server. This applies to both BMP and CMP beans. BMP beans may be able to take advantage of the IDE's Use Database enterprise resource capability, whereas CMP beans generally must specify a reference to the application server's CMP resource. (The form varies among application servers but typically is similar to what is done to set up a standard JDBC data source.)

Mapping a Bean's Schema to a Persistence Model

If the application server's CMP capability is being used, the mapping from the abstract schema specified in the CMP entity bean to a persistence model (most commonly a relational database) must be performed. The Sun Java System Application Server provides the capability to create the tables during deployment (so no mapping is necessary in this case). This can be done by specifying the Create Table At Deploy (and, optionally, Drop Table at Undeploy) properties in the IDE's visual editor for the project's `sun-ejb-jar.xml` file, as shown in [Figure 13-4](#).

Figure 13-4. Visual editor for the `sun-ejb-jar.xml` file (Sun Configuration panel)

[View full size image]



Mapping is done by describing how each CMP field and relationship represents a column in the database. As of the EJB 2.1 specification, this mapping is server specific, and thus must be performed using the application server editor. When performing the mapping using Sun Java System Application Server (versions 8.1 and 8.2), the first step is to create a database schema file from an existing database. The database schema must be created as a peer to the `sun-ejb-jar.xml` file (which is located in the `src/conf` folder and can be accessed from the Configuration Files node in the Projects window). A database schema object provides an XML snapshot of the

database metadata. The application server uses this during design-time mapping as a way to access database metadata (without requiring a live database connection). This file is also passed to the application server during deployment for use during code generation.

Once the database schema object has been created, each CMP bean must specify the database schema that contains the table to which it is mapping. Double-click the node for the `sun-ejb-jar.xml` file to open its visual editor. In the visual editor, select the node for the entity bean in the left pane, select the Cmp Mapping tab, and click the Advanced Settings button. The visual editor with the Cmp Mapping tab selected is shown in [Figure 13-5](#). The Advanced Settings dialog box is shown in [Figure 13-6](#).

Figure 13-5. Visual editor for the `sun-ejb-jar.xml` file (EJB panel with Cmp Mapping tab selected)

[\[View full size image\]](#)

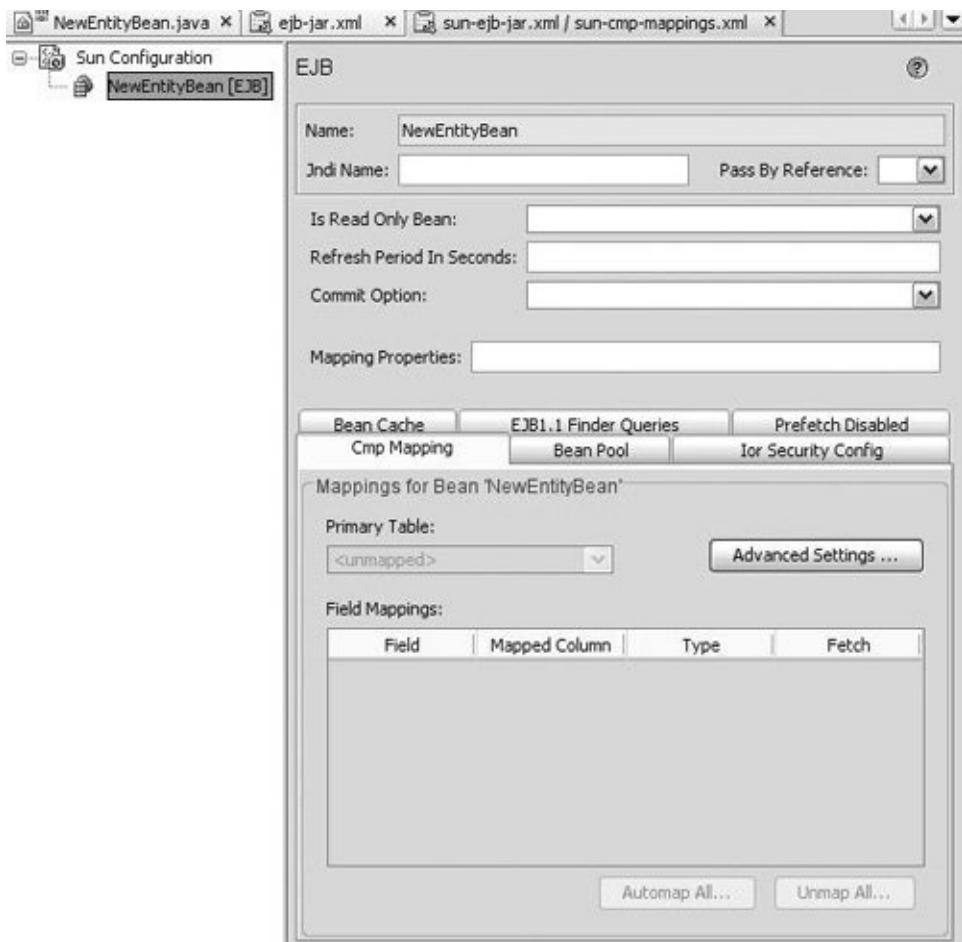
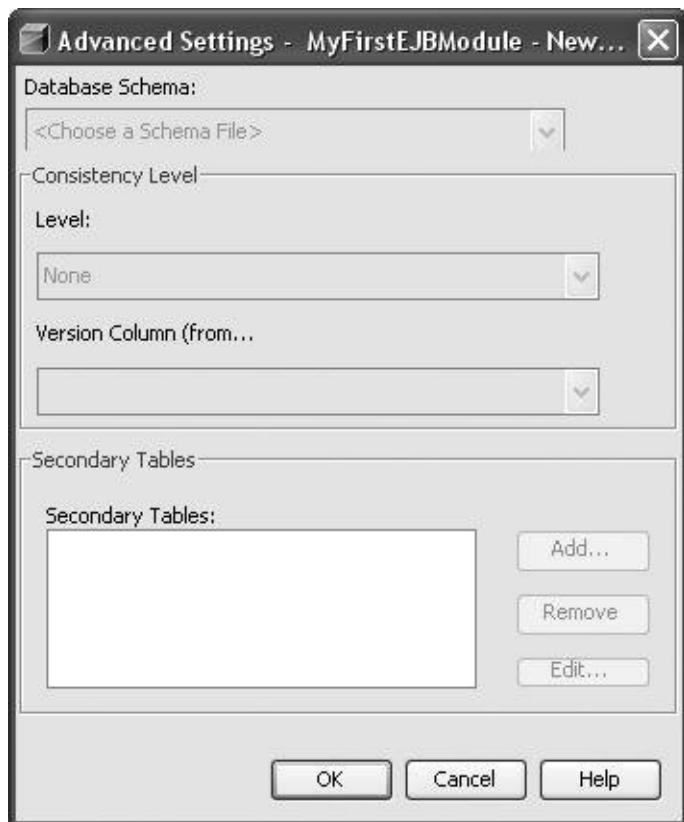


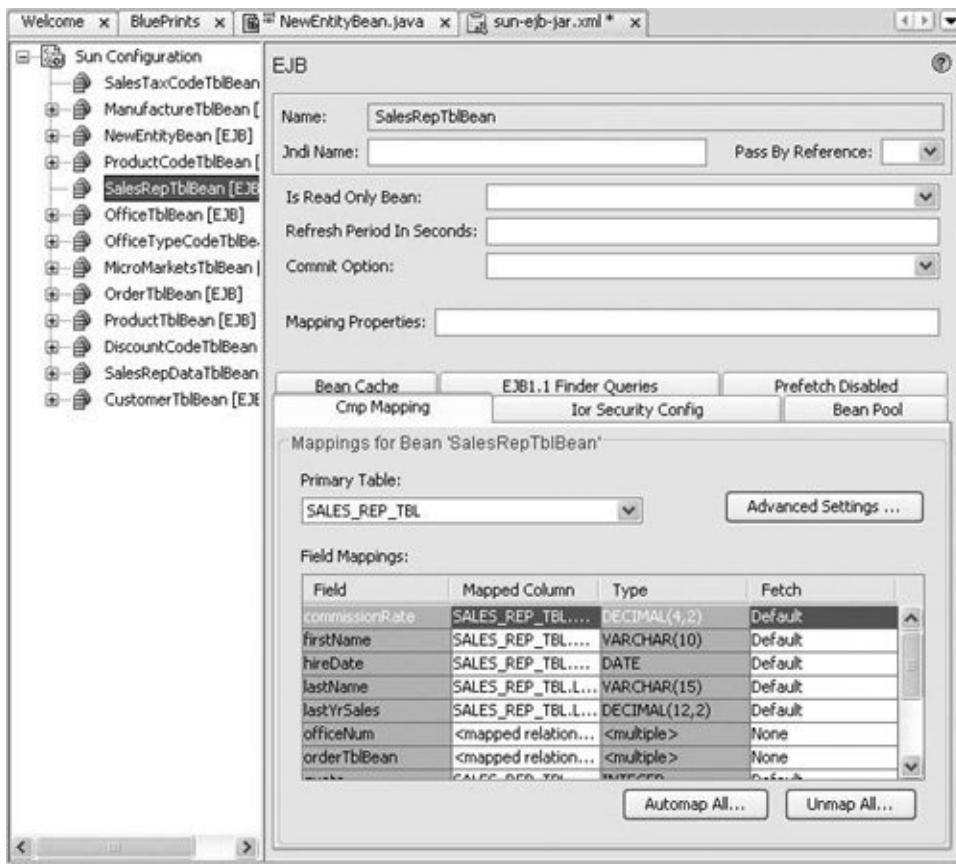
Figure 13-6. CMP Mapping Advanced Settings dialog box



The final step of mapping CMP and relationship fields from a table can be performed from the Cmp Mapping tab (see [Figure 13-7](#)).

Figure 13-7. CMP mapping in the sun-ejb-jar.xml visual editor

[[View full size image](#)]



The Automap All button automates the mapping between tables and fields. Using this feature provides full automation for most mappings, which can be further customized later (or even removed using the Unmap All feature).

Creating Entity Beans with the Bottom-Up Approach

Starting with an existing relational database and generating entity beans that represent a natural mapping is fully automated in NetBeans IDE. From a single wizard, you can create CMP entity beans, generate CMP fields with types appropriate for the relational mapping, generate relationship fields (along with the relationship cardinality) based on the foreign key and index constraints, and provide the mapping information to the application server (see the top-down approach for details when performed manually). The wizard will detect joined tables, so there may not be a one-to-one mapping between tables and entity beans from the selected database. The wizard will also generate primary key classes if necessary.

To create a set of entity beans based on an existing database:

1. In the Projects window, right-click the node of the EJB module that you want to add the beans to and choose New | CMP Entity Beans From Database.
2. Complete the wizard to create the mappings.

The wizard contains two steps (see [Figures 13-8](#) and [Figure 13-9](#)). The first step specifies how to obtain database metadata. The package for the generated Java classes is also required. The wizard also allows default generation of finder methods and exposing the fields in the component interface. Make sure the underlying database process is already started. If you are using the Derby database that is included with Sun Java System Application Server 8.2, you can start it from the IDE via the Tools | Derby Database menu item (Tools | Pointbase Database menu item if you are using version 8.1 of the application server).

Figure 13-8. Database Source and Beans Location page of the New File wizard for the CMP Entity Beans from Database template

[View full size image]

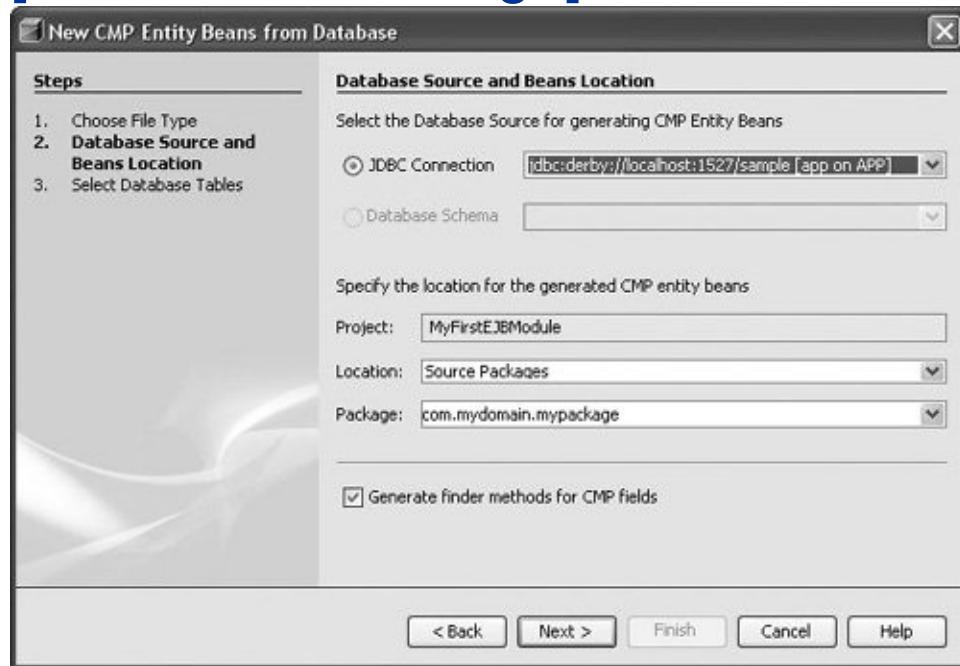
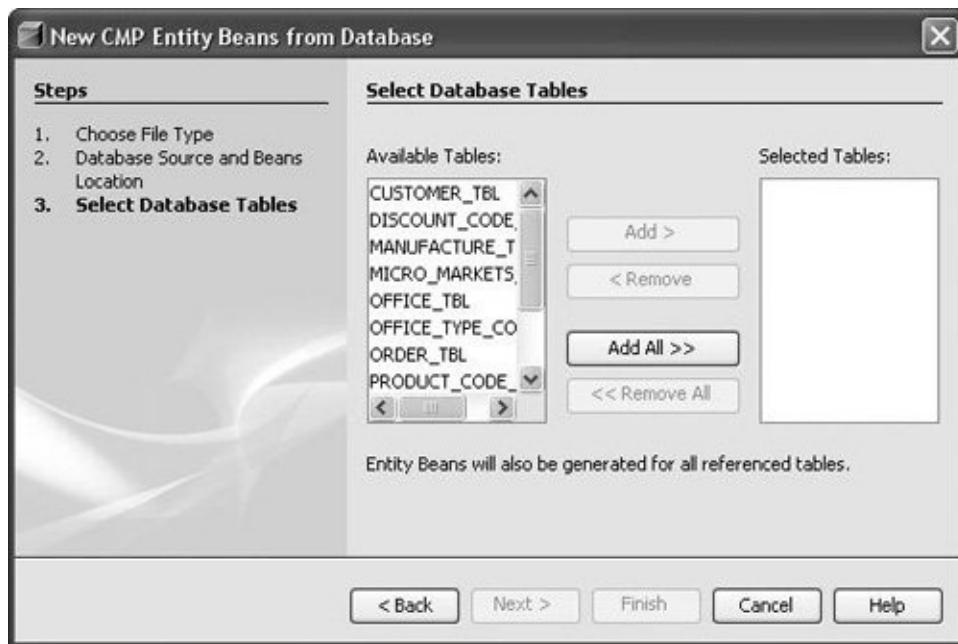


Figure 13-9. Select Database Tables page of the New File wizard for the CMP Entity Beans from Database template

[View full size image]



If the Generate Finder Methods for CMP Fields checkbox is selected, a singleargument finder method, along with the corresponding EJBQL query for selecting the collection of entity beans matching the input parameter, is generated for each CMP field.

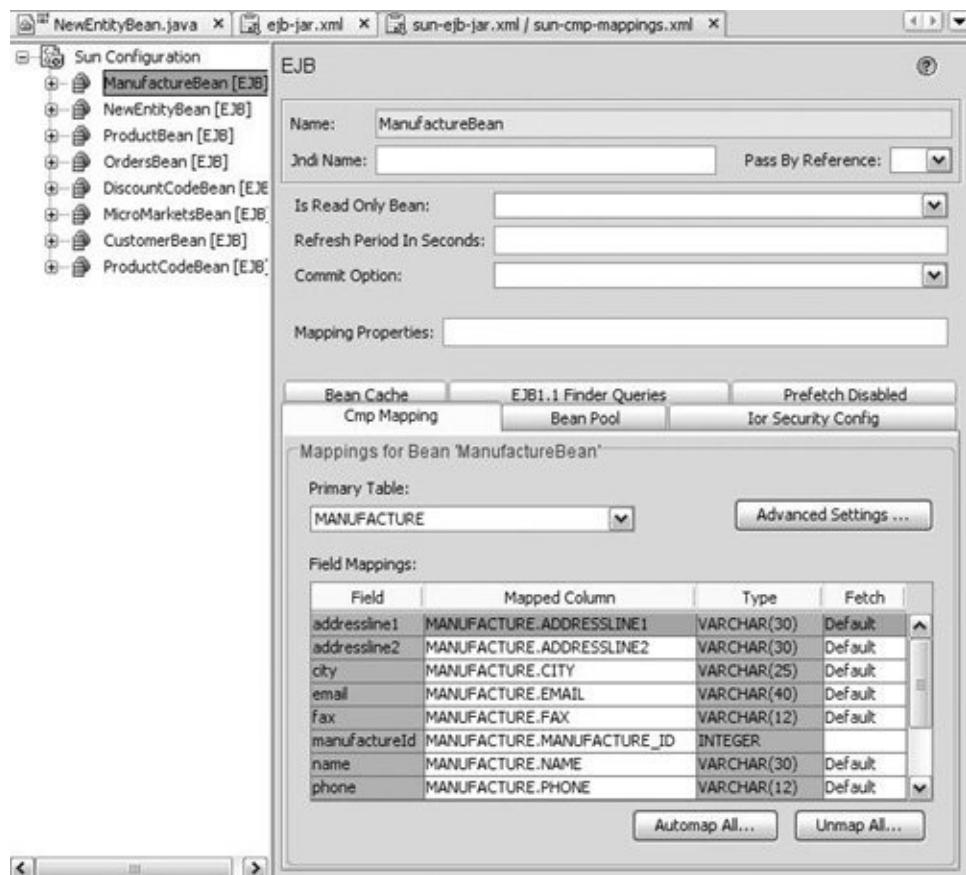
Selecting the Add CMR and CMP Fields to the Local Interface checkbox will generate a method in the local interface for each CMP and relationship field. This typically is used in conjunction with the transfer object and transfer-object assembler pattern from a session facade to create domain objects based on entity beans. The transfer objects are standard serializable Java classes that represent the data and can be used outside of Java EE components. The transfer object assembler typically is a Java class that can create transfer objects (from an entity bean, in this case). A session facade typically is used to provide a coarse-grained set of services. A session facade can encapsulate entity bean access and reduce coupling (and increase performance) for clients.

The second step of the wizard selects the set of tables to include in the generation. The wizard will perform transitive closure (based on foreign key constraints) to ensure that relationships are generated. The most common scenario is to select a group of logically related beans to generate CMP beans for a single module.

After you select all the tables you need, the IDE will create a set of CMP beans, one per table selected (and maybe others that are part of the transitive closure of the foreign keys/primary keys of the selected table). The Sun Java System Application Server mapping will be correctly configured. You can still modify it from the `sun-ejb-jar.xml/sun-cmp-mapping.xml` visual editor, as shown in [Figure 13-10](#).

Figure 13-10. Visual editor for the `sun-ejb-jar.xml/sun-cmp-mapping.xml` files with a CMP bean selected

[\[View full size image\]](#)



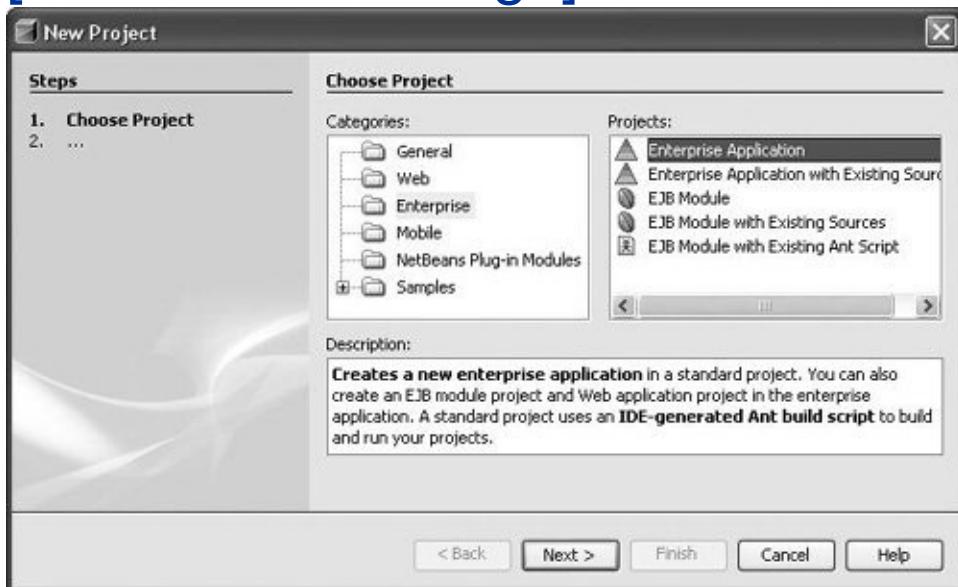
Assembling Enterprise Applications

A Java EE application is a collection of web applications and EJB modules that interact with one another and can be assembled, deployed, and executed as a unit. NetBeans IDE takes advantage of the capability to declare dependencies between projects in defining an enterprise application project. Previous chapters covered creation of individual web application and EJB Module projects. A NetBeans IDE enterprise application project aggregates these individual projects.

You can create an enterprise application project by opening the New Project wizard and selecting the Enterprise Application template as shown in [Figure 13-11](#). You can then add individual modules (one IDE project per module) to the enterprise application.

Figure 13-11. New Project wizard with the Enterprise Application template selected

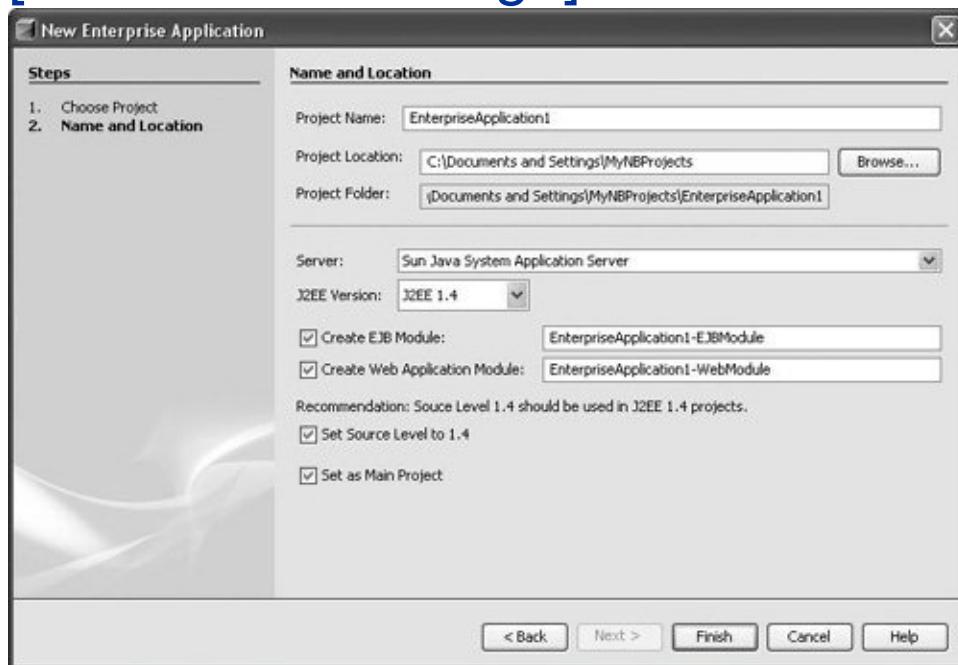
[[View full size image](#)]



When creating an enterprise application project, the Name and Location page of the wizard (shown in [Figure 13-12](#)) gives you the option of creating one empty EJB Module project and one simple Web Application project.

Figure 13-12. New Project wizard Name and Location page for the Enterprise Application project template

[[View full size image](#)]

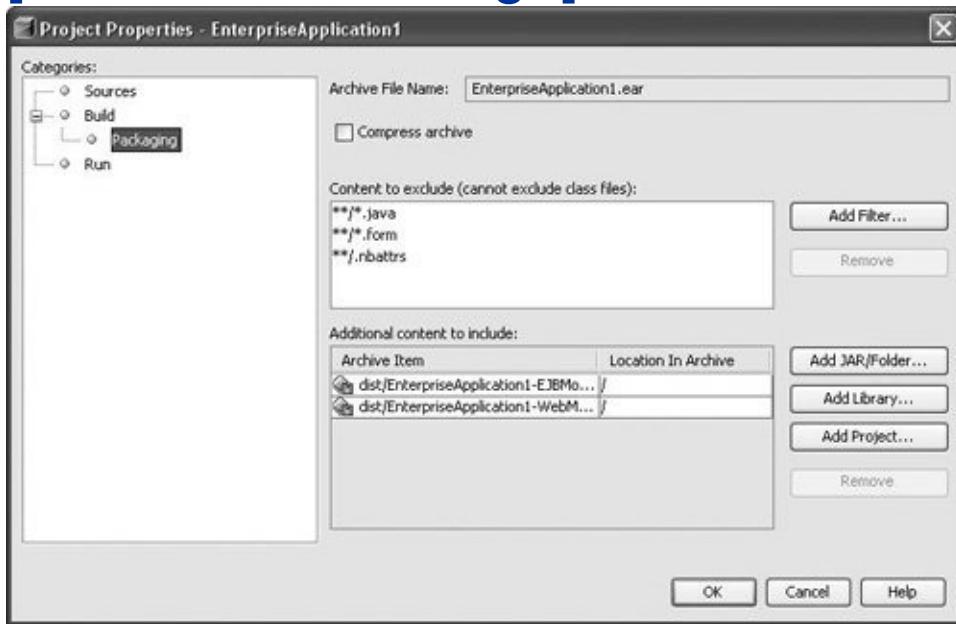


You can later add web application modules and EJB modules to an enterprise application project (or remove them from the project) through the Project Properties dialog box of the enterprise application project. To do so, right-click the enterprise application's project and choose Properties. Then, in

the Project Properties dialog box, select the Packaging node (see [Figure 13-13](#)) and click the Add Project button to add the web application or EJB Module projects.

Figure 13-13. Project Properties dialog box for an Enterprise Application project with the Packaging panel displayed

[[View full size image](#)]

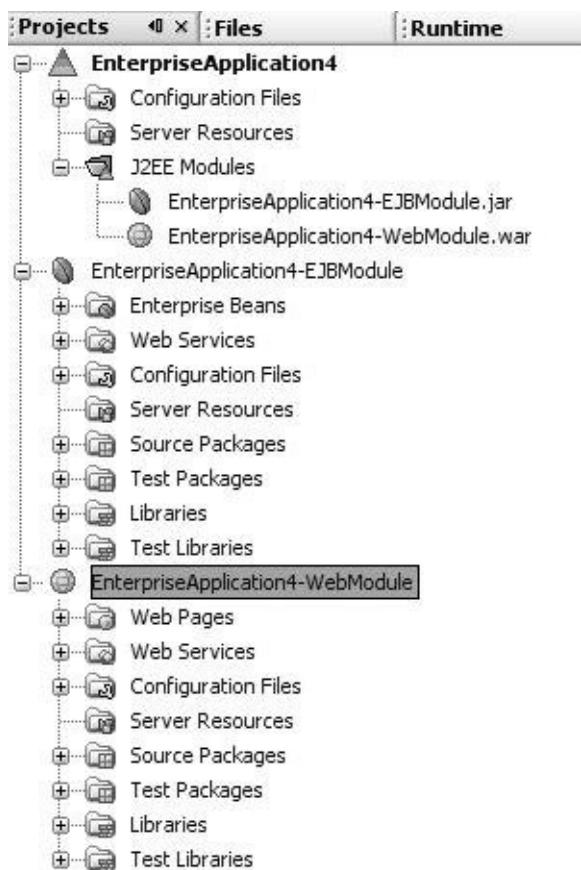


You will see multiple NetBeans IDE projects in the Projects window when you are working with an enterprise application (see [Figure 13-14](#)):

- The enterprise application project itself that exposes the application deployment descriptors (`application.xml` and possibly `sun-application.xml`)

- One NetBeans IDE project per application module

Figure 13-14. Projects window showing an Enterprise Application project that contains an EJB module and a web module (each of which is also an IDE project)



The enterprise application project delegates the build steps for all the sub-modules and packages the resulting EJB JAR files or web archive (WAR) files into a deployable application archive called an EAR (Enterprise Archive) file.

The deployment descriptor for an enterprise application is called

`application.xml` (under the Configuration Files node in the Projects window). NetBeans IDE does not provide a visual editor for this file, mainly because you are unlikely to need to edit it. The IDE automatically updates this file whenever a module is added to or removed from the application.

An enterprise application project can be cleaned, built, executed, and deployed like any other projects in the IDE. You can also verify the application's J2EE 1.4 compliance by running the Verify Project command on the project. Right-click the project's node to see the menu of commands.

Importing Existing Enterprise Applications

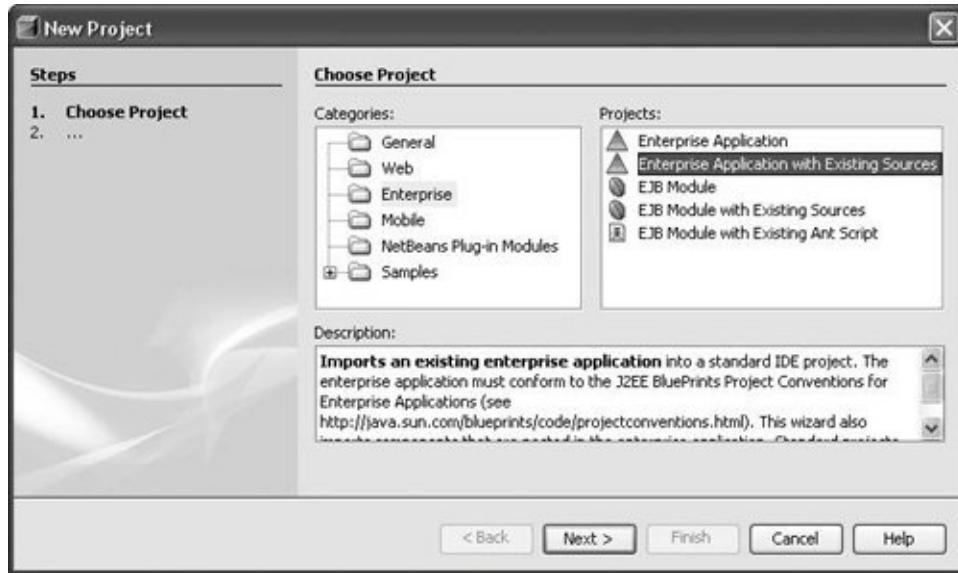
If you have already started developing Java enterprise application outside of NetBeans IDE, it is not too late to switch your development environment to NetBeans IDE. This process will be very easy for you if your source structure already complies with the Java BluePrints conventions. It might be a good time to study these conventions and modify your project to follow these guidelines. Your application will be easier to maintain, new developers joining your project will know immediately where things are, and the integration with NetBeans IDE and its powerful build system (entirely based on Ant) will be straightforward.

Importing Java BluePrints Conventions-Compatible Enterprise Application Projects

If you have started your development without NetBeans IDE, and you are following the Java BluePrints conventions for your project structure, you can use the Enterprise Application with Existing Sources template in the New Project wizard (shown in [Figure 13-15](#)).

Figure 13-15. New Project wizard with Enterprise Application with Existing Sources template selected

[\[View full size image\]](#)

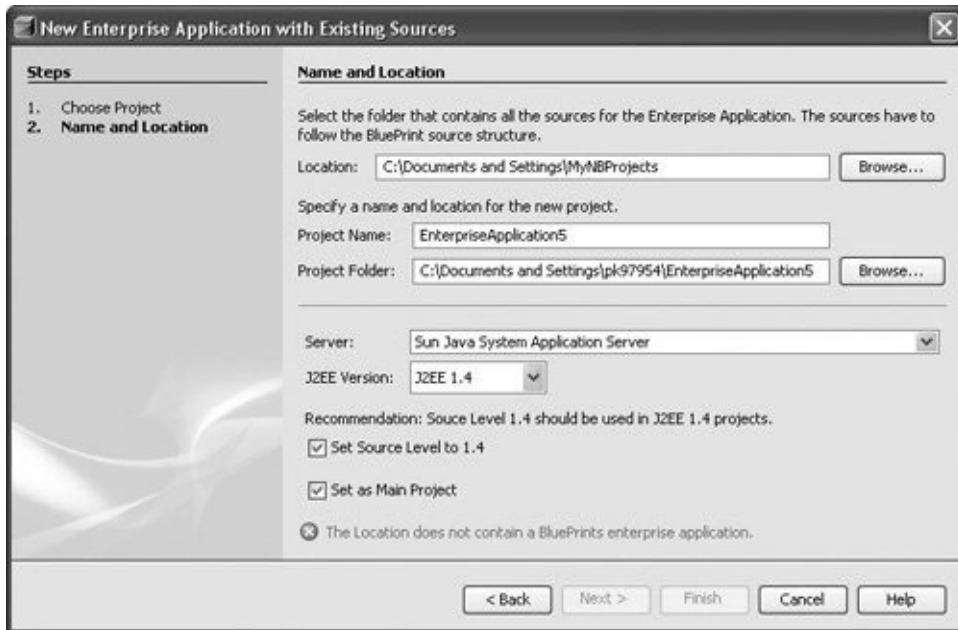


To import an existing enterprise application:

1. Choose File | New Project.
2. In the New Project wizard, select the Enterprise folder, select the Enterprise Application with Existing Sources template, and click Next.
3. In the Name and Location page of the wizard (shown in [Figure 13-16](#)), fill in the Location field with a valid top-level directory containing a Java BluePrints-compliant enterprise application.

Figure 13-16. New Project wizard Name and Location page for the Enterprise Application with Existing Sources template

[\[View full size image\]](#)



This directory should contain subdirectories for each module (web application or EJB Module) that comprises the enterprise application, and a `src/conf` subdirectory containing a valid enterprise application deployment descriptor file called `application.xml` (and, optionally, one called `sunapplication.xml`). If the Finish button does not get enabled, it means that these conditions are not matched.

4. Fill in the Project Name and Project Folder fields to designate a display name and folder to contain the IDE's settings for the project, including the Ant script. The project folder specified should not already contain a `build.xml` file, because the IDE will generate one automatically and will not replace an existing one.
5. After you click Finish, the enterprise application project is created, as are individual projects for each module in the application.

Importing EJB Modules

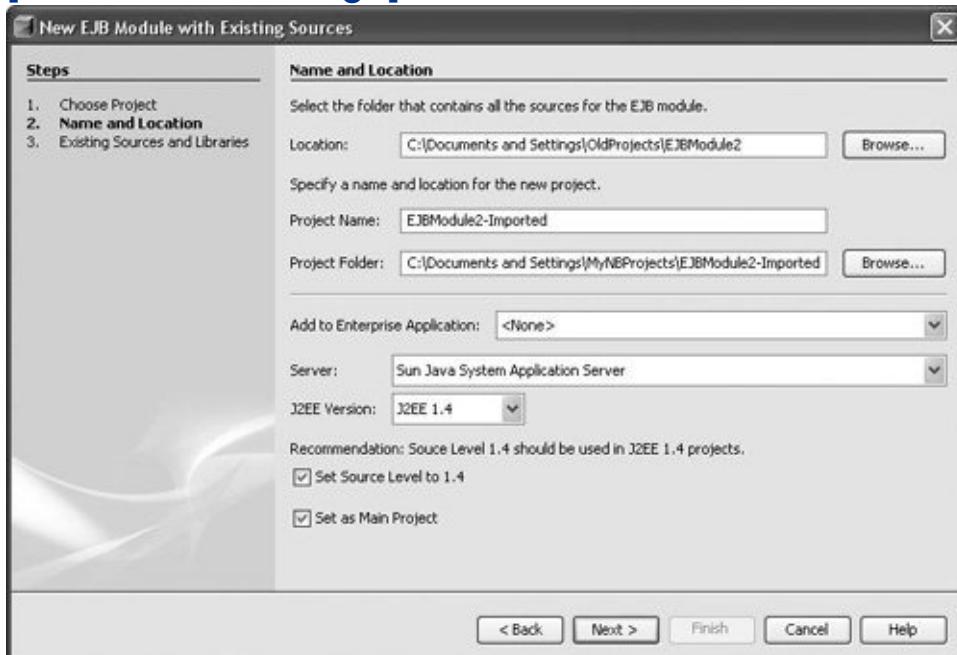
You can also import into the IDE stand-alone EJB modules (similar to how you import web application projects) by using the EJB Module with Existing Sources template in the New Project wizard.

To import an EJB module:

1. Choose File | New Project.
2. In the New Project wizard, select the Enterprise folder, select the EJB Module with Existing Sources template, and click Next.
3. In the Name and Location page of the wizard (shown in [Figure 13-17](#)), fill in the Location field with a valid top-level directory containing the EJB module.

Figure 13-17. New Project wizard Name and Location page for the EJB Module with Existing Sources template

[[View full size image](#)]



Your existing EJB module source does not have to adhere to any particular directory structure. You specify the locations of the configuration files, libraries, and source roots. The main requirement is that the module contain a valid `ejb-jar.xml` deployment descriptor.

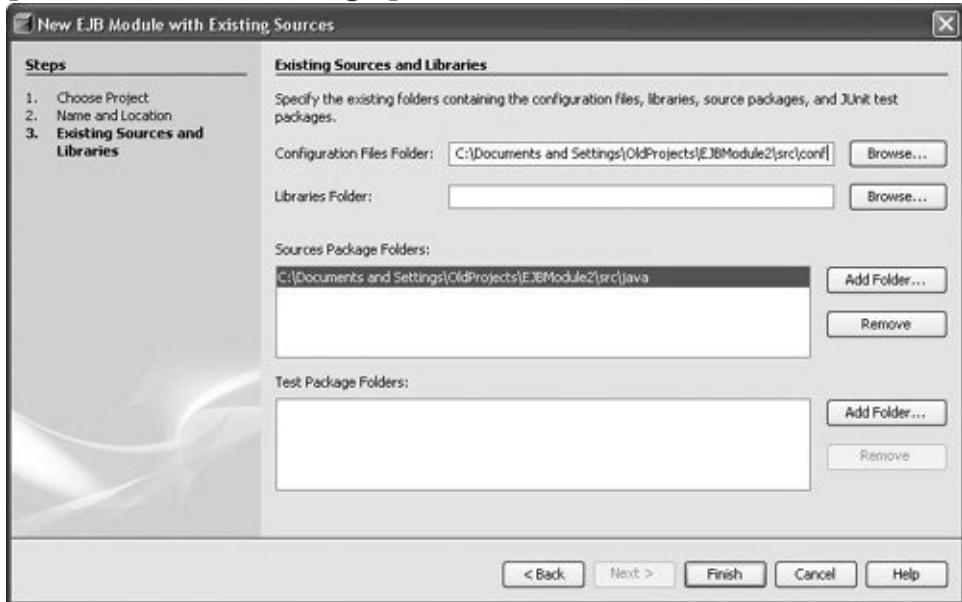
The location directory should not contain folders named `dist`, `nbproject`, or `build` or have an existing `build.xml` file. Warning messages should appear at the bottom of the wizard in such cases.

4. On the Existing Sources and Libraries page of the wizard (shown in [Figure 13-18](#)), specify the following properties:
 - **Configuration Files Folder.** Specifies the location of your deployment descriptors and other configuration files. You must have at least a valid `ejb-jar.xml` deployment descriptor to complete the wizard.
 - **Libraries Folder.** Specifies the location of the class libraries that the EJB module depends on. All JAR files in this folder are added to the EJB Module project's classpath and packaged with the module for deployment.

The IDE scans the folder you designate in the Libraries Folder field only once when you create the project. After the project is created, adding JAR files to this folder outside the IDE does not add them to the module's classpath. You have to add them manually through the Libraries tab of the module's Project Properties dialog box.

Figure 13-18. New Project wizard Existing Sources and Libraries page for the EJB Module with Existing Sources template

[View full size image]



5. In the Source Package Folders field, add any source root folders. In the Test Package Folders field, add any test package folders (containing JUnit tests).
6. Click Finish. A new NetBeans IDE project is created in the Projects window, and you can start using this EJB Module project like any other IDE project.



The IDE does not convert deployment descriptors from other application servers to Sun Java System Application Server deployment descriptors. An external tool you can use for this purpose, the Sun Java System migration tool, is available for free at

<http://java.sun.com/j2ee/1.4/download.html#migration>.

Consuming Java Enterprise Resources

One of the goals of deployment descriptors in the Java EE platform is to provide a standard way of describing external facilities that need to be available for successful execution of the enterprise application. The declaration in the deployment descriptor references the expected interface and the name used to locate this instance. This binding and declaration mechanism allows the late binding of resources. (In Java EE role terms, the deployer can change the actual resource used without changing the source code or the mandatory deployment descriptors.)

A *resource* is an external entity required by an enterprise application that is referenced using the standard deployment descriptor. A large number of resources can be used, but some of the most common are JDBC (via the `javax.sql.DataSource` interface), JMS (via the `javax.jms.*` interfaces), and enterprise beans (using the interfaces exposed by the bean developer).

Several steps are required to incorporate a resource into an application:

- Declaration of the resource in the standard deployment descriptor
- Resource setup in the server-specific deployment descriptor, which may also include setup on the server instance itself
- Acquisition of the resource using JNDI
- Using the resource in the programming environment

NetBeans IDE provides commands to automate the use of resources. Typically, you realize that a resource should be used

while you are writing Java code. The IDE makes it easy to add a resource as you are coding by providing an Enterprise Resources submenu on the contextual menu in the Source Editor for Java classes contained in enterprise projects (web applications and EJB Modules). See [Figure 13-19](#).

Figure 13-19. Source Editor with the contextual menu open and the Enterprise Resources submenu selected

The screenshot shows a Java code editor window with the file `NewServlet.java` open. The code defines a servlet named `NewServlet` that processes HTTP requests. A context menu is open at the end of the `processRequest` method, specifically at the line `getWriter();`. The menu is titled "Enterprise Resources" and includes options like "Call Enterprise Bean", "Use Database", "Send JMS Message", and "Send E-mail". Other menu items visible include "Go To", "Select in", "Refactor", "Reformat Code", "Fix Imports", "Surround With try-catch", "Run File", "New Watch...", "Toggle Breakpoint", "Cut", "Copy", "Paste", and "Code Folds".

```
package com.acme;

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class NewServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        getWriter();
    }

    // Rest of the code...
}
```

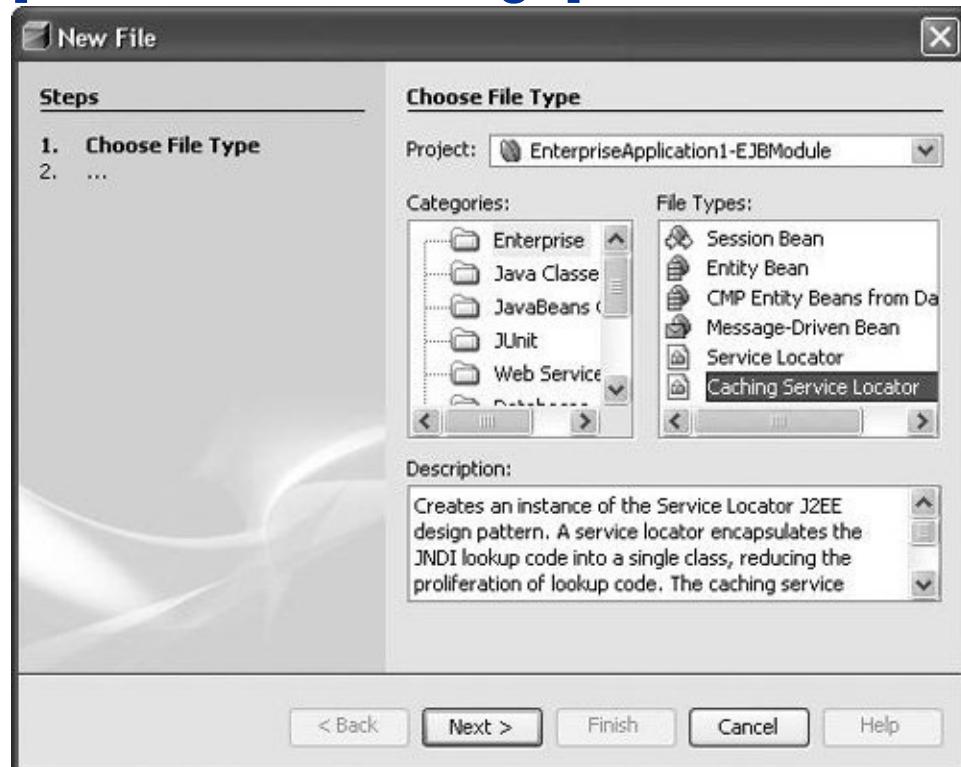
The Enterprise Resources menu provides a set of commands that automate the use of JMS, JDBC, enterprise beans, and email. Subsequent sections in this chapter describe what happens when the wizards are invoked and how to use the wizards.

All the actions are able to incorporate the Java BluePrints Service Locator pattern, which encapsulates and caches the JNDI initial context lookup. This pattern aggregates all boilerplate lookup code and provides the potential for performance enhancements.

NetBeans IDE provides two different service locator strategies: caching and non-caching. The caching strategy uses the singleton pattern and caches both the initial context and the results from the JNDI lookup. The non-caching locator does not cache the lookup results, but instead reuses the initial context. The Service Locator templates are available under the Enterprise node in the New File wizard, as shown in [Figure 13-20](#).

Figure 13-20. New File wizard with the Caching Service Locator template selected

[[View full size image](#)]



The Service Locator template generates a Java source file that can be further customized. The service locator last used for the project is stored with the project settings, allowing the last locator to be reused easily. The service locator naming conventions provided in the templates are incorporated into the enterprise resource commands; therefore, changing the name of the public lookup methods after generation may require manual intervention following the use of the enterprise resource commands.

The caching service locator strategy is most useful from a web module where resource definitions are shared across the entire module. The singleton pattern can be used effectively in this scenario to reuse lookup instances. In an EJB module, resource declarations are scoped to a single enterprise bean; thus, the singleton pattern is not applicable. An EJB module will commonly use a non-caching service locator.

Using Enterprise Beans

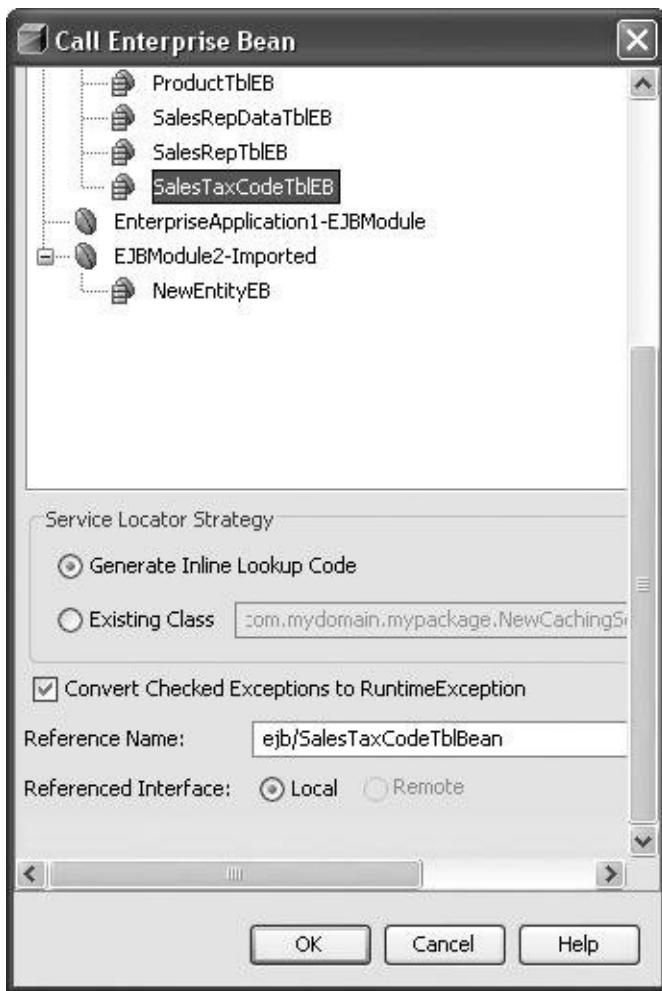
If you want to use an enterprise bean, perform the following steps:

1. Declare an `ejb-ref` entry or an `ejb-local-ref` entry (depending on whether the enterprise bean exposes local or remote interfaces) in the deployment descriptor. The deployment descriptor entry will specify the home and component interface, a name used in the JNDI lookup to specify the instance, the type of enterprise bean (either session or entity), and a reference to the actual enterprise bean.
2. Perform a JNDI lookup to obtain an instance of the home interface.

3. Use a create or finder method on the home interface to obtain an instance of the component interface.

The Call Enterprise Bean command (which you can access by right-clicking the Source Editor and choosing Enterprise Resources | Call Enterprise Bean) automates this process. The dialog box shown in [Figure 13-21](#) provides a way to select the enterprise bean to invoke. This dialog further enables you to specify the enterprise bean, the desired lookup strategy, and whether checked exceptions should be converted to runtime exceptions.

Figure 13-21. Dialog box for the Call Enterprise Bean command



Using the Call Enterprise Bean command in an EJB Module project or a Web project will result in the following:

- Generation of an `ejb-ref` or `ejb-local-ref` in the deployment descriptor. If this is done from an EJB module, this feature can be invoked directly only from the bean class. This feature requires that the referenced enterprise bean is in the same deployment unit (either the same module or the same enterprise application) as required by the semantics of the `ejb-link` (the reference is based on *ModuleName#EjbName* syntax and requires the reference to be collocated).

- Generation of the lookup code. If a service locator is being used, the generated code delegates the lookup to the locator; otherwise, lookup code is generated. The generated code uses the JDK logging framework to log the exception. The lookup code returns the home interface unless a stateless session bean is referenced, in which case an instance of the component interface is returned.
- Establishment of a project dependency between the current project and the referenced project, as the interfaces from the enterprise beans need to be used during compilation. The interfaces will not be included in the archive of the referencing project, but will instead assume that the server supports delegation to the application class loader (for references outside a module). If this is not true, the Java Extension mechanism can be used to add a classpath reference to the EJB module.

Using a Database

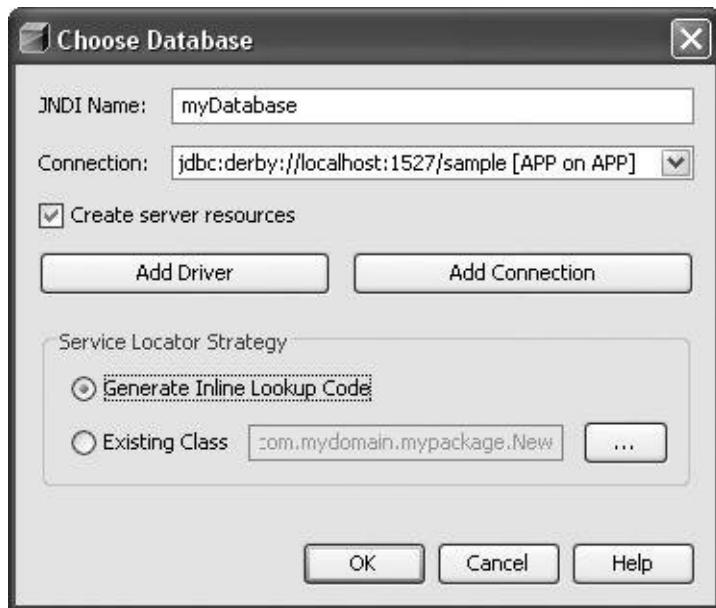
The Use Database command automates the process of incorporating a relational database into an application. If you intend to use a database, you must go through the following steps:

1. Declare a `resource-ref` entry in the deployment descriptor. The deployment descriptor entry must specify the resource-type `javax.sql.DataSource`, the ability to share the resource, and the authorization mechanism (typically, the authorization is done by the container).
2. Add code to perform a JNDI lookup to obtain an instance of `DataSource`.

- 3.** Use server specific mechanisms to configure the application server to use the JDBC resource. This typically involves adding the JDBC driver to the server classpath, as well as setting up a connection pool for the JDBC connection describing how to connect (including authentication) to the database.

The Enterprise Resources | Use Database command (available by right-clicking a file in the Source Editor) automates this process. When you choose this command, the Choose Database dialog box (shown in [Figure 13-22](#)) appears and prompts you to select the database to use in either a web module or an EJB module. In the dialog box, specify a JNDI name (used as the resource reference name), the connection to use, and whether a service locator should be used or inline code should be generated (lookup code is generated in the current class). The list of connections is the same as what is provided in the Runtime window.

Figure 13-22. Choose Database dialog box



After you fill in the Choose Database dialog box, the following occurs:

- A resource reference is added to the deployment descriptor using the form `jdbc/JNDIName`. Existing resource references are reused if possible (matching is done using the description, which initially is populated via the connection string).
- Code is generated to automate the JNDI lookup, using the specified lookup strategy to obtain the data source.
- If the Add Driver and Add Connection buttons are used, a connection node (and also possibly a driver node) is added to the Runtime window.
- Assuming that you are using the Sun Java System Application Server, the necessary steps to enable successful deployment and execution are performed. (Technically, it's the IDE's Sun Java System Application Server integration plug-in module that provides this support. By the time you read this, there might be IDE plug-in modules to provide equivalent features for different servers.) This might involve creating a connection pool and adding the JDBC driver to the classpath of the application server. The connection pool setup can be configured later in the Server Resources node or via the administrative capability of the server. The Server Resources node enables you to provide version control for and deploy server resources that are required to successfully execute an application. The resources provide all the necessary information to configure the setup. This may include usernames and passwords, so editing these files may be necessary when sharing the application with other team members.

Sending a JMS Message

The Send JMS Message command automates the process of sending a message to a queue or topic. The J2EE 1.4 specification defines the concept of a message destination, which provides a way to specify logical destinations for messages. A message destination allows the decoupling of the message consumer and message producer. The most common scenario is to have a message-driven bean (MDB) linked to the message destination. However, the message destination mechanism allows only the logical message destination to be used by the module sending the message. Although this concept defines a generic linking mechanism, additional information, such as the type of destination (the J2EE 1.4 specification allows other messaging systems to be used), and the message format must be exchanged.

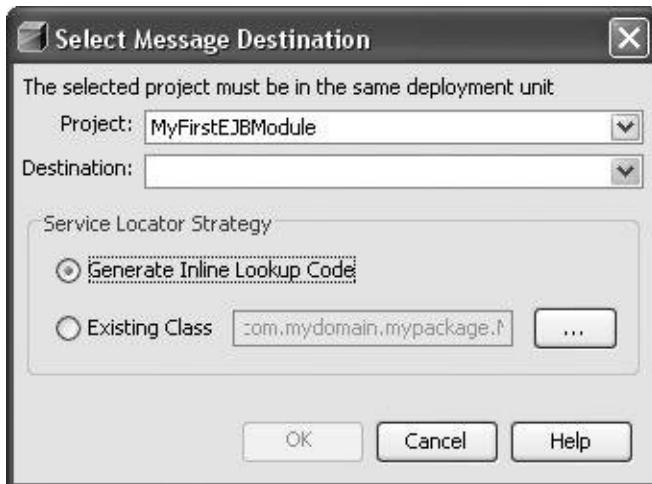
If you want to use JMS, you need to do the following:

- Add a resource reference to declare a `ConnectionFactory`. The connection factory is used to create JMS queue or topic connections.
- Add a message destination reference that declares the message destination type (which must match the type declared for the message destination), the use (the sender will produce messages), and the destination (in the form of a link). The link will be in the format `moduleName#destinationName` and must be included in the same application.
- Add code to send the message. The code to send a message must do a JNDI lookup to obtain the connection factory, use the connection factory to create a session, use the session to create a message producer, use the session

to create a message, send the message, and close the session and connection.

The Send JMS Message command automates this process. To use this command, right-click in the Source Editor and choose Enterprise Resources | Send JMS Message. The Select Message Destination dialog box (shown in [Figure 13-23](#)) appears and provides a way to select a message destination and to specify whether a service locator should be used or inline code should be generated (lookup code is generated in the current class). Only projects that can supply a destination are shown in the list of projects (web and EJB Modules).

Figure 13-23. Select Message Destination dialog box



After you complete the Select Message Destination dialog box, the following occurs:

1. A resource reference is added to the deployment descriptor, specifying the intention to use a [ConnectionFactory](#). The IDE's Sun Java System Application Server integration plug-in

module shares the connection factory with the message consumer. This can be changed by creating a new `ConnectionFactory`, if desired.

- 2.** A message destination reference is created and linked to the selected destination. The message type is discovered by determining the message type of the consumer linked to the destination.
- 3.** A private method providing the code necessary to send a single JMS message (named `sendJMSDestinationName`) is created. This method handles connection, session, and producer creation and destruction. This method accepts a context parameter that allows data to be passed via parameters instead of instance variables, although instance variables can be used as well.
- 4.** A second private method (`createJMSMessageForDestinationName`) is added to create the message (this is the template pattern) from the supplied session and potentially the context parameter.

Java EE Platform and Security Management

The Java EE platform offers a rich environment for securing web applications, web services, and enterprise beans in a declarative manner by working with application resources and user roles.

The two concepts are defined as follows:

- Resources are visible or callable features of the applications. For EJB modules, resources are public EJB methods declared on home or remote interfaces. For web modules, resources are URL patterns that are mapped to JavaServer Pages (JSP) files, servlet methods, and other components.
- Roles define access privileges and can be associated with one or more users.

Java enterprise applications are secured by mapping resources to roles. When a resource is called, the caller must map to a role name that is authorized to access the resource. If the caller cannot map to an authorized role, the call is rejected. In enterprise applications, the application server verifies the caller's role before allowing the caller to execute the resource.

The authorized combinations of roles and resources are declared in deployment descriptors. The application server reads them from the deployment descriptors and applies them. This process is known as *declarative security*.

The necessary tasks that you must perform to secure an enterprise application are:

- Declare the different roles.
- Specify which roles are permitted to access the resources.

This section describes the different steps to follow to secure a simple web application with NetBeans IDE. For a complete tutorial on enterprise application security, you can refer to the J2EE 1.4 Tutorial at <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security2.html>.

Simple Declarative Security

If you want to secure access for the web pages exposed within an enterprise application, you need to declare `<security-constraint>`, `<security-role>`, and `<login-config>` elements in the `web.xml` deployment descriptor (which you can find in the Projects window by expanding the web project's Configuration Files node). The visual `web.xml` editor does not expose those elements in NetBeans IDE, so you need to switch to the XML view and add the following elements:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>
            basic security test
        </web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>staffmember</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>basic-file</realm-name>
</login-config>

<security-role>
    <role-name>staffmember</role-name>
```

```
</security-role>
```

These settings protect the access of all the web pages (see the `<url-pattern>` element), using the BASIC login configuration. The authorized logical user is called `staffmember`.

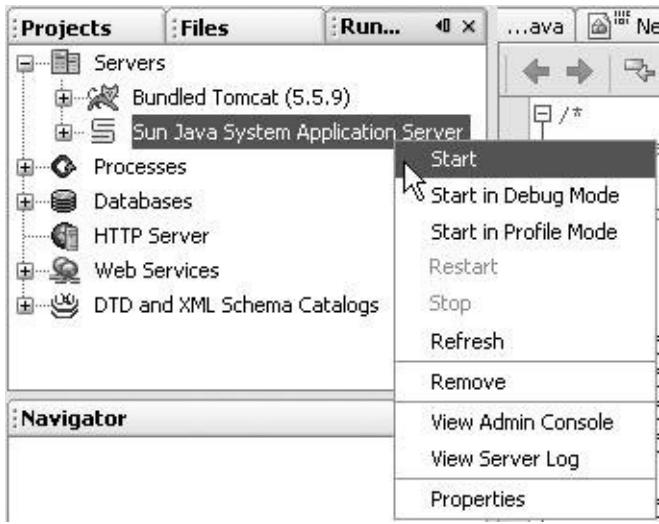
Authentication establishes the identity of a user by challenging the user to provide a valid username/password pair. Authentication can be used to protect any web-accessible resource, including web applications, web services, page flow applications, and individual JSP pages. In BASIC authentication, the browser provides the login window and it cannot be customized. If you require a customizable login page, use FORM Authentication.

Registering Users for an Application Server Instance

To add authorized users to the Sun Java System Application Server, follow these steps:

1. Make sure the server instance is up and running by opening the Runtime window, right-clicking the server instance's node, and choosing Start. If the Start menu item is not enabled, the server is already running.

Figure 13-24. Starting the application server from the IDE's Runtime window



2. Right-click the server instance's node and choose View Admin Console. The login page for the Admin Console appears in a web browser.
3. Log into the application server's Admin Console, and enter the username and password of a user in the `admin-realm` who belongs to the `asadmin` group. The name and password you entered when installing the server will work. The NetBeans IDE/Sun Java System Application Server bundle uses these default values: `admin` for the username and `adminadmin` for the password.
4. In the Admin Console tree, expand the Configuration | Security | Realms node and select the `file` realm to add users you want to enable to access applications running in this realm.
5. Click the Manage Users button.
6. Click New to add a new user to the realm. In this case, we will use the username `ludo` and the password `ludo` as well. You can also enter a group to which the user belongs, but leave that field blank for this example.

7. Click OK to add this user to the list of users in the realm. [Figure 13-25](#) shows the state of the console after entering the user `ludo` to the realm.

Figure 13-25. Sun Java System Application Server Admin Console after having created a user account

[[View full size image](#)]



8. Click Logout when you have completed this task.

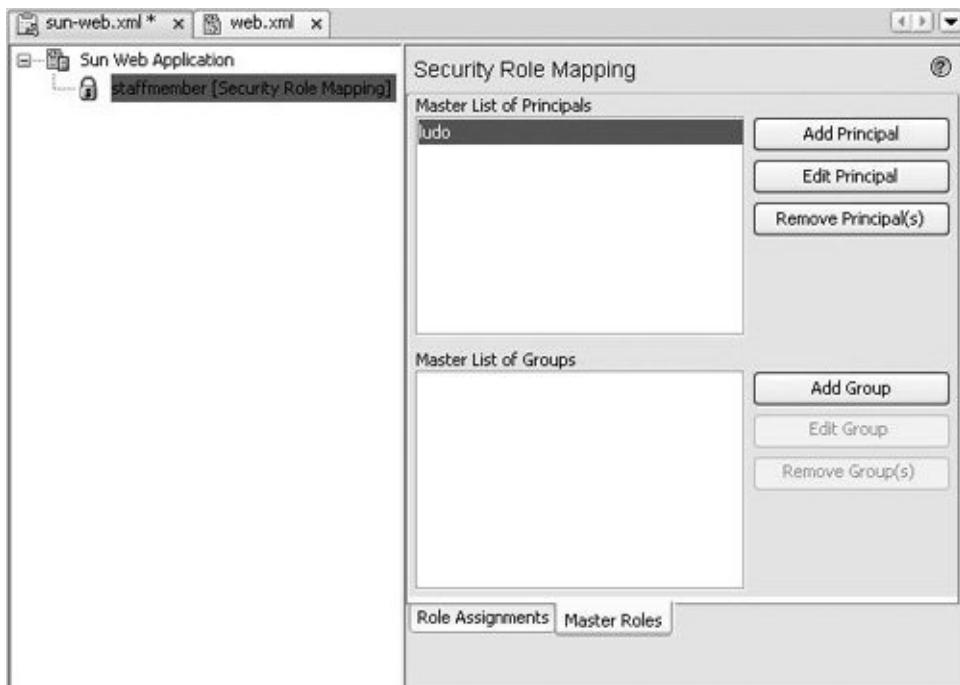
Now that you have registered a username for this application

server instance, you can map the logical security role called `staffmember` to this physical user called `ludo`. To do that, use the IDE's visual editor for the `sun-web.xml` file:

1. Open the `sun-web.xml` file from the Projects window by expanding the project's Configuration Files node and double-clicking the `sun-web.xml` node.
2. In the visual editor, expand the Sun Web Application node. If you have declared the `<security-constraint>`, `<security-role>`, and `<login-config>` elements as described previously in the Simple Declarative Security section, the `staffmember` node should appear.
3. Select the `staffmember` node, click the Master Roles tab in the right pane of the visual editor, click Add Principal, and type `ludo` in the New Principal Name dialog box. ([Figure 13-26](#) shows what the visual editor should look like after you have completed this step.)

Figure 13-26. Visual editor for the `sun-web.xml` file, Master Roles tab, after having added the principal `ludo`

[\[View full size image\]](#)



4. Click the Role Assignments tab, select `ludo` in the Principal Master List, and click the Add button to assign that user to the `staffmember` role. [Figure 13-27](#) shows what the visual editor should look like after you have completed this step.

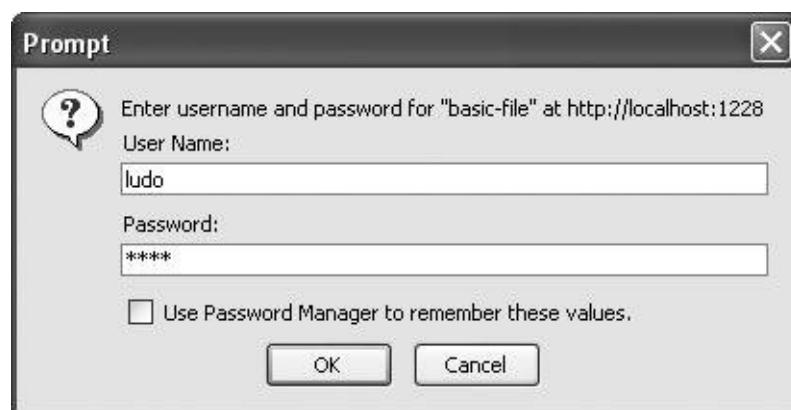
Figure 13-27. Visual editor for the `sun-web.xml` file, Role Assignments tab, after having mapped the `staffmember` role to the user `ludo`

[\[View full size image\]](#)



Now when you run this web application, you are prompted for a username and password when you first try to access the application's welcome (as shown in [Figure 13-28](#)). Enter `ludo` as the username and `ludo` as the password, and you should be able to access the requested web page.

Figure 13-28. Prompt for username and password to access a web application that has security constraints set up



This section is only an introduction to Java EE security settings. Make sure you read the J2EE 1.4 Tutorial, which covers more advanced security concepts for enterprise applications, such as web-services message protection for service endpoints (WSS in the SOAP layer is the use of XML Encryption and XML Digital Signatures to secure SOAP messages). You can find a complete tutorial at the following URL: <http://docs.sun.com/source/819-0079/dgsecure.html#wp14462>.

Remember, once you are ready to use more advanced security settings, you can take advantage of the IDE's visual configuration editors (for `sun-ejb-jar.xml` and `sun-web.xml`) to edit these security settings.

Understanding the Java EE Application Server Runtime Environment

The Sun Java System Application Server integrates nicely with NetBeans IDE, via both well-published interfaces (JSR 88) for configuration and deployment and NetBeans IDE extensions called the J2EE Server Integration APIs (see <http://j2eeserver.netbeans.org>). Although the most complete runtime management tool for the application server is the Admin Console (a management tool in the form of a web page shown in [Figure 13-25](#)), some developer-oriented management features have been exposed from the IDE's Runtime window under the Servers node. From there, for each registered server instance, you can view the server log file (for local servers), launch the Admin Console tool, explore the list of deployed enterprise applications or enterprise resources (and undeploy them), and more.



If you want to explore the entire content of an application server domain or to see the configuration files, the log files, or the repository of the deployed applications, you can add this domain directory in the IDE's Favorites window.

By default, the Favorites window is not visible within the IDE, but you can display it in the explorer area of the IDE by choosing Windows | Favorites. Then you can right-click the Favorites root node, choose Add to Favorites, and navigate to the application server's domain directory. Once you have added this directory in the Favorites window, you can view those files in the IDE's Source Editor and take advantage of other IDE features, such as the Validate XML command, which is useful for validating things such as the `config/domain.xml` file.

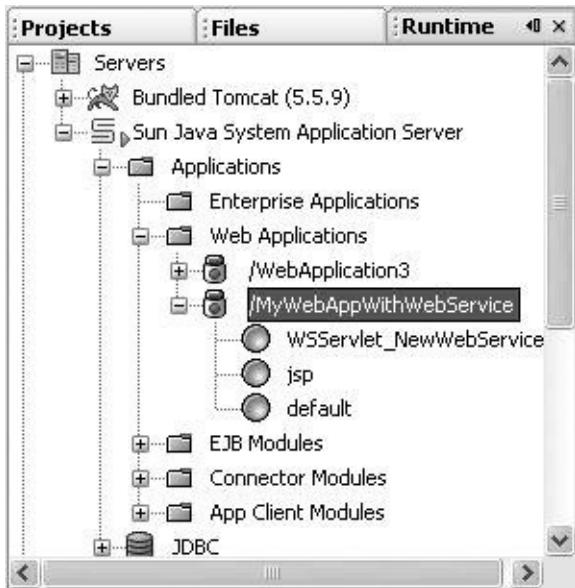
Server Log

The Sun Java System Application Server instance container process uses a log file to track system and user messages ranging from severe errors to informational messages describing the current state of the server. The server log file is an important source of information that an enterprise application developer needs to use actively during development or deployment and execution of Java enterprise applications. There is one server log file per instance, usually located under the domain directory in the `logs/server.log` file. NetBeans IDE can display the content of the most recent lines of the server log via the server instance node popup menu called View Server Log. This menu is active only for local server instances, because the IDE needs access to the file.

Server Management

For a registered application server instance, whether the server is local (on the same machine that the IDE is running on) or remote, the IDE offers extensive administration capabilities via the Runtime window tree view (shown in [Figure 13-29](#)).

Figure 13-29. Runtime window with the node for a Sun Java System Application Server instance selected



Once the server is running, you can get read/write access to most of the developer-oriented administration parameters, such as:

- The list of deployed enterprise applications, from where you can introspect their properties and undeploy them
- The list of registered resources and their properties (JDBC resources, connection pools, Persistence Manager resources, JMS resources, JavaMail resources, JDNI names, and Connector and JVM settings)

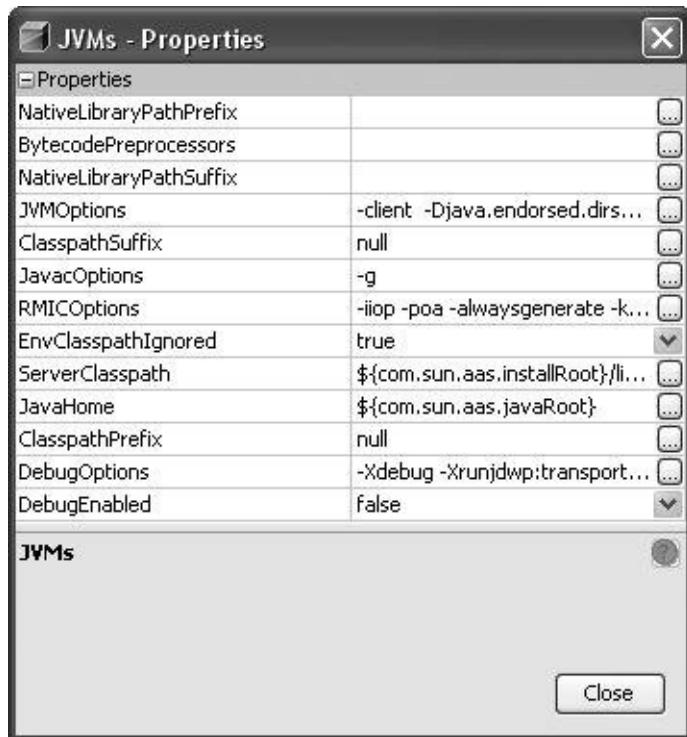
Each of these management artifacts is represented as a node. The corresponding commands (Delete, Undeploy, Refresh, and so on) are available as popup menu items (when you right-click the node), and corresponding properties are available via the Properties popup menu item. For other management artifacts, you can always access them by opening the application server's Admin Console (right-click the server's node in the Runtime window and choose View Admin Console). This command opens

a web browser that then displays the URL for the administration console.

JVM Options

The JVM parameters of a server instance are among the most important server administration parameters from a developer standpoint. You can access these parameters by right-clicking the server's JVMs node in the Runtime window and choosing Properties. The Properties dialog box for the JVMs node is shown in [Figure 13-30](#).

Figure 13-30. Property sheet for the JVMs node for an application server in the Runtime window



Use the `server-classpath` entry (shown as `ServerProperty` in the property sheet) when you want to add shared libraries or JAR files that would be used by all the enterprise applications deployed to this server instance, as well as the necessary JDBC driver JAR files to access databases. Use the `jvm-options` parameter (shown as `JVMOptions` in the property sheet) to define Java options like HTTP proxies or options that would be relevant to the deployed enterprise applications. You can modify the `java-home` entry (shown as `JavaHome` in the property sheet) to change the location of the JDK used by the server instance.

Most of these settings require a server instance restart to take effect. NetBeans IDE can figure out when a local instance has to restart, and at the next enterprise project deployment, execution, or debugging session, the server is restarted automatically. For remote servers, you have to log into the remote machine to perform a server restart, as there is no command (CLI or Admin Console based) to restart a remote server in Sun Java System Application Server 8.1.

JDBC Drivers

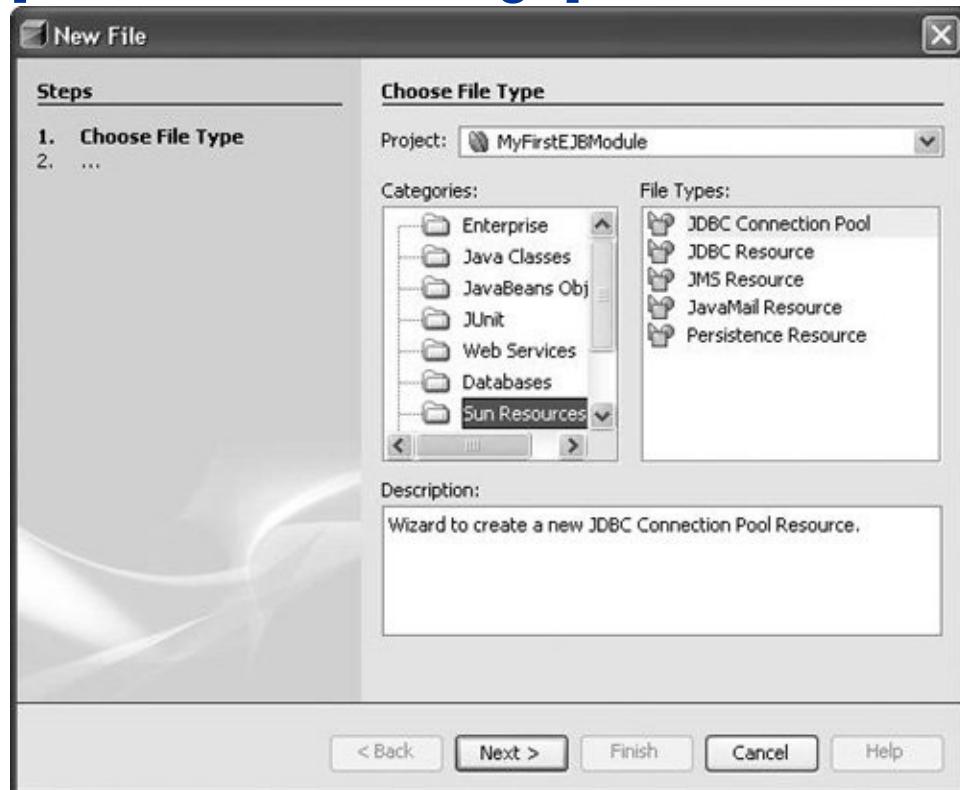
Using the Sun Resources template category in the New File wizard (shown in [Figure 13-31](#)), you can define Java enterprise resources such as:

- JDBC connection pools
- JDBC resources
- JMS resources
- JavaMail resources

- Persistence resources

Figure 13-31. New File wizard with the JDBC Connection Pool template selected

[[View full size image](#)]

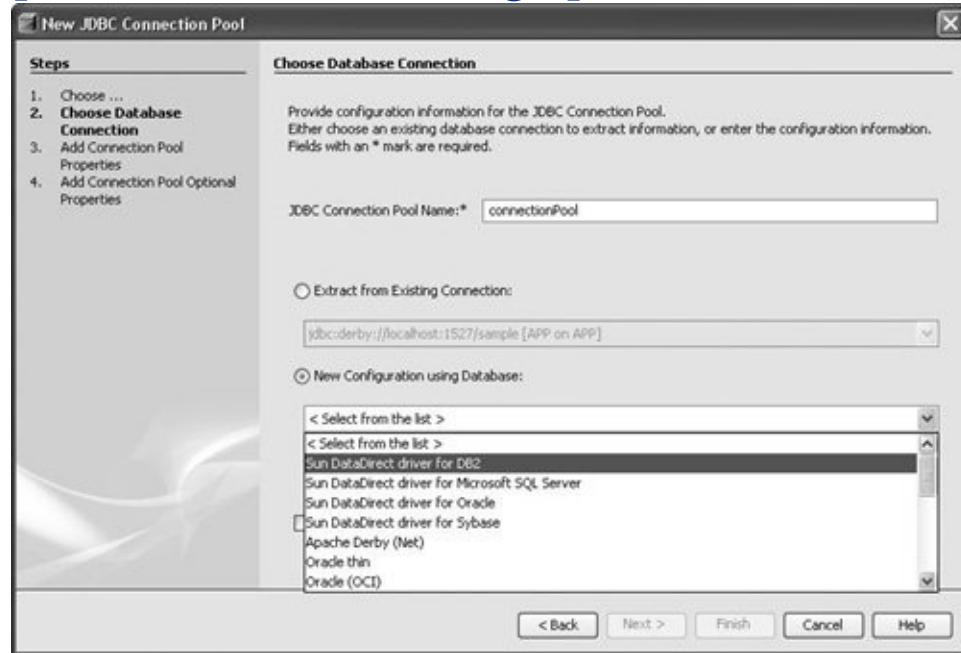


The wizard for the JDBC Connection Pool template (shown in [Figure 13-32](#)) is of particular help in improving your productivity. It enables you to create resources, either from live database connections registered within NetBeans IDE or from a predefined list of well-known JDBC drivers including all the Sun DataDirect drivers for DB2, Oracle, Microsoft SQL Server, and Sybase, as well as drivers for the Pointbase and Apache Derby databases. Note that in the Sun Java System Application Server

8.1 Platform Edition, only the Pointbase driver is provided; for versions 8.2 or 9.0, the Derby driver is provided. The Sun DataDirect drivers are included with the Standard Edition and Enterprise Edition of Sun Java System Application Server 8.x.

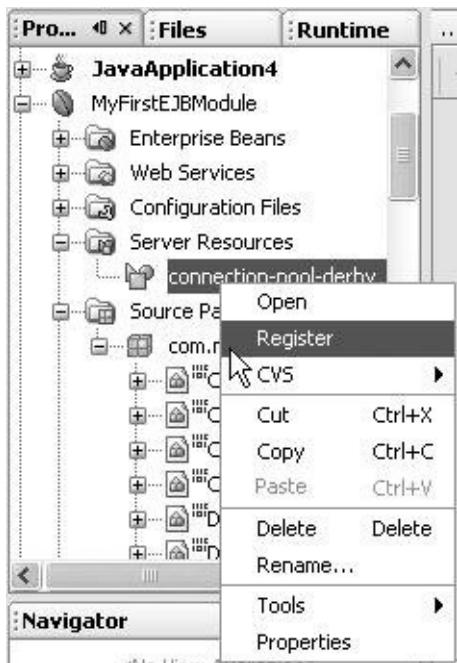
Figure 13-32. Choose Database Connection page for the JDBC Connection Pool template in the New File wizard

[[View full size image](#)]



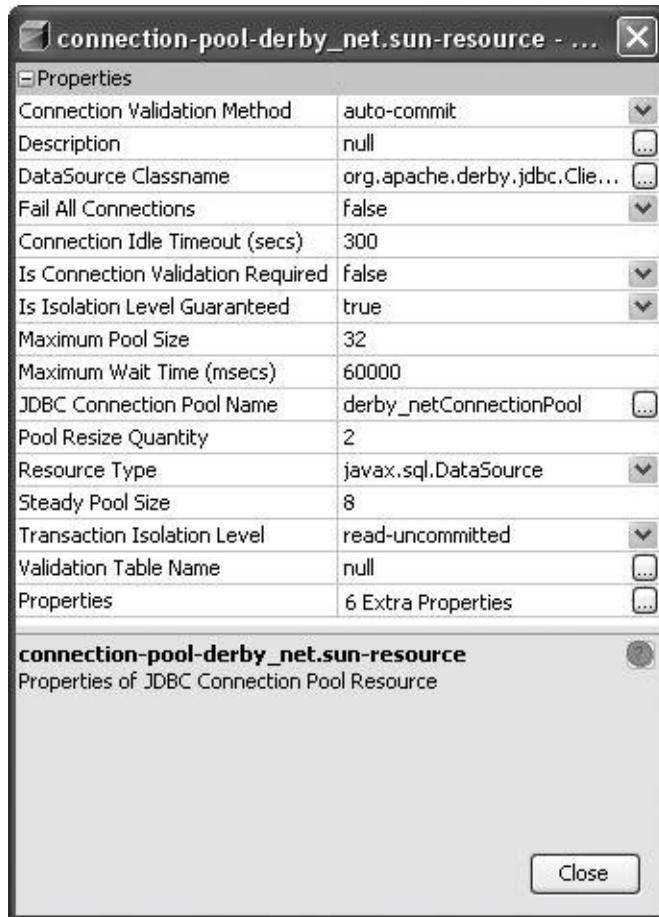
These server-specific resources are created under the `setup` directory, which in the Projects window is represented as the project's Server Resources node. These server resources are then either automatically deployed to the server whenever the project is executed or manually registered, using the Register menu item, as shown in [Figure 13-33](#).

Figure 13-33. Registering a database connection in a project



Double-clicking a server resource node opens the resource's property sheet (such as the one for the JDBC Connection Pool as shown in [Figure 13-34](#)), where you can modify all the necessary properties before doing a registration.

Figure 13-34. Property sheet for a JDBC connection pool

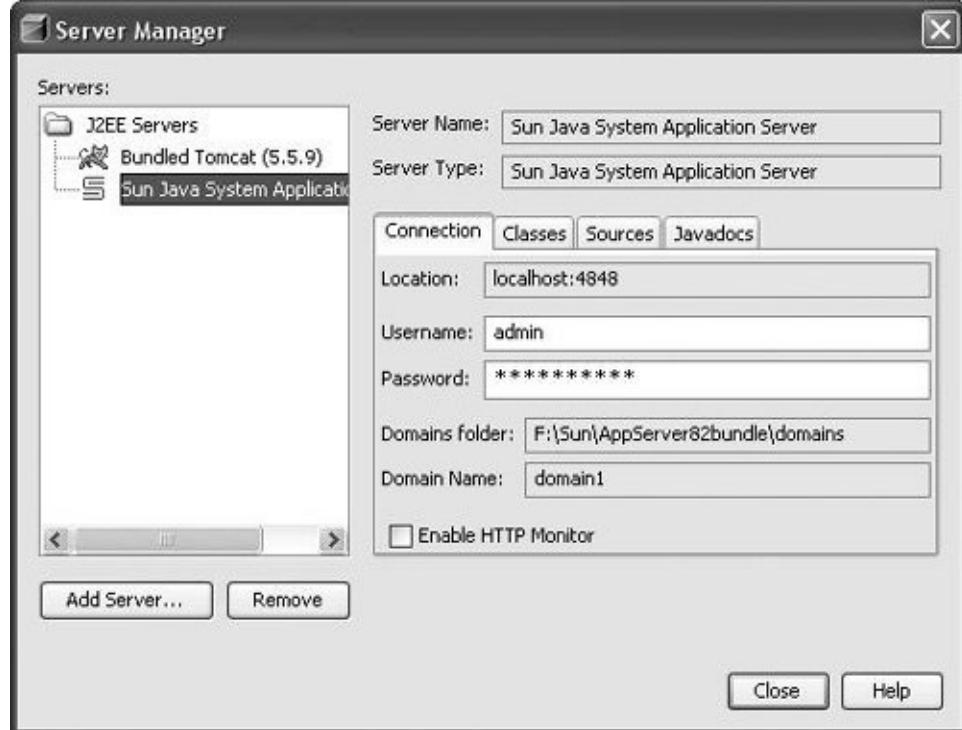


Server Properties

The server's Properties dialog box (shown in [Figure 13-35](#)), accessible via the Properties menu item of a server node, allows the editing of the admin username or password, as well as the setting for enabling the IDE's HTTP Monitor. The HTTP Monitor works only for local application servers.

Figure 13-35. Properties dialog box for the Sun Java System Application Server

[\[View full size image\]](#)



Using JavaServer Faces Technology in a Web Application

Sun Java System Application Server 8.2 bundles all the latest JavaServer Faces(JSF) APIs and the JSF implementation out of the box. The JSF config

DTDs are also registered in the IDE, meaning that XML code completion and validation are available in the IDE for the JSFconfig 1.0 and 1.1 XML files.

The JSF libraries are automatically in the web application project classpath when the project targets the Sun Application Server, so that all the classes are available to import and use

from your web application.

Furthermore, the Java BluePrints Solution Catalog is accessible directly through the IDE's Help | BluePrints Solutions Catalog menu item and contains some web tier solutions that incorporate JSF technology, which are installable as IDE projects with one click. Install a solution, run the project, study it, debug it, modify it, and use it as a starting point for your JSF project. See [Chapter 10](#) for more details.

Working with Ant

NetBeans IDE's Ant-based project system is perhaps the biggest thing that sets NetBeans apart from other IDEs. All projects created within NetBeans IDE rely on the Ant tool for building. This applies to projects types for web applications, EJB modules, and enterprise applications. Therefore, these projects can be built both inside and outside the IDE using Ant.

The IDE-generated build script also includes targets for executing, debugging, and deploying the application. In NetBeans IDE 5.0, it is not possible to use these targets outside the IDE, because they would require an operational runtime environment.

Another NetBeans IDE advantage is that all of the Sun Java System Application Server optional Ant tasks are registered in the IDE. These tasks include

- `sun-appserv-deploy`
- `sun-appserv-undeploy`
- `sun-appserv-instance`

- sun-appserv-component
- sun-appserv-admin
- sun-appserv-input
- sun-appserv-update

So if you have existing Ant scripts that are using these Ant task extensions, you can run these `build.xml` files directly from the IDE itself as though you were invoking the Sun Application Server tool `asant` (which itself is a wrapper around Ant to declare these extensions).

For example, if you download all the samples for the Sun Java System Application Server, all of the samples can be built from the IDE using the sample `build.xml` files provided with the sample. You could make the samples visible in the IDE by choosing Window | Favorites, right-clicking the Favorites node, choosing Add to Favorites, and navigating to the directory to display. Once the samples are visible in the Favorites window, you can navigate to a sample's `build.xml` file, right-click it, and choose an Ant target to run.

For more information regarding these tasks, refer to the application server documentation at <http://docs.sun.com>.

Ensuring Java EE Compliance

The Java EE platform extends the many benefits of the Java language to the enterprise, such as portability, by providing a set of specifications for application servers and an extensive set of compatibility tests. The Java EE platform enables you to run enterprise applications on a variety of Java EE application servers.

The IDE provides a rich environment for developing Java enterprise applications. But one of your most crucial objectives as an enterprise developer is ensuring that your application is Java EE compliant. By being Java EE compliant, the application will deploy and run in any Java EE vendor's application server as long as that vendor's application server is compliant with the given Java EE specification (e.g., the J2EE 1.4 specification).

To help you ensure that your enterprise application is Java EE compliant, the IDE provides a built-in verifier to help you test your web application or enterprise application for the correct use of Java EE APIs. In addition, the verifier confirms portability across Java EE compatible application servers, which helps you avoid inadvertently writing non-portable code. In NetBeans IDE 5.0, the verifier tests against the J2EE 1.4 specification.

When the verifier is invoked in NetBeans IDE, it runs a series of tests against a selected NetBeans IDE project. The output from running the verifier is displayed in the IDE's Output window in table form and contains the following information:

- Test Status pass, fail, or warning
- Test Description a description of the compatibility test that was run

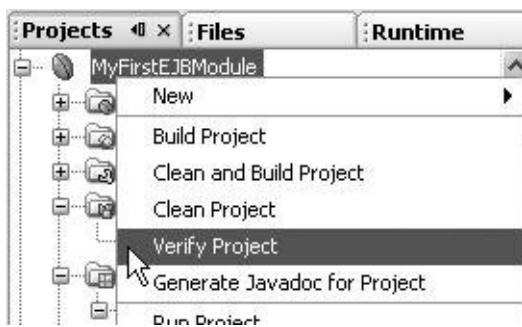
- Test Result Descriptiona detailed description of the test result

You can also filter the Verifier output. You can view all compatibility test results, compatibility failures only, or compatibility failures and warnings only.

After you finish developing a web application or enterprise application and before you deploy it, you should run the verifier against your project to ensure that you are not using any vendor-specific proprietary extensions to the Java EE platform. Using a proprietary extension will make it difficult to port to other Java EE application servers.

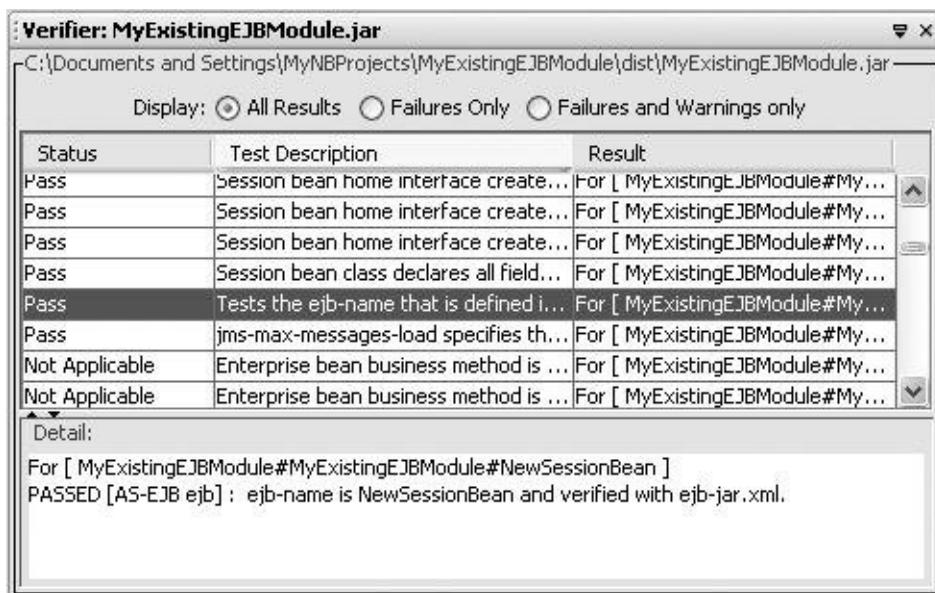
To launch the verifier against your web application or enterprise application project, you must have your project open in NetBeans IDE. From the Projects window, right-click the main node of the project you want to verify and choose Verify Project. [Figure 13-36](#) shows the contextual menu that is displayed when you right-click a web application or enterprise application project.

Figure 13-36. From the Projects window, running the Verify Project command for an EJB module application



Once the Verify Project command is chosen, the Verifier window (shown in [Figure 13-37](#)) appears at the bottom of the IDE and starts displaying results of the verification.

Figure 13-37. Output from the Verify Project command



The screenshot shows the 'Verifier' window titled 'Verifier: MyExistingEJBModule.jar'. The path 'C:\Documents and Settings\MyNBProjects\MyExistingEJBModule\dist\MyExistingEJBModule.jar' is displayed above the table. Below the table, a 'Detail:' section shows the result for a specific test: 'For [MyExistingEJBModule#MyExistingEJBModule#NewSessionBean] PASSED [AS-EJB ejb] : ejb-name is NewSessionBean and verified with ejb-jar.xml.'

Status	Test Description	Result
Pass	Session bean home interface create...	For [MyExistingEJBModule#My...
Pass	Session bean home interface create...	For [MyExistingEJBModule#My...
Pass	Session bean home interface create...	For [MyExistingEJBModule#My...
Pass	Session bean class declares all field...	For [MyExistingEJBModule#My...
Pass	Tests the ejb-name that is defined i...	For [MyExistingEJBModule#My...
Pass	jms-max-messages-load specifies th...	For [MyExistingEJBModule#My...
Not Applicable	Enterprise bean business method is ...	For [MyExistingEJBModule#My...
Not Applicable	Enterprise bean business method is ...	For [MyExistingEJBModule#My...

The results consist of all of the compatibility tests performed against your application. The Verifier lists test status, test description, and a detailed test result. The default for the Verifier results is to display all compatibility tests that fail first, followed by those that have a warning result and then those that have passed. When a row or test is selected in the table, a detailed description of the test result is displayed. Also, the Verifier test results display can be filtered by selecting one of the following radio buttons:

- All Results displays all of the compatibility test results

- Failures Onlydisplays only the compatibility test failures
- Failures and Warnings Onlydisplays both compatibility test failures and warnings

You should fix all compatibility test failures before deploying the application. As you make corrections to your web or enterprise application, you can rerun the Verifier by again right-clicking the project's node in the Projects window and choosing Verify.

Refactoring Enterprise Beans

Software applications evolve over time and require updates, bug fixes, feature enhancements, and so on. To support this type of development activity, NetBeans IDE refactoring features extend to Java enterprise applications.

Enterprise beans are among the types of code most likely to need refactoring during an enterprise software application's lifetime. However, refactoring enterprise beans can be very time consuming as a result of the complexity of changes that may be involved with the refactoring, due to the number of classes and deployment descriptors that may be affected by a refactoring.

For example, a business method rename may require changes not only in the EJB class you are editing, but also in a local interface, remote interface, calling classes (such as a servlet class or a remote client class), and deployment descriptors. NetBeans IDE simplifies this task by applying the refactoring changes to the Java source code, and by making those more difficult changes in deployment descriptors.

Following are some of the things you can do when refactoring your enterprise project source code.

- Rename an enterprise bean
- Rename a field or method of an enterprise bean
- Change method parameters (including parameter names, parameter types, and method visibility), add method parameters, and remove method parameters
- Encapsulate fields by changing their visibility, adding getter and setter accessors, and determining whether to use

accessors even when the field is accessible

NetBeans IDE allows you to review your refactoring changes before applying them. Likewise, you can undo your refactoring changes after having applied them, including any changes that have been made to deployment descriptors.



If you want to undo a refactoring operation, you must call the Undo command from a Java source file. Although Undo affects deployment descriptors, the command cannot be called from a deployment descriptor.

For complete descriptions of the IDE's refactoring features, see [Chapter 5](#). For notes on servlet refactoring, see [Chapter 8](#).

Following are some notes regarding the refactoring operations that are specific to Java enterprise applications:

- For CMP beans, the method rename of a selector or finder automatically updates the EJBQL (Enterprise JavaBeans Query Language) statements and query element in the deployment descriptor. However, the IDE does not allow you to change the method name of a mandatory EJB method such as `ejbCreate()`.

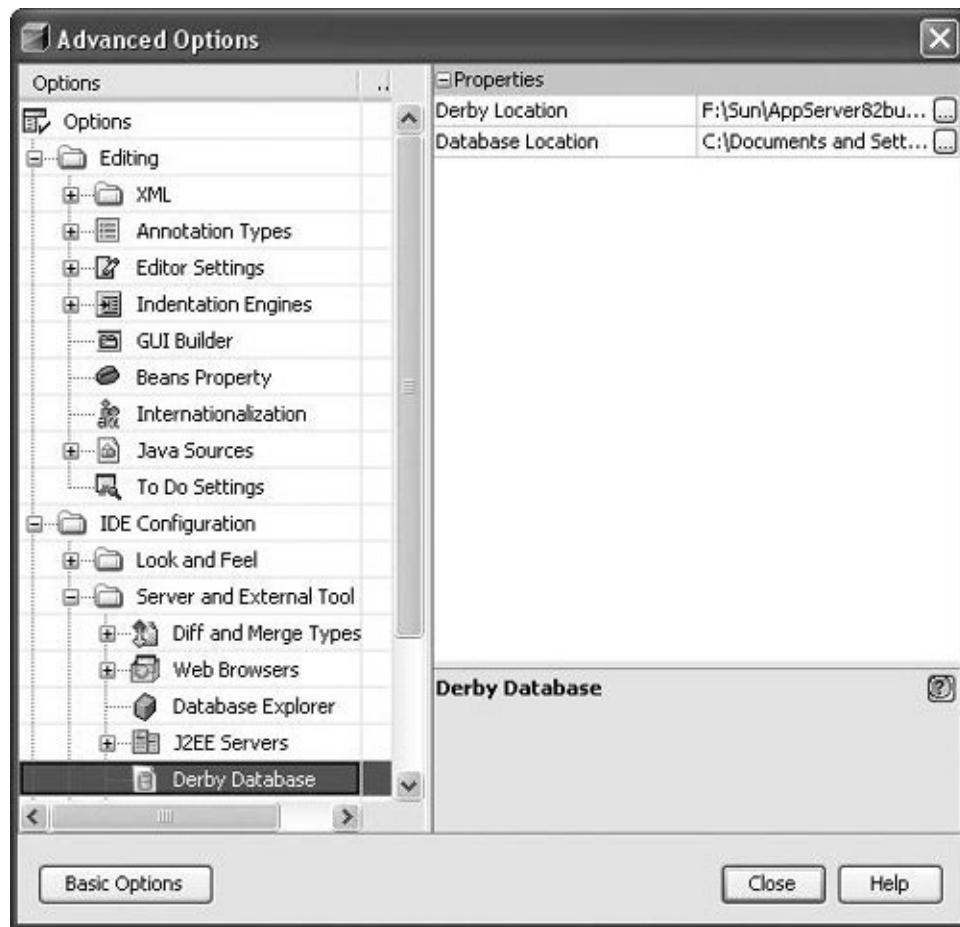
Even in the case where a renamed method was previously overloaded, a new deployment descriptor entry is made for the refactored method name. For instance, updates to deployment descriptors tagged as primary key fields and composite primary key fields as a result of method rename refactoring are supported. Also, a rename of web service endpoint interfaces are handled by NetBeans IDE by updating the appropriate WSDL.

- Even in the case of a CMP bean, a field rename refactoring of a primary key results in field accessors deployment descriptors being updated. In addition, if a relationship field is renamed, the `cmr-field` in the deployment descriptor is updated.
- Encapsulate fields refactoring in an enterprise bean class works the same way as in standard classes, with the exception of primary keys in CMP enterprise beans, which cannot be refactored in NetBeans IDE.

Database Support and Derby Integration

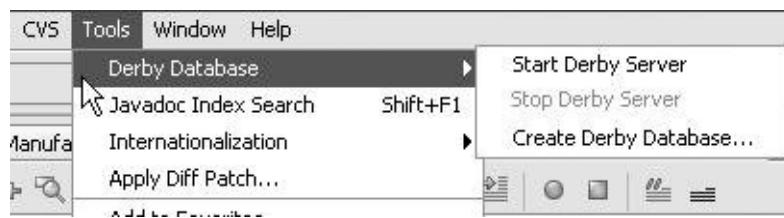
When an instance of Sun Java System Application Server 8.2 is registered within NetBeans IDE, the Derby database is registered as well, and the IDE provides a new set of menu items to start and stop the database or even create a new database. If you are not using the Sun Java System Application Server, it is still possible to manually register the Derby Database via the Tool | Options menu. Navigate in the Advanced Options and select the Derby Database node, as seen in [Figure 13-38](#), where you can enter the Derby installation location and the location where Derby will store the databases.

Figure 13-38. Derby Database Options



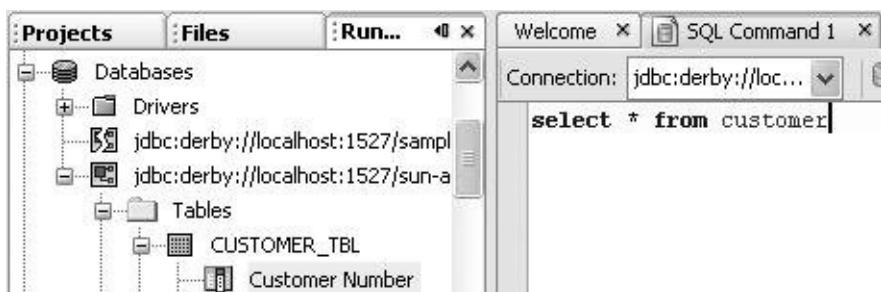
If Derby Database is correctly registered in the IDE, the Derby menu is available as a submenu of the Tools Menu, as seen in [Figure 13-39](#).

Figure 13-39. Derby Database menu



Once a database is registered in the IDE, you can create new databases or explore the structure of existing databases in the Runtime window and execute SQL statements using the SQL command editor (see [Figure 13-40](#)). You can access the SQL editor by right-clicking the node of a database table and choosing Execute Command.

Figure 13-40. SQL Query editor and executor



In NetBeans IDE 5.0, it is not possible to type more than a single line of SQL code. (This limitation will be fixed in the next version of NetBeans IDE.) However, you can use the ij tool that comes with the Derby database to execute whole blocks of SQL statements. The easiest way to work with this tool is to create a wrapper script. For example, on Windows, you can create the following ij.bat file in your Windows directory (so it would automatically be on the path):

```
@echo off
rem -- Run Derby ij tool--
set LIBPATH=D:\db-derby-10.1.2.1-bin\lib
java -classpath
"%LIBPATH%\derbytools.jar;%LIBPATH%\derby.jar;%LIBPA%
org.apache.derby.tools.ij %1
```

Now, from anywhere on the system, you can call `ij` against an SQL script containing a collection of SQL statements for creating multiple tables, or inserting data in your database.



Forgot the password for the sample database? For Derby, try to use `app` and `app` for username and password. For Pointbase database, you might have a better chance with `pbsub` and `pbsub`.

Chapter 14. Developing Java ME Mobile Applications

- [Mobility Primer](#)
- [Setting up Mobility Projects](#)
- [Creating a Project from Scratch](#)
- [Importing a Project](#)
- [Using Mobility File Templates](#)
- [Configuring the Project's Classpath](#)
- [Configuring Your Project for Different Devices](#)
- [Reusing Project Settings and Configurations](#)
- [Structuring Project Dependencies](#)
- [Managing the Distribution JAR File Content](#)
- [Handling Project Resources for Different Configurations](#)
- [Writing Code Specific to a List of Configurations](#)
- [Using the Preprocessor](#)
- [Using Configuration Abilities](#)

- [Creating and Associating an Ability with a Configuration](#)
- [Localizing Applications](#)
- [Using the MIDP Visual Designer](#)
- [Understanding the Flow Designer](#)
- [Understanding the Screen Designer](#)
- [Deploying Your Application Automatically](#)
- [Incrementing the Application's MIDlet-Version Automatically](#)
- [Using Ant in Mobility Projects](#)
- [Using Headless Builds](#)
- [Using the Wireless Connection Tools](#)

THE NETBEANS MOBILITY PACK can simplify many aspects of your MIDP development process. It includes, among other things, the MIDP Visual Designer for managing the flow and content of your application screens, an integrated device fragmentation solution with editor support, and an end-to-end build process that can even include deployment of the application to a remote server.

This chapter covers a range of the Mobility Pack functionality and should give you a better understanding of how you can use many of its features. Projects created for NetBeans Mobility Pack share many qualities with general Java projects. However, you will probably find it useful to read through other chapters in this book as well to learn more about general IDE features concerning project setup, editing, and debugging.

Downloading and Installing the Mobility Pack

The standard NetBeans IDE 5.0 download does not include support for developing mobile applications. You need to download the Mobility Pack separately and use its "add-on" installer to integrate the Mobility Pack functionality with the NetBeans IDE installation you already have on your system.

For NetBeans IDE 5.0, you should be able to find the Mobility Pack installer on the same download page as the IDE installer.

Once you have downloaded the installer, launch it (in the same way that you launched the NetBeans IDE installer) and complete the wizard. The wizard will help you identify the installation of NetBeans IDE to build on.

Mobility Primer

NetBeans Mobility Pack makes it easy for all programmers to start development of Java ME applications quickly, even if you have had only Java SE programming experience.

If you are trying out mobile application programming for the first time, you can just think of Java ME as having a more limited version of the Java SE API and let NetBeans Mobility Pack take care of the foreign Java ME issues. Although there are more differences than just a smaller API, it's not necessary to know all the intricacies before producing a working mobile application.

To help you get started, here are some terms that are used throughout this document:

- **Java ME.** Java Platform, Micro Edition, is the Java platform meant to run on small devices. Currently, these devices are typically cell phones or PDAs, but Java ME is also used on other embedded systems. A configuration, a profile, and optional packages are what compose a Java ME platform.
- **CLDC.** Connected Limited Device Configuration is a Java ME configuration that is currently most often used on mobile phones. It contains a runtime environment and a core API that are appropriate for the limited processor speed and memory size of mobile devices.
- **MIDP.** Mobile Information Device Profile is the set of APIs that provides higher-level functionality required by mobile applications, such as displayable components ("screens") and network communication.

- **MIDlet.** A class required by all MIDP applications. It acts as the interface between the application and the device on which it is running. A MIDlet is similar to a main class in a J2SE project.
- **Preverification.** When building an application that runs with CLDC, all compiled classes must be preverified. Preverification is a process that adds annotations used by the CLDC JVM to a class file's bytecode. The preverification process also ensures that the class contains only code that will run on its CLDC version.
- **Device fragmentation.** Term used for the variations between mobile platforms that prevent a single application from automatically running optimally on all phones. These differences can be physical (screen size, screen color depth, available memory, and so on) or software related (available APIs, CLDC/MIDP version, and so on).
- **Preprocessor.** NetBeans Mobility Pack ships with a preprocessor that is used as part of its device fragmentation solution. The preprocessor is an Ant task that runs before files are compiled. It looks for special Java comment tags within the file and adds or removes line comments based on these tags.
- **Obfuscation.** A process that makes class files difficult to reverse-engineer. This is usually accomplished by, at least, replacing names of packages, classes, methods, and fields with short identifiers. This also has the result of decreasing the size of your application and, therefore, is an important aspect of the mobile application build process.

If you're interested in learning more, countless online and offline resources are available. A good place to start looking is <http://java.sun.com/j2me/>.

Configuration vs. Configuration

One naming conflict that can cause confusion in this text is that between the NetBeans concept of project configuration and the Java ME concept of device configuration. Because this text focuses on covering NetBeans IDE, the unmodified term *configuration* will henceforth refer to project configurations.

Setting up Mobility Projects

After you have installed NetBeans IDE and the Mobility Pack, creating a new project is your next step in developing a mobile application. The IDE does the following upon creation of a Mobility project:

- Creates a source tree optionally containing a simple MIDlet.
- Selects an appropriate emulator platform for your project. Normally, the bundled J2ME Wireless Toolkit platform is selected.
- Sets the project runtime and compile-time classpath.
- Creates a build script that contains commands for running, compiling (including mobile-specific tasks such as preprocessing, preverification, and obfuscation), debugging, and building Javadoc.

Creating a Project from Scratch

The process of creating a project from scratch is nearly identical to how it is done with general Java projects:

1. Choose File | New Project.
2. In the Categories tree, select the Mobile folder.
3. Select Mobile Application or Mobile Class Library.

The Mobile Application template provides an option to generate an example MIDlet automatically. The Mobile Class Library template does not generate any classes for you.

The MIDP 1.0 UI, the MIDP 2.0 UI, and the Bluetooth examples (all available in the Samples | Mobile folder in the New Project wizard) use the Visual Designer.

4. Click Next.
5. Optionally, fill in the following fields in the Name and Location panel of the wizard:

Project Name. The name by which the project is referred to in the IDE's user interface.

Project Location. The location of the project on your system.

The resulting Project Folder field shows the root project directory that contains folders for your sources, build and properties files, compiled classes, and so on.

6. Optionally, deselect the Set As Main Project checkbox if you have another project open that you want associated with the IDE's main project commands (such as Build Main

Project).

7. Optionally, if you selected Mobile Application project, you may deselect the Create Hello MIDlet checkbox to avoid generating an example MIDlet. Otherwise, a small sample MIDlet is created that uses the Visual Designer.
8. Click Next or Finish.
9. Choose the emulator platform you would like your project to use by default. This setting controls your project's classpath as well as which emulator and device are launched when you run your project.

Note that you can change this setting once the project has been created.

10. Click Next or Finish.
11. Optionally, create additional project configurations from templates. Check those that you need. You can find more information on project configurations in Configuring Your Project for Different Devices.
12. Click Finish.

Importing a Project

The Java ME platform supports automatic imports from existing J2ME Wireless Toolkit projects, Sun Java Studio Mobility projects, and projects consisting of stand-alone MIDP sources. When you import projects, the sources remain in their existing locations.



Unlike with general Java projects, only a single source root is supported for Mobility projects. If the project you are importing has multiple source roots, you have to create one project for each source root and then make dependencies between the projects. This is explained further in Structuring Your Projects in [Chapter 3](#).

To create a new NetBeans project from a Sun Java Studio Mobility project or stand-alone sources:

1. Choose File | New Project.
2. In the Categories tree, select the Mobile folder.
3. Select Import Mobility Studio Project or Mobile Project from Existing MIDP Sources, and click Next.
4. In the Imported Sources Location field, enter the folder that contains the default package of your sources (the folder might be called `src`).
5. In the Imported Jad/Manifest Location field, enter the folder that contains the application's JAD file. You can leave this field blank if your project does not have a Java Application Descriptor (JAD) file.

- 6.** Click Next.
- 7.** Optionally, edit the following fields in the Name and Location panel of the wizard:
 - Project Name.** The name by which the project is referred to in the IDE's user interface.
 - Project Location.** The location for the new project.
- 8.** Optionally, deselect the Set As Main Project checkbox if you have another project open that you want associated with the IDE's main project commands (such as Build Main Project).
- 9.** Click Next.
- 10.** Configure the project's default emulator platform.
- 11.** Click Finish.

To create a new NetBeans project from a J2ME Wireless Toolkit project:

- 1.** Choose File | New Project.
- 2.** In the Categories tree, select the Mobile folder.
- 3.** Select Import Wireless Toolkit Project and click Next.
- 4.** On the Specify WTK Project page of the wizard (shown in [Figure 14-1](#)), specify the location of your Wireless Toolkit installation directory that contains the project you would like to install.

Figure 14-1. New Project wizard, Specify WTK Project page

[\[View full size image\]](#)



5. A list of all projects contained in the installation directory is displayed. Select the project you would like to import.
6. Click Next.
7. Fill in the Name and Location page and the Platform Selection page of the wizard as described in steps 7 through 11 in the previous procedure.



Two projects that use the same source root should not be open concurrently within the IDE. Although the IDE takes steps to prevent this situation, the projects will become unstable should it occur.

Physical Structure of Mobile Projects

You can examine the physical structure of your Mobile project using the Files window. This window is located, by default, next to the Projects window or accessible via Window | Files in the main menu.

When you create a Mobile project, the IDE creates the same directories and files as those that are created for general Java projects, except that a separate `test` directory is not created. Tests, if generated, are placed in the same packages as the classes they are testing.

The `build` and `dist` folders, created the first time you build the project, are slightly different due to the more complicated nature of the MIDP build process. The following directories are created under `build`:

- `compiled` folder, which contains all compiled classes.
- `preprocessed` folder, which holds preprocessed versions of your source files. The files contained here are different from the original sources only if you are using configurations.
- `obfuscated` folder, which holds the obfuscated versions of your class files.
- `preverified` folder, which holds the preverified versions of your class files. These are the class files that are packaged in your project's distribution JAR file.

These files are maintained both to increase the speed of the build process and to give you information you can use to isolate where in the build process bugs may have been introduced into

your application.



The NetBeans build process follows Ant standards for management of temporary build classes. This means that only source files with a more recent timestamp than that of the class files in the build directories are automatically rebuilt when the Build command is run. Also, deleting a source file from the project does not automatically delete the associated class files from the build directories. Therefore, it is important to clean and rebuild the project after a source file has been removed from the project.

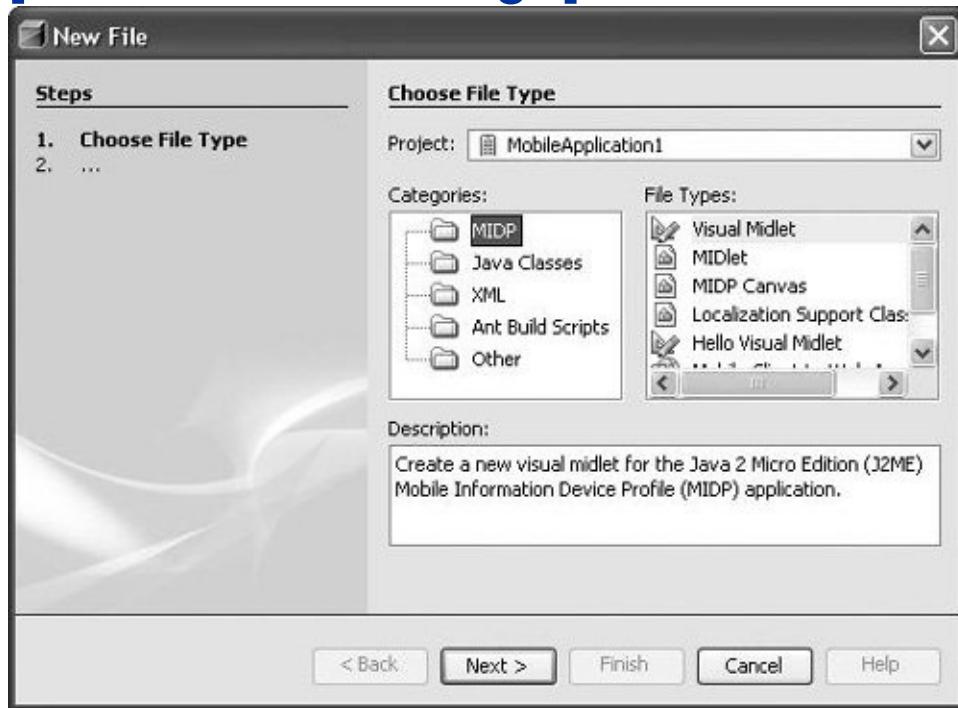
The `dist` folder contains your project's distribution JAR file and JAD file.

Using Mobility File Templates

The process of creating files in Mobility projects is identical to that of creating files in general Java projects. Templates for creating MIDP classes can be found under the top-level MIDP category in the New File wizard, as shown in [Figure 14-2](#). You can open the New File wizard by choosing File | New File.

Figure 14-2. New File wizard, MIDP templates

[[View full size image](#)]



MIDP Canvas

To create a MIPDCanvas:

1. Select MIDP Canvas in the File Types list and click Next.
2. Optionally, modify the following fields in the Name and Location panel of the wizard:

MIDP Class Name. The name for the new class.

Package. The package in which the new class is located.

The resulting Created File field shows the full path location of your new class.

3. Click Finish.

MIDlet

The wizard for creating MIDlet file types differs slightly from that of other file templates in that it also collects information that is used in the application's JAD file.

1. Select one of the MIDlet file types in the File Types list and click Next.
2. Optionally, modify the following fields in the Name and Location panel of the wizard:

MIDlet Name. This is the value that is shown in the list of MIDlets displayed when starting the application. It can be different from the MIDlet's class name.

MIDP Class Name. The name for the new MIDlet class.

MIDlet Icon. The location of this MIDlet's icon. The combo box contains all `.png` files located in your project.

Package. The package in which the new MIDlet is located.

The resulting Created File field shows the full path location of your new class.

3. Click Finish.



All MIDlets that you add to the project are automatically added to the project's application descriptor.

Localization and Support Class

This is an advanced template that uses configurations; therefore, it is described in Localizing Applications later in this chapter.

Configuring the Project's Classpath

Like most project settings, the classpath used to compile Mobility projects is managed in the Project Properties dialog box. Two panels in particular control the classpath: the Libraries & Resources panel and the Platform panel. In addition to compilation, code completion is controlled by these settings.

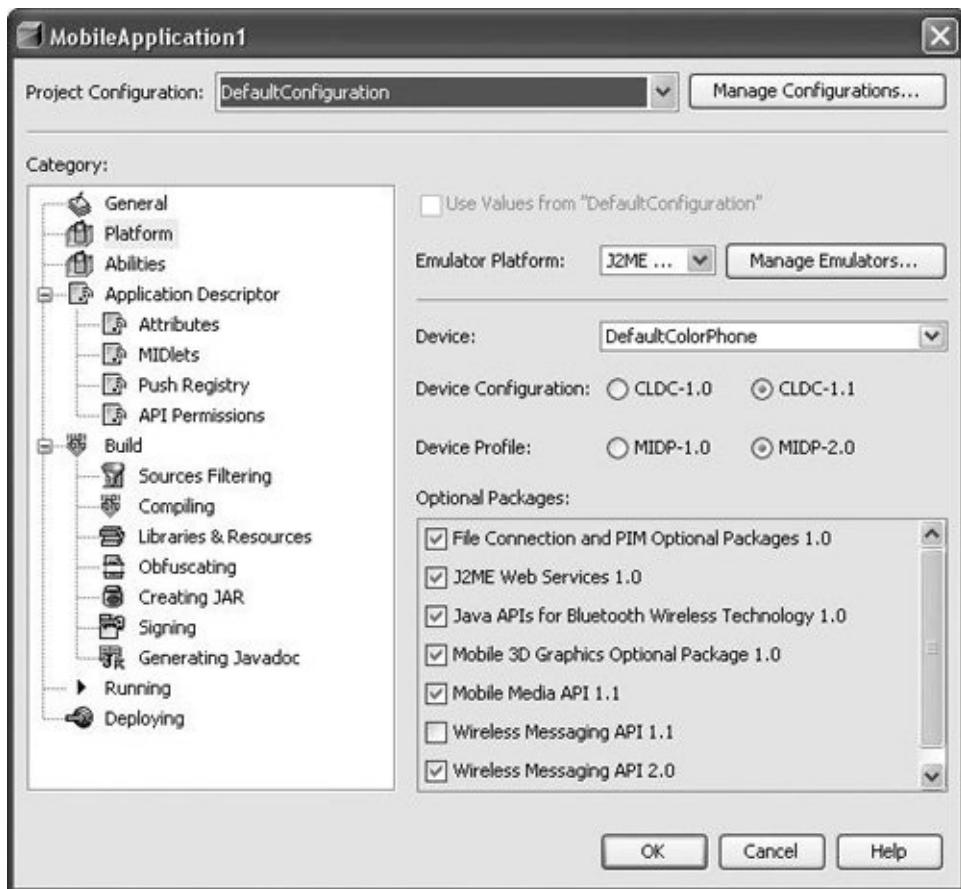
Changing the Project's Emulator Platform

The selected emulator platform is used to determine which vendor-supplied libraries are to be used for project compilation. By default, your project is configured to use the latest version of the J2ME Wireless Toolkit. You can modify this using the Platform panel of the Project Properties dialog box:

1. Right-click your project in the Projects window and choose Properties.
2. Select the Platform node in the Project Properties dialog box (as shown in [Figure 14-3](#)).

Figure 14-3. Mobile Application Project Properties dialog box, Platform panel

[\[View full size image\]](#)



3. Optionally, modify any of the following fields to change which emulator libraries are used to build and run the project:

Emulator Platform. The installed emulator platform you would like to use. This controls which values are available for all following fields.

Device. Selects which device skin to use when launching the application.

Device Configuration. Determines the device configuration. This setting affects your classpath.

Device Profile. Determines the device profile. This setting affects your classpath.

Optional Packages. Select or deselect any number of the

optional packages. Again, this setting affects your project's classpath.



Sometimes, one device profile or device configuration is disabled when a given emulator platform is selected. This is a result of that emulator platform's response to a Unified Emulator Interface (UEI) query regarding its profile and configuration abilities.

You can still develop an application for the disabled configuration or profile using the selected platform (for example, developing a MIDP-1.0 profile with the WTK2.0 emulator platform). First, ensure that you do not use any invalid code for the desired profile or configuration; then add a MicroEdition-Configuration or MicroEdition-Profile attribute to the project's Application Descriptor. These attributes are managed in the Project Properties dialog box's Attributes panel.

Although NetBeans Mobility Pack ships only with the J2ME Wireless Toolkit, it is also possible to install additional third-party-vendor emulators into the IDE. This is done using the standard Platform Manager, which can be accessed in the standard way from the Tools | Java Platform Manager main menu option or from within the Project Properties dialog box.

- 1.** Right-click your project in the Projects window and choose Properties.
- 2.** In the Project Properties dialog box, select the Platform node.
- 3.** Click Manage Emulators. This opens the Java Platform Manager.
- 4.** Click Add Platform.

- 5.** Select the Java Micro Edition Platform Emulator radio button and click Next.
- 6.** After the IDE finishes its scan for installed emulators, select those emulators that you would like to add.
- 7.** If the IDE has not detected an emulator you are looking for, click Find More Java ME Platform Folders and navigate to the location of the emulator you would like to install.

Note that automatic detection is available only on Windows systems (through the Windows registry). On UNIX-based systems, you must always specify the path to search.

- 8.** Click Next and wait for the IDE to retrieve detailed information about the SDKs. If there is at least one valid emulator platform, the Finish button is enabled.

Note that only UEI-compliant emulators are detected. See Installing Nonstandard Emulator Platforms later in this chapter to learn how you can install non-UEI-compliant emulators into the IDE.

It is possible to add the same emulator twice; however, a different name has to be specified.

- 9.** Optionally, click the Javadocs tab or Sources tab to specify the locations of emulator sources and documentation.
- 10.** Click Finish.

Installing Nonstandard Emulator Platforms

This section describes the process by which you can use an emulator platform installed on your system that is not automatically recognized by the Java Platform Manager. Platforms are not automatically detected if they do not comply

with the Unified Emulator Interface (UEI) specification. This specification defines queries that allow external tools to determine the capabilities of the emulator (for example, supported MIDP and CLDC versions). Although most modern emulators are written in accordance with this standard, it can still be useful to use the older, non-UEI-compliant emulators.



Emulators that are not UEI-compliant can sometimes be installed in the J2ME Wireless Toolkit as a device. Once installed in the J2ME Wireless Toolkit, the device appears in the Devices combo box of the Platforms panel when that platform is selected.

Although the Nokia 7210 can be installed in this manner, it is still used here as an example illustrating how platform descriptor files can be manually created or edited.

An emulator platform that is not UEI compliant can be added using Tools | Java Platform Manager:

1. Click Add Platform.
2. Choose Custom Java Micro Edition Platform Emulator. Click Next.
3. The information you must enter is:
 - **Platform Home.** The path to the directory where the emulator platform is installed. You can enter a path, or use the Browse button to navigate to the directory.
 - **Platform Name.** A name for the emulator platform.
 - **Device Name.** A name for the specific device the platform emulates.

- **Preverify Command.** The command-line syntax that invokes the emulator to preverify a MIDlet.
- **Execution Command.** The command-line syntax that invokes the emulator to execute a MIDlet.
- **Debugger Command.** The command-line syntax that invokes the emulator to debug a MIDlet.

To see descriptions of the command-line syntax parameters, click in the appropriate field.

4. Click Finish.

Adding JAR Files and Folders to the Classpath

If your project depends on additional APIs or classes that aren't part of the selected emulator platform, you can add them to your project's source path manually. Unlike classpath items for general Java projects, all classes added by this panel are packaged in the project's distribution JAR file. Care must be taken to ensure that all included classes pass preverification.

1. Right-click your project in the Projects window and choose Properties.
2. Select the Libraries & Resources node in the Project Properties dialog box.
3. Click Add Jar/Zip or Add Folder. In the file browser, choose the file or folder containing the classes you would like to include.
4. Click OK to close the Project Properties dialog box.



If you want to use a library solely for compilation without packaging it with your application, you should make that

library an optional API of one of your emulator platforms. See Changing the Project's Emulator Platform earlier in this chapter.

Also, it is important to remember that the preverification process can cause the build process to fail even for projects that compile without any problems. All classes in the Libraries & Resources panel must preverify correctly using the selected emulator platform's preverify command for the build process to succeed.

Debugging Your Project

Debugging Mobility projects can be accomplished in the same way as in general Java projects; right-click your project in the Projects node and choose the Debug Project command. The emulator opens, and after you select the MIDlet to run on the phone, the program execution stops at any set breakpoints.

Nevertheless, there are some things you should be aware of:

- Not all third-party emulators support debugging commands equally well. If you encounter problems like breakpoints being skipped or step-into commands not working, it may be the result of a faulty runtime environment.
- Always remember to disable obfuscation when debugging.
- Method invocation is not supported by the CLDC JVMs, which can affect your debugging experience. For example, adding something like

`myObject.toString()` to the Watch panel of the debugger will not return a correct result.

- Debugging a mobile application is often slower than debugging J2SE applications.

Configuring Your Project for Different Devices

One of the most problematic aspects of development for Mobility projects is the issue known as device fragmentation. Writing a single application that will run on disparate platforms can be challenging. Differences in physical aspects of the platform, such as screen size and free memory, as well as software issues such as available APIs, are some of the reasons why building a single application for multiple devices can require variations in both code and project settings. Fortunately, NetBeans Mobility Pack has built-in support for this issue that simplifies the problem.

The solution is based on the concept of project configurations. You may have noticed that one major difference in the Project Properties dialog box between Mobility and general Java projects is the combo box located at the top of the dialog box labeled Project Configurations. This combo box is the starting point for using the device fragmentation solution. It contains one element for each configuration in your project, as well as an item that creates new configurations.

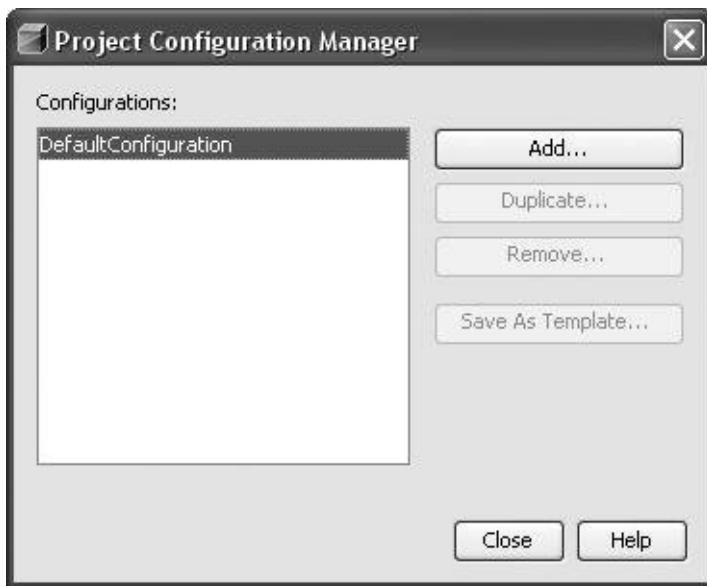
In general, you should create one configuration for each distribution JAR file you plan on building for your project. So, for example, if you are planning to support three different screen sizes, using two sets of vendor-specific APIs, you would create six configurations.

Creating a Configuration

1. Right-click your project in the Projects window and choose Properties.
2. Click the Manage Configurations button. This opens the

Project Configuration Manager dialog box, shown in [Figure 14-4](#).

Figure 14-4. Project Configuration Manager



3. Click Add to open the Add Project Configuration dialog box.
4. Enter a name for your new configuration and click OK.

This name should be an identifier that describes the distribution that this configuration is used for. For example, some descriptive configuration names might be NokiaSeries40, SmallScreen, or HighResNoSizeLimit. The only restrictions are that the identifiers must be valid Java identifiers and that they must not conflict with existing configuration names within the project.

5. Click Add again if you'd like to create another configuration. Otherwise, click Close to close the dialog box.



When creating a new configuration you may use an existing configuration template. The IDE automatically creates a template for every emulator device registered

through Java Platform Manager. For instance, the QwertyDevice_template is pre-configured to use the QwertyDevice of Wireless Toolkit emulator.

Customizing Configurations

Once you have created your configurations, you may make customizations within the Project Properties dialog box that pertain to only one configuration. There are two basic principles of the configurations within the Project Properties dialog. First, by default, all configurations take the values set for the default configuration. Second, user-added configurations can override the default settings for any panel (other than the General panel) if you deselect the checkbox labeled Use Values from "Default Configuration." Once this checkbox has been unchecked, that configuration uses its own settings for the panel.

The following is an example of changing the default configuration and customizing a single configuration:

- 1.** Right-click your project in the Projects window and choose Properties.
- 2.** Make sure DefaultConfiguration is selected in the Project Configurations drop-down box. In the Emulator Platform drop-down box, select J2ME Wireless Toolkit 2.2, and in the Device drop-down box of the Platform panel, select DefaultGreyPhone.
- 3.** In the Project Configurations combo box at the very top of the Project Properties dialog box, select one of the

configurations you created in the previous portion of this section.

4. Select the Platform panel. All controls in this panel should be disabled. This is because the configuration is currently drawing its values from DefaultConfiguration.
5. Deselect the Use Values from "DefaultConfiguration" checkbox. All controls on the panel are now enabled, as they are no longer taking their values from the default configuration.
6. Change the Device combo value to DefaultColorPhone.

You have now configured the DefaultConfiguration to use the DefaultGreyPhone device and your user-added configuration to use the DefaultColorPhone.

Setting the Active Configuration for Your Project

At any given time, only one configuration is active for your Mobility project. Project settings from the Project Properties dialog box are based on this active configuration.

You can view or set the active configuration in several ways:

- Right-click the project node in the Projects window and hover over the Set Active Project Configuration menu option. The active configuration is marked by a bullet. You can click any configuration in this list to activate the selected configuration.
- Right-click the project node in the Projects window and choose Properties. Changing the selected configuration in the Project Configuration combo box activates the newly selected configuration.
- Use the combo box in the Build toolbar. The selected item in the combo box is the active configuration.

Physical Structure of Mobility Projects with Configurations

Once a project containing multiple configurations has been built, the `build` folder will contain the build directories for the default configuration, and the `dist` directory will contain its distribution JAD file and JAR file. Additionally, the `build` and `dist` directories will contain one subdirectory for each configuration in the project. These subdirectories will hold the build and distribution files for the like-named configuration.

Reusing Project Settings and Configurations

Creating a project with several different configurations and settings within those configurations can be a time-consuming process. If you often want to support the same set of deployment platforms for each application you create, it is useful to reuse those configurations and settings. The NetBeans IDE has several features that address this need.

Duplicating Project Settings

If you have configured an entire project that has settings you would like to reuse on a new project, you can do so using the Copy Project command.

To reuse a project's settings in another project:

1. Right-click your project in Projects window and choose Copy Project.
2. Optionally, update the Project Name and Project Location fields.
3. Click Copy.

All files in the original project's source root are copied to the `src` folder contained in the new project. This occurs even when the original project has an external source root.

Your newly created project has all of the settings and configurations from the original project. These may include some settings that are no longer appropriate (for example, JAD file and JAR filename, filtering options, and some JAD file attributes).

Using Project Configuration Templates

Alternatively, you might want just to reuse the settings from one or two configurations. You can do this with configuration templates, which allow you to create new configurations based on the saved settings of other useful configurations. Only the settings from panels that are not taking their values from the default configuration are stored in templates. They are very easy to create:

- 1.** Right-click your project in Projects window and choose Properties.
- 2.** Click Manage Configurations.
- 3.** Select the configuration that contains the reusable settings and click Save As Template.
- 4.** In the Save Project As Configuration Template dialog box, enter a name for the template and click Save.

Templates are uniquely stored in the IDE, so choose a descriptive name.

Now that you have saved this configuration template, it is possible to create new configurations based on it:

- 1.** Right-click your project in the Projects window and choose Properties.
- 2.** Select Add Configuration from the Project Configurations combo box.
- 3.** Select the template you would like to use from the Use Configuration Template combo box.
- 4.** Modify the name of the configuration in the New Configuration Name text box if desired. Click OK.

This template-based configuration has the same settings as the template for each panel of the template that was not using the values from the default configuration.



Once a template is created, it does not remain synchronized with its original configuration. Also, it is not possible to update existing templates directly. Instead, create a new configuration based on the template, update it, delete the original template, and then save the new configuration as a new template with the original template name.

Structuring Project Dependencies

Unlike other NetBeans project types, Mobility projects are based on the concept of a single source root that results in a single distribution JAR file. It is possible to simulate multiple source roots by setting up one project for each source root and then using project dependencies. It may be helpful to think of the source root containing the MIDlet as the application project, while all other source roots should be considered library projects.



There are no programmatic differences between Library and Application projects other than the option to include a MIDlet when creating an Application project. The differences between project types are purely conceptual in nature.

To structure your project dependencies:

- 1.** Right-click your application project in the Projects window and choose Properties.
- 2.** Select the Libraries & Resources panel in the Project Properties dialog box.
- 3.** Click Add Project.
- 4.** Navigate to the project root of a Library project and ensure that the correct Project JAR file is selected in the Select Project JAR Files dialog box.
- 5.** Click Add Selected Project JAR Files.

6. Repeat for each Library project.

If there are interdependencies among the Library projects, these must be set up in the same manner. Note that dependencies may not be circular; if project A depends on project B, project B cannot depend on project A (or any other project that depends on project A).

Using Dependencies with Configurations

If your Library project and Application project make use of project configurations, it is important to take care when setting up the project dependencies. Although it is tempting to assume that the Application project will automatically depend on the correct version of the Library project (based on configuration name matching), this is not the case.

Assume, for example, you are working with an Application project that has configurations named SmallScreen and LargeScreen, and a Library project that has two configurations of the same name. To set up the dependency between the two projects:

- 1.** Right-click your Application project in the Projects window and choose Properties.
- 2.** Select the Libraries & Resources panel in the Project Properties dialog box.
- 3.** Select the SmallScreen configuration from the Project Configuration combo box and then uncheck the Use Values from "DefaultConfiguration" checkbox.
- 4.** Click Add Project.
- 5.** Navigate to the project root of the Library project and Select the project JAR file that is prefaced with

/dist/SmallScreen in the Select Project JAR Files dialog box.

- 6.** Click Add Selected Project JAR Files.
- 7.** Select the LargeScreen configuration from the Project Configuration combo box and then repeat these steps (but selecting the */dist/LargeScreen* distribution JAR file when appropriate).
- 8.** Click OK.

Managing the Distribution JAR File Content

By default, all class files resulting from compilation, and all non-source files contained under the project's source root folder, are placed in the distribution JAR file. All classes and resources specified in the Libraries & Resources panel are also placed in this distribution JAR file.

Your project may contain files under its source root that should not be distributed. These files can be filtered in the following manner:

1. Right-click your project in the Projects window and choose Properties.
2. Select the Sources Filtering panel.
3. Select the default filters provided that are appropriate for your project. Hovering your mouse over the checkbox text displays which regular expression is used to filter files from the distribution JAR file.
4. In the Select Application Packages/Files field, make any necessary adjustments to the files to be distributed. Selected files are distributed.

Deselect a node, and notice that all children nodes are also deselected. If any node is deselected, all of its ancestor nodes are grayed out to indicate a partial selection.

5. Click OK to close the Project Properties dialog box.

The next time you build the project, the selected files and resources will not be added to the distribution JAR file.

Handling Project Resources for Different Configurations

One of the most common causes of device fragmentation is different screen sizes on mobile devices. Disparate screen sizes require different resources, and usually, it is not advisable to include unused resources in the distribution JAR file.

Imagine the scenario in which your application has a fixed-size splash-screen image, and you want to include only the correct-size image with each configuration. There are two different methodologies that you can use to manage this problem using NetBeans Mobility Pack. Which method you should use depends on whether you are working with an existing project structure or have the freedom to handle the problem in any way that suits you.

Using Different Resource Locations

One technique is to create size-specific resource JAR files or folders that contain identically named resources. For example, you might have a directory structure like this:

```
/res/small/splashScreen.png  
/res/medium/splashScreen.png  
/res/large/splashScreen.png
```

Your source code would read simply:

```
Image splashScreen = Image.createImage("splashScreen.p
```

Then you can modify the Libraries & Resource panel such that each configuration imports only the correct resource file for the screen size on the devices to which it will be deployed.

Using Configuration-Specific Code Blocks and the Filtering Panel

An alternative solution is to have distinctly named resources all contained under the project's source root. So you might have a directory structure something like this:

```
/src/res/splashScreenSmall.png  
/src/res/splashScreenMedium.png  
/src/res/splashScreenLarge.png
```

Then you would need three configuration-specific code blocks in your source code for the `create` statement, meaning that your source would look like the code in [Figure 14-5](#).

Figure 14-5. Example of configuration-specific code blocks in the Source Editor

[\[View full size image\]](#)

```
///#ifdef smallScreen  
    splashScreen=Image.createImage("/res/splashScreenSmall.png");  
///#elifdef mediumScreen  
    splashScreen=Image.createImage("/res/splashScreenMedium.png");  
///#else  
    splashScreen=Image.createImage("/res/splashScreenLarge.png");  
///#endif
```

Finally, the Filtering Sources panel would be modified for each configuration such that only the used resource would be selected in each.

Both of these solutions result in similar-size application JAR files that include only the used resources. Therefore, it is up to you which technique is best suited to your project structures.

Writing Code Specific to a List of Configurations

Perhaps one of the most uniquely challenging aspects of handling the device fragmentation problem has been managing differences in code between distributions of an application. The NetBeans approach to the problem has been to integrate a preprocessor into the build process that activates or deactivates sections of code based on which configuration is active at compile time.

The NetBeans preprocessor is a low-impact tool that uses Java comments both to define code sections and to activate and deactivate these sections. As such, all files before and after preprocessing can be valid, syntactically correct Java source. This also ensures that the preprocessed files integrate seamlessly with the debugger.



Source files are always saved to the hard drive as though the default configuration were active. This eliminates the VCS conflicts that could otherwise occur due to the local version's having a different active configuration than the VCS version.

A series of context-menu commands exist to assist you in adding these special comments to your source files. These commands are described in the following sections.

Duplicating a Code Block

Sometimes, there is a section of code that must be defined

differently for some configurations and that must exist even for configurations that don't explicitly define a code block in the section. An example of this would be if your class definition is fragmented. For example, all of your configurations that will be deployed to Nokia devices might define your Canvas object like this

```
public class MIDPCanvas extends com.nokia.mid.ui.FullC
```

while the rest of your configurations would just extend `Canvas`. After adding the appropriate code blocks, you would be left with the code shown in [Figure 14-6](#).

Figure 14-6. Duplicated code blocks in the Source Editor

```
public class MIDPCanvas
    //#ifdef Nokia
    //##      extends com.nokia.mid.ui.FullCanvas {
    //##else
    extends Canvas {
    //#endif
}
```

These types of blocks can be created in one step using the Create If / Else Block context-menu command:

1. Highlight the code section that you would like to be duplicated.
2. Right-click in the Source Editor and choose Preprocessor Blocks | Create If / Else Block.
3. Type in a configuration name or other statement right after

the `#if` directive. The IDE offers a list of all available configurations and abilities.

Two code blocks are created, each containing the highlighted code.



Only full lines of text may be associated with a configuration. If you would like to associate only part of a line with the configuration, first break the line into smaller sections and associate the sections appropriately.

How to Interpret Code Block Visualization

The visualization of code blocks within the IDE is that of color highlighted sections. This visualization can help you quickly determine some information about code sections:

- Sections that are active for the currently selected configuration are pink. Also, code contained within the section is uncommented.
- Inactive sections are gray. Each line in the code section is prepended with a specially formatted line comment `(//#)`.



The state of the special comments within the IDE's Source Editor is unimportant, as the preprocessor runs before any compilation command and places the lines of each code block in the correct comment state.

If the special comments within the code blocks fall out of sync with the current state of the code block in the IDE, you can correct them by right-clicking them in the editor and choosing Preprocessor Block | Re-comment or by

simply switching configurations. The preprocessor automatically fixes the errors.

Using the Preprocessor

The following text describes the syntax of the preprocessor. It should be consulted for any questions regarding the syntax of the directives and their arguments.

General

The preprocessor uses CPP-like syntax but is still a commenting preprocessor to fulfill the requirements set forth by Java editor integration and the Java language itself (e.g., you are not allowed to change the number of lines in the source file). The directives are specified by creating a commented-out line that starts with the `//#` character sequence immediately followed by the directive, for example `//#ifdef`.

The preprocessor blocks must be well-formed, meaning that when a block is started with one of the `//#if` directives, it must be closed with an `//#endif` directive. Blocks can also be nested, which means that inside a `if/elif/else/endif` block there can be an arbitrary number of additional `if/elif/else/endif` blocks. For example:

```
//#if mmedia
  //#if nokia
    //#if s60_ver=="1.0"
      import com.nokia.mmapi.v1
    //#elif s60_ver=="2.0"
      import com.nokia.mmapi.v2
    //#else
      import com.nokia.mmapi.def
    //#endif
  //#else
    import javax.microedition.mmapi
```

```
//#endif  
//#endif
```

Directives

The following directives are available in the given syntax and function as follows:

- **#ifdef identifier**. The identifier represents a variable of any type (Boolean, String, Integer) and checks whether it is defined. If it is, then the result is true and the code that follows will be left alone. Any nested blocks will be processed as well. Otherwise, code that follows will be commented out and any nested blocks will not be evaluated. Must be closed with **endif**. This commenting behavior is same for all the other directives.
- **#ifndef identifier**. Works like **ifdef** but returns **TRue** only if the identifier is not defined. Must be closed with **endif**.
- **#elifdef identifier**. Works like a standard else if statement but automatically checks whether or not the identifier is defined. Can only complement inside blocks started by **ifdef/ifndef**.
- **#elifndef identifier**. Works like a standard else if statement but automatically checks whether the identifier is not defined. Can only complement inside blocks started by **ifdef/ifndef**.
- **#ifexpression**. Evaluates an expression passed to it and fires the appropriate action. How expressions are evaluated is

explained below. Must be closed with `endif`.

- `#elif` expression. Works like a standard `else if` statement and can complement only in blocks started by an `if` statement. Will preprocess the code that follows based on the result of the expression.
- `#else`. Works like a standard `else` and will only preprocess the code that follows when none of the previous conditions in the defining block are true. Complements inside any block started with `if/ifdef/ifndef` directive.
- `#endif`. This directive must be used to close any block started with `if/ifdef/ifndef`.
- `#conditionexpression`. Must be on the first line in a file. This directive determines if the file should be included in the build based on the result of the expression.
- `#debug level`. Determines if the line following the directive should be commented out or not based on the debug level set in project's compile properties. If the level is omitted and the debug level is not set to off in the project properties, it will automatically debug the line in question. Used for debugging purposes in conjunction with `System.out.println`, for example. Can be nested.
- `#mdebug level`. Behaves same as `#debug` but will uncomment/comment a whole block of lines following the line it is on until it reaches `#enddebug`. Used for debugging purposes in conjunction with `System.out.println`, for example. Can be nested. If `mdebug` block partially intersects `if/ifdef/ifndef` block (e.g., `enddebug` is outside a closed `if` block in which `mdebug` is called), the preprocessor will generate errors.

- `#enddebug`. Must terminate `#mdebug block`.
- `#define identifier` or `identifier=value` or `identifier value`. Adds temporary abilities/variables to the preprocessor memory programatically on the fly. Nested block insensitive. Global variables defined in the configuration customizer override these temporary variables.
- `#undefineidentifier`. Removes temporary abilities/variables from the memory. Can also be used to remove global variables defined in the configuration customizer from the preprocessor memory but will not remove them from the list of project or configuration variables.

Variables

The preprocessor supports three types of variables: Strings, Integers, and Booleans. The variable names must start with characters that are the same as the start characters of valid Java identifiers. After the first character, you can use the characters period (.), slash (/), and backslash(\), which are normally not allowed in Java identifiers. This is to make migration easier for developers with existing Antenna or J2ME Polish sources that happen to use these characters in some variable and symbol names.

Most common comparisons (such as `<=`, `<`, `>=`, `>`, and `==`) are supported between the variables. Comparisons should not be done on different variable types and a warning will occur but the expression will get evaluated according to the behavior described in the Variable Comparison section. Boolean operations such as `&&`, `||`, `!`, and `^` can also be performed on all symbol types.

Strings can contain just about anything but must be enclosed in

quote marks (""). Definition checks can be made on any variable and, unlike with J2ME Polish, no additional keywords are needed. The preprocessor checks for the J2ME Polish `:defined` construct and removes it from the variable name before checking for the definition. For example, let string `nokia_model` be "N60", then `//#ifdef nokia_model` or `//#if nokia_model` will yield true because it is defined and is evaluated as such. On the other hand, `//#if nokia_model=="7610"` will yield `false` because `nokia_model` is not "7610".

Here is what it would look like in J2ME Polish:

```
//#define nokia_model="N60"
//#ifdef nokia_model:defined
System.out.println("Nokia");
//#else
System.out.println("Other");
//#endif
```

Here is what it would look like in NetBeans IDE:

```
//#define nokia_model="N60"
//#ifdef nokia_model
System.out.println("Nokia");
//#else
System.out.println("Other");
//#endif
```

Integers are numbers and behave the same as strings when it comes to Boolean operations. However, they are compared as true integers and can be used for such tasks as preprocessing code where various screen resolutions require different images that are optimized for different resolutions. For example, `//#if`

`screen_width>100 && screen_height>120` can specify a code block that will import images only for devices with screens larger than 100x120.

Booleans are basically empty variables. Better yet, think of them as symbols. If the variable is an empty variable (e.g., configuration name), then it is considered a Boolean with value of `TRue`. If the variable is not defined anywhere in preprocessor scope (as an ability or configuration), it is considered a Boolean with a value of `false`.

Operators

There are three types of operators with different priorities.

! (the "not" Operator)

The exclamation point (!) operator (the "not" operator) has the highest priority and can be used on **variables** and **expressions** (e.g., !<identifier> or !<expression>). For example, `//#if !nokia` will check whether nokia is not defined. `// #if !(screen_width>100 && screen_height>120)` will check if screen size is smaller than 100x120. The expression must be a valid one enclosed in parentheses as shown. Since the ! operator has the highest priority, expressions such as `// #if !screen_size=="100x200"` are illegal and will yield syntax errors because a Boolean result cannot be compared to a string.

Comparison Operators

The comparison operators have the second highest priority and perform typical comparison operations. They can compare strings lexically and integers mathematically. Cross-type

comparisons are supported and behave as defined in [Table 14-1](#). They can be used only in expressions and should compare *two variables*, not symbols.

Table 14-1. Variable Comparisons

LeftSide	RightSide	==	!=	>	<	>=	<=	@
undef	undef	warn	warn	warn	warn	warn	warn	warn
undef	def	warn	warn	warn	warn	warn	warn	warn
undef	string	warn	warn	warn	warn	warn	warn	warn
undef	integer	warn	warn	warn	warn	warn	warn	warn
def	undef	warn	warn	warn	warn	warn	warn	warn
def	def	warn	warn	warn	warn	warn	warn	warn
def	string	warn	warn	warn	warn	warn	warn	warn
def	integer	warn	warn	warn	warn	warn	warn	warn
integer	undef	warn	warn	warn	warn	warn	warn	warn
integer	def	warn	warn	warn	warn	warn	warn	warn
integer	integer	math	math	math	math	math	math	warn
integer	string	lex	lex	lex	lex	lex	lex	math
string	undef	warn	warn	warn	warn	warn	warn	warn
string	def	warn	warn	warn	warn	warn	warn	warn
string	integer	lex	lex	lex	lex	lex	lex	warn

string string lex lex lex lex lex math

Key: **def**true, **undef**false, **warn**warning, **lex**lexical comparison,
mathmathematical comparison

There is also a special comparison operator that performs a "subset" relationship operation. This operator is denoted by the @ token, and both left and right arguments should be strings that represent two sets of tokens delimited by specific delimiters. It first tokenizes both the left and right string arguments into sets and then determines if the set from the left argument is a subset of the set from the right argument. The valid word delimiters are <whitespace>, ', ' and ';' and they can be mixed arbitrarily within each argument. It behaves just as in the following four examples:

```
"gif" @ "gif86, jpeg, gifaboo" = false
"gif" @ "gif gif86 jpeg" = true
"1 2 4;7,8" @ "0,1,2,3,4,5,6,7,8,9" = true
"3 5 7 11 13" @ "0,1,2,3,4,5,6,7,8,9" = false
```

Even though the variables should be of the same type when comparing, the preprocessor will not fail if you compare variables of different types. Instead, you will be presented with a warning (in the form of an annotation in the Source Editor or a warning in the task's output) and an evaluation of the expression. When the left and right sides of the comparison operation are of different types, both will be considered strings and compared lexically with the exception of the @ operation, where the "subset" relationship operation will still be performed. (As such, you can check if a certain integer is in a particular set). If one of the variables is not defined or is defined as a Boolean symbol, it will be considered an empty string (""). If

one of the variables is an integer, it will also be converted to a string and lexical comparison will take place. You should avoid comparing variables of different types, but doing so by accident will not break the build process.

Boolean Operators

The Boolean operators have the lowest priority relative to the rest of the operators, but they do have different priorities among each other, just like in the Java language. They perform typical logical operations such as `&&`, `||`, and `^` on Booleans; provide expression results; and check for variable definitions and then treat those as Booleans as well. The `&&` operator has the highest priority of all three Boolean operators, while `^` and `||` have lower priorities, respectively. For example, if we preprocess the following example with active configuration being Series40:

```
//#ifdef Series60
    //#define tmpNokia3650
    //#define tmpNokia3660
    //#define tmpSiemensSX1
    //#define tmpScreenWidth=176
    //#define tmpScreenHeight=208
//#elifdef Series40
    //#define tmpNokia3220
    //#define tmpNokia3210
    //#define tmpScreenWidth=80
    //#define tmpScreenHeight=100
//#elifdef Series20
    //#define tmpNokia8320
//#else
    //#define default
#endif
#ifndef tmpScreenWidth==176 && tmpScreenHeight==208 || tmpNokia3660
```

```

        System.out.println("Series 60 configuration active
//#elif tmpScreenWidth==176 || tmpScreenWidth==80 && ti
|| tmpNokia8320
        System.out.println("One of the Nokia configuration
//#else
        System.out.println("Default configuration active!")
//#endif

```

The result is:

```

//#if tmpScreenWidth==176 && tmpScreenHeight==208 ||
tmpNokia3650 && tmpNokia3660
    //# System.out.println("Series 60 configuration ac
//#elif tmpScreenWidth==176 || tmpScreenWidth==80 && ti
|| tmpNokia8320
    System.out.println("One of the Nokia configuration
//#else
    //# System.out.println("Default configuration acti
//#endif

```

Expressions

Expressions are evaluated according to the following Backus-Naur Form (BNF) grammar:

```

<expression> ::= <term> {<boolop> <term>} }
<term> ::= <factor> {<compop> <factor>} }
<factor> ::= <notop> <factor> | value <expression>
<boolop> ::= && | || | ^
<compop> ::= > | >= | < | <= | == | != | @@
<notop> ::= !

```

Assuming that the variables in [Table 14-2](#) are in the scope of the current configuration, the expressions in [Table 14-3](#) are examples of what is legal and is not.

Table 14-2. Example Variables

VariableName	VariableType	Value
nokia	Boolean null	null
screen_width	Integer	100
screen_height	Integer	160
symbVer	String	v7.0
mmapi	Boolean	null

Table 14-3. Example Expressions and Results

Expression	Yields
<code>!nokia && mmapi</code>	true
<code>symbVer=="v7.0" (screen_width>=100 true</code> <code>&& screen_width>=100)</code>	
<code>siemens && !screen_width!=100 //</code>	Syntax error
<code>siemens !nokia</code>	false
<code>nokia && (screen_width>100 </code> <code>screen_height>100</code>	Syntax error

Preprocessor Coding Tips Compilation Based on Device Platform Versioning

On occasion, device lines from certain manufacturers have various platforms and platform versions. In this example, we take Nokia and their s60 and s40 platforms, both having versions 1 and 2. To accomplish this, we create configurations and add abilities as shown in [Table 14-4](#).

Table 14-4. Example Configurations and Abilities

Nokia.s60.v2	Nokia.s60.v1	Nokia.s40.v2	Nokia.s40.v1
manufacturer=Nokia	manufacturer=Nokia	manufacturer=Nokia	manufacturer=Nokia
platform=s60	platform=s60	platform=s40	platform=s40
platform_version=2	platform_version=1	platform_version=2	platform_version=1

Now when the symbols and variables are properly defined, the code that will compile for all four platforms based on their differences could look something like the following:

```
//#if manufacturer=="Nokia"
.
compile all code common to nokias, e.g. using FullC
.
//#if platform=="s40"
...
add s40 specific code
...
```

```
//#if platform_version==1
...
compile s40.v1 specific stuff
...
//#elif platform_version==2
...
compile s40.v2 specific stuff
...
//#else
...
compile s40 generic
//#endif
...
...
//#elif platform=="s60"
...
add s60 specific code
...
    //#if platform_version==1
    ...
    compile s60.v1 specific stuff
    ...
    //#elif platform_version==2
    ...
    compile s60.v2 specific stuff
    ...
    //#else
    ...
    compile 640 generic
    //#endif
    ...
    ...
//#else
...
add generic Nokia code
//#endif
//#endif
```

Under many circumstances, however, this approach might not be possible or will need a lot of redundant code writing, especially when actual features in the software depend on the platform specification. This approach will yield good results when handling various video/audio formats, for example. When functionality and features in the software are based on the platform specification, a more incremental approach is desirable:

```
code common to all devices
...
//#if manufacturer=="Nokia" && platform=="s60" &&
    ...
//#elif manufacturer=="Nokia" && platform=="s60" &&
    ...
//#elif manufacturer=="Nokia" && platform=="s40" &&
    ...
//#elif manufacturer=="Nokia" && platform=="s40" &&
    ...
//#endif
```

Depending on the situation, both approaches can be mixed together to get the desired results. The IDE provides some predefined abilities such as CLDC and MIDP versions, as well as symbols for various JSRs that can be used and do not have to

be defined by hand.

Using Configuration Abilities

Although associating code blocks with individual configurations may be a sufficient solution for some applications, it can sometimes be difficult to maintain. You might find that you are always adding a certain group of configurations to the same code blocks. Or, when adding a new configuration to an existing project, you might find it onerous associating that configuration with each existing and appropriate code block.

In reality, code blocks are usually defined to address some specific feature of the platform to which that code will be deployed; and often, several deployment platforms share the same features and, thus, the same code blocks. To address these problems, configurations have the concept of abilities.

Abilities are identifiers that can be associated with code blocks in much the same way that configuration names are. Configurations can then be associated with as many abilities as desired. Once this is done, activating a configuration uncomments any code block containing either the configuration or any of its associated abilities.

For example, you might be creating an application with six different distributions. In your source, you create some code blocks that handle calls to a vendor-specific Bluetooth implementation. Three of your target distribution platforms will have that specific Bluetooth API available. Rather than list in each code block those three configuration names, you would instead associate that code block with an ability called `VendorSpecificBluetoothAPI`. Then you associate it with the three configurations that support the Bluetooth API.

This makes maintenance much easier for existing code blocks. Now, if you decide to support a new device that has a certain ability, you can simply attach all appropriate abilities to the new

configuration. Then it will automatically be compatible with all existing code blocks.

As such, it is almost always preferable to use abilities when creating code blocks rather than simply configurations. The only cases for using pure configuration names are when you are never planning on supporting more than a few platforms for a given application or when the code block is specific to only one deployment platform.



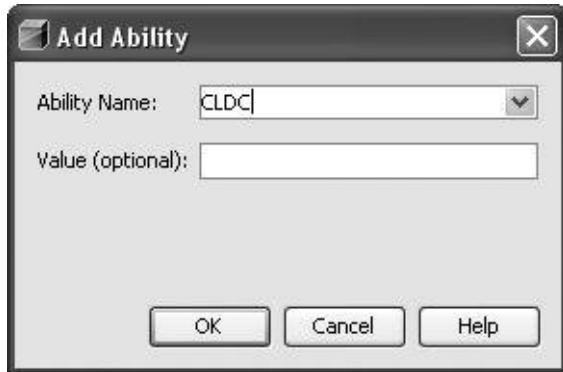
In principle, abilities can be used as three types of values even though they are all stored as `String` objects internally. They can be treated as Boolean symbols and the preprocessor can check for their definition; they can be treated and compared as Strings (e.g., `ScreenSize="100x120"`); and they can also be used as Integers (e.g., `ScreenWidth=100`), provided that the string value they hold is a valid integer.

Creating and Associating an Ability with a Configuration

To create and associate an ability with a configuration:

1. Right-click your project in the Projects window and choose Properties to open the Project Properties dialog box.
2. Select the Abilities panel.
3. From the Project Configuration drop-down box, select the configuration you would like to associate with the new ability.
4. Uncheck the Use Values from "DefaultConfiguration" checkbox (if you did not select DefaultConfiguration in the previous step).
5. Click the Add button to bring up the Add Ability dialog box, shown in [Figure 14-7](#).

Figure 14-7. Add Ability dialog box



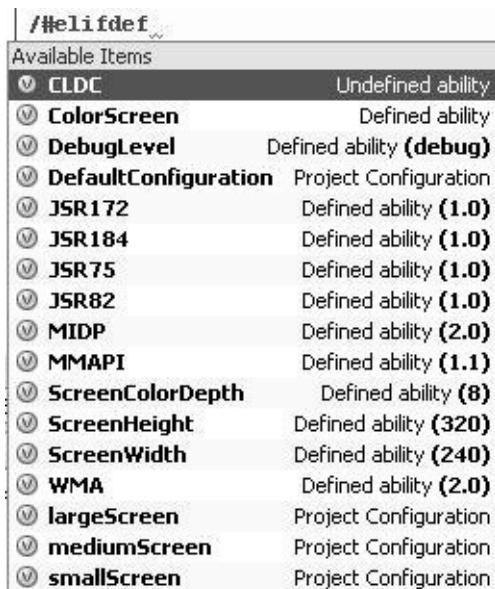
6. If the ability you wish to associate with the configuration is not in the Abilities list, simply write your own identifying name in the text field, add a value if necessary, and click

OK to add the ability. Otherwise, you can use one of the predefined abilities.

7. The newly created ability is listed and selected in the Abilities panel of the Project Properties dialog. If you are done adding abilities to the device configurations, click OK to close the Project Properties dialog box.

Once an ability is associated with at least one configuration in the project, that ability will appear in the code completion popups when you edit preprocessor blocks in the editor as shown in [Figure 14-8](#). The preprocessor code completion is invoked in two regimes, either for directives following the //# token or for abilities following a directive statement.

Figure 14-8. Ability code completion



The list of abilities contained in the Abilities List view of the Add Ability dialog box contains all the abilities that are attached to configurations in any project that the IDE is

currently aware of. The IDE is aware of more projects than those that are open, so don't be surprised if you see some abilities there that don't belong to any open project.

There is no Remove Ability button in the Add Ability dialog box, because it simply displays all abilities that are associated with any configuration.

Localizing Applications

In the mobile development world, there are many different ways of handling localization or translation of your product into different languages. The official support within the NetBeans Mobility Pack is based on the concept of using bundled `message.property` files. The messages file, and a small support class that uses it, can be generated for you automatically by the Localization Support Class template:

1. In the Projects window, right-click the package you would like to contain your `LocalizationSupport` class and choose New | Localization Support Class. If that option is not available directly in the menu, choose File/Folder to display the full array of templates.

Alternatively, select File | New File (Ctrl-N) from the main menu.

2. Select the MIDP node in the Categories list box and then Localization Support Class in the File Types list, and click Next.
3. Optionally, fill in the following fields in the Name and Location panel of the wizard:

Class Name: The generated class' name.

Messages Bundle Name: The name of the message bundle.

Package: The package in which to create the files.

Default String Value: The value that is used if a property is not found in the message bundle.

Error Message: The error message that is displayed when

there is a problem loading the message bundle.

4. Click Finish to create the files in the locations displayed in the Created Class File and Created Bundle File fields.

The automatically generated support class should now appear in your project and can be modified as desired. The following process outlines how an existing application can be localized using the support class:

1. All hard-coded text strings from your application should be added to the created `message.properties` file using the following format:

`PROPERTY_NAME=My Translatable Text String`

2. The strings in your source files should then be replaced with code like this:

`LocalizationSupport.getMessage(" PROPERTY_NAME")`

3. Once all strings have been added to `message.properties` file, right-click its node in the Projects window and choose Add Locale.

4. Select a locale that you want to support from the Predefined Locales list box, or use the combo boxes at the top of the form to define a new locale.

5. Click OK.

6. Expand the `message.properties` node in the Projects window and doubleclick the newly added locale.

7. Translate all properties into the appropriate language.

8. Repeat these last five steps until all supported languages have been added.

This technique uses the `microedition.locale` property of the phone to determine which version of the `message.properties` file should be used. If the region is not found, the default bundle is used.

If you prefer, you can force a particular region to be used by calling the following code before using `LocalizationSupport.getMessage()`:

```
LocalizationSupport.initLocalizationSupport("en_US")
```

In this case, the `en_US` version of the `message.properties` file is always used. Forcing a region is useful when you are planning on supporting only one language per distribution JAR file. The Filtering panel should be used to ensure that only the used properties file is bundled with the application.

Using the MIDP Visual Designer

NetBeans Mobility Pack includes the MIDP Visual Designer to assist with the creation of MIDlets. This full-featured designer allows you to create your application's flow rapidly and modify the screen content of standard MIDP 1.0 or MIDP 2.0 components using an intuitive GUI. See the Understanding the Flow Designer and Understanding the Screen Designer sections later in this chapter for an overview of the tool and the following sections for some common task descriptions.

Creating a New Visual Design

To create a new visual design:

1. In the Projects window, right-click the package you would like to contain your visual design and choose New | Visual Midlet. If that option is not available, select File/Folder.
Alternatively, select File | New File (Ctrl-N) from the main menu.
2. Select the MIDP node in the Categories list box and then Visual MIDlet or HelloMIDlet.java in the File Types list, and click Next.
3. Select the MIDP version for your MIDlet to use (according to the version supported by the device you are developing for) and click Finish.

Your visual design opens in the Flow Design view. If HelloMidlet was selected, the MIDlet will already contain a TextBox screen that displays the text "Hello, World!"

Adding Objects to the Component Palette

User-defined screens and items can be added to the component palette for use with the designer. Though the screen editor does not allow editing of all attributes of user-added components, they can still be used with the Flow Designer:

1. Ensure that the class you want to use is on your project's classpath.
2. Right-click the Palette window and choose the Palette Manager.
3. Click Add from Project.
4. Select the project that contains the object you want added to the component palette and click Next.
5. Select all the objects to add, using the Found MIDP Classes list.

All classes inheriting from `Displayable` or `Item` on the selected project's classpath are shown in the list.

6. Optionally, use the Add into Category combo box to select the palette category to which the object will be added, and click Finish.
7. Click Close.

Your component(s) will now be available in the chosen category whenever you open the Visual Designer. This is true for any project you open. As such, it is possible that a class in the component palette won't be on your project's classpath.

Building a Small Application with the Visual

Designer

The following steps illustrate how to create the skeleton of a two-screen application that could be used to send an SMS:

1. Create a new visual design document and click the Flow Design button in the designer toolbar.
2. Click and drag the TextBox from the Screens group in the Palette window to the Flow Designer.
3. Click and drag the transition source labeled Start Point on the Mobile Device screen to your newly added TextBox.

Your first screen has now been added. If you run the application at this point, the application will start with a text field, with no commands associated with it.

4. Click and drag the OK command from the Commands group in the Palette window to your TextBox screen. Now add an Exit command to the same screen.
5. Click and drag an Alert from the Screens group in the Palette window to the Flow Designer.
6. Double-click the new Alert screen. It will open in the Screen Designer.
7. Click the Device screen where it reads <Enter Text>. This text box will now be editable. Enter the message "SMS Sent" and press Ctrl-Enter to save your changes.
8. Use the combo box at the top of the Screen Designer to switch to the TextBox screen, as shown in [Figure 14-9](#).

Figure 14-9. Visual Designer with a TextBox screen added

[\[View full size image\]](#)

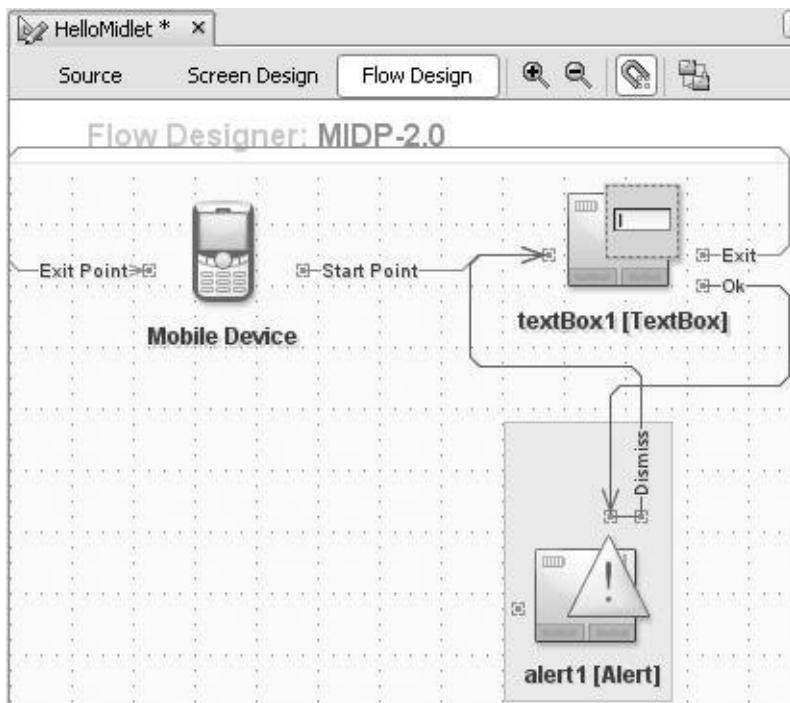


9. Click in the text box and delete the `<Edit Text>` string. Again, press Ctrl-Enter to save your changes.
10. Use the property editor to set the Title property to `"Enter SMS text:"`.
11. Click the Flow Design button in the toolbar to return to Flow Design view.
12. Click and drag the transition source labeled `okCommand` from the TextBox screen to the Alert screen.
13. Click and drag the transition source labeled `exitCommand` from the TextBox screen to the Mobile Device screen.

The Form Designer should now look similar to what is shown in [Figure 14-10](#). If you were to run the application at this point, it would begin with a textbox screen and OK and Exit options on its soft keys. Pressing the OK soft key would

bring you to the alert screen. Pressing the Exit soft key would exit the program.

Figure 14-10. Visual Designer in Flow Design view with an Alert screen added



14. Using the Flow Design, right-click on the transition from `okCommand1` and choose Go to Source. There you can enter your code that will do the SMS sending.

You have now created a very simple MIDP application.

Understanding the Flow Designer

The Flow Designer visually represents the different paths that can be taken between your application's different screens. Using this view, you can add or remove screens, as well as the transitions between them. It is also possible to modify the properties of the screens and transitions.

Figure 14-11. Action dialog box

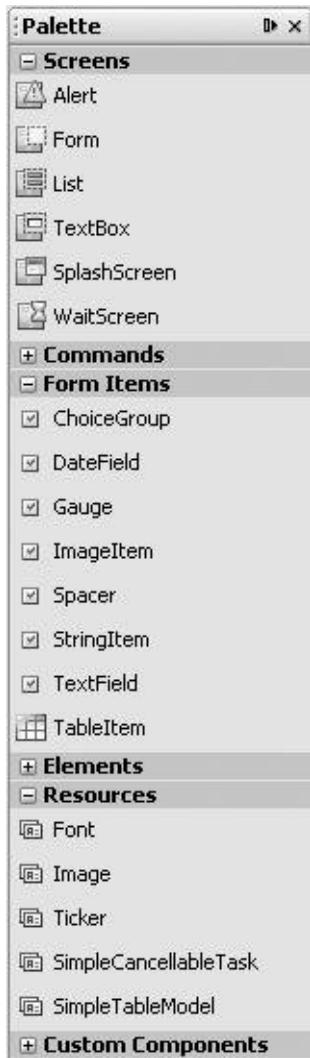


The Flow Designer is composed of four parts: the component palette, the Inspector, the Flow Design panel, and a property sheet. Following is a quick overview of these panels, as well as their respective capabilities.

Component Palette

The component palette (shown in [Figure 14-12](#)) contains groupings of all Java objects that can be added to your application with the designer. If an object does not exist in the component palette, it cannot be added to the application using the Flow Designer.

Figure 14-12. Component palette



The following categories are used:

- **Screens.** Contains items for creating Alert, Form, List, and TextBox components.
- **Commands.** Contains an item for each command type defined in `javax.microedition.lcdui.Command`. Commands are used as sources for transactions between screens.
- **Form Items.** Contains items that can be added to screens.
- **Elements.** Contains elements that can be added to List and ChoiceGroup components.
- **Resources.** Resources that can be used by other items. Includes Font, Image, and Ticker.
- **Custom Components.** Any custom components that you have added to the palette.

Special Components

The Visual Designer provides certain special components that can be used when creating applications and the standard MIDP components are not enough.

Splash Screen

Splash screen is a simple component that can be used to present logos during application startup cycle, for example. To create a splash screen, simply drag the SplashScreen component onto the Flow Design work space. Select it using the mouse and use the property editor to customize it. Image, Text,

Text Font, and Timeout are amongst the most commonly used attributes.

Table

The table component is a custom component based on the MIDP CustomItem class and can be used to present data in a formatted table. The component can be used only for viewing data and it might not work properly with all real devices, as it is not fragmented and implementations of the CustomItem vary from device to device.

In order to use this component, you must first create a form and then add the Table component to the form by dragging it from the Palette window onto its icon. Using the Property Editor, you can set attributes such as fonts, preferred size, and title. You populate the table with data using your own custom code.

Wait Screen

The WaitScreen is a non-visual component that can be used as a junction box that splits the program flow. If the `execute()` method implemented in the `get_simpleCancellableTask()` method throws an exception, the program will follow the Failure route; otherwise, it will follow the Success route. The `execute()` method's code runs in a separate thread so the application does not freeze in the case that the task takes too long to complete.

To use the Flow Designer to create a program with a WaitScreen component as a junction:

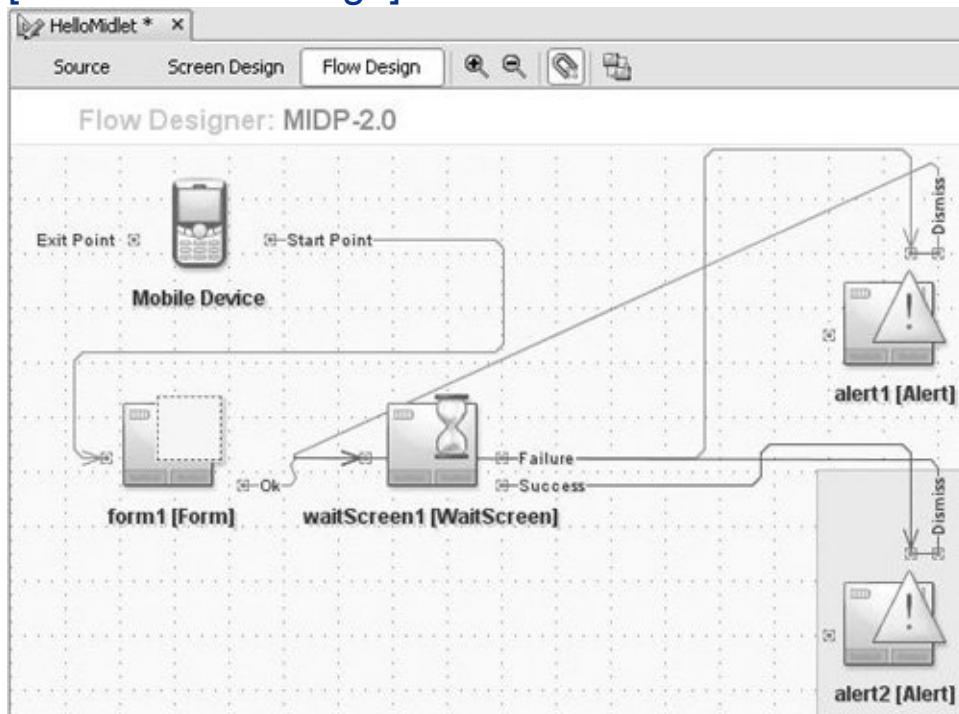
1. Drag a Form component from the Palette window onto the Flow Design workspace.
2. Drag a ChoiceGroup item from the Palette window onto the

Form component you just added.

3. Using the Inspector, expand the Form component, then the ChoiceGroup item and add two elements named **Pass** and **Fail** to it by twice right-clicking the Elements node and choosing Add Choice Element.
4. Drag WaitScreen and two Alert components onto the workspace.
5. Connect the components as shown in [Figure 14-13](#).

Figure 14-13. Sample WaitScreen program flow

[[View full size image](#)]



6. Using the Inspector, again select the ChoiceGroup item by clicking on it, and using the Property Editor, set the type to Exclusive.
7. Select the WaitScreen by clicking on it and then, again

using the Property Editor, click on the Task item. This will bring up a simple Source Editor window. There you will type the following Java source code (everything starting with `String` and ending with `()`); needs to be on the same line in the Source Editor. The only space in the line is between `String` and `action`).

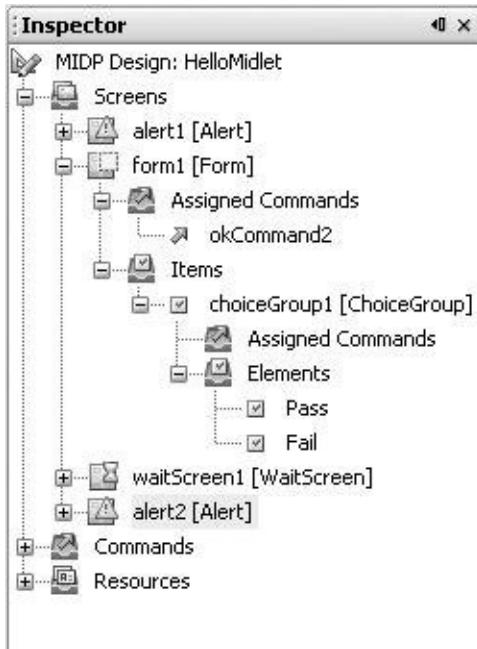
```
String  
action=get_choiceGroup1().getString(get_choiceGroup  
));  
if (action.equals("Fail"))  
    throw new Exception("Failure");
```

When you run the program using your favorite emulator, you use the ChoiceGroup item to select an action, and then based on that, you will get a Fail or Pass Alert.

Inspector

The Inspector (shown in [Figure 14-14](#)) displays all objects that have been added to the visual design in a tree formation. The root node of the tree is the MIDP Design node with child nodes for Screens, Commands, and Resources. Any objects from these groups that have been added to the design appear under their respective group header nodes.

Figure 14-14. Inspector



If a node represents an object that can have action commands associated with it, it has an Assigned Commands group node containing each assigned action. Similarly, objects that support elements or items contain an Elements or Items group.

Selected objects in the Inspector are highlighted in Screen Designer or Flow Designer view (if the object appears in the currently active design view). Additionally, the object's properties are displayed in Property Sheet view.

Right-clicking a component node opens a menu consisting of standard node commands (Rename, Delete, Move Up, Move Down, and Properties).

Right-clicking the Action Command, Elements, or Items node opens a menu containing actions for adding a new node to the list or changing the order of the list.

Property Sheet

A standard property sheet is visible when the designer is open. This property sheet shows the editable properties for any object selected in Designer or Inspector view.

The property sheet contains all MIDP2 properties for the selected object. Additionally, there are some items specific to the source code and components, as described here:

- **Instance Name.** The name of the object.
- **Action Source.** When a transition is selected that is initiated with a command, this property appears. It defines which command object to use as the transaction source.
- **Lazy Initialized.** Checkbox that determines when the component is initialized. If not selected, it is initialized when the MIDlet's `startApp()` method is invoked. If selected, it is initialized the first time the component is needed.
- **Action.** Property that appears when an action command is selected. A special property editor is used to set this value.
- **Radio Buttons.** Used to control which screen is shown when this transaction occurs. Selecting nothing means there is no target. Selecting Exit means the application will end when the transition is followed. Selecting Switch enables the target screen drop-down box, which can be used to select a new screen.
- **Target Displayable.** Contains all screens available in the visual design. The selected screen is the one displayed when the command is invoked.
- **Forward Displayable.** Select what will be displayed after an alert. Appears only when a connection is selected that

targets an alert.

Additional properties are determined by what is supported by the associated object.

Several property types have special editors associated with them:

- **String Editor.** A dialog box opens, containing a text box. Values entered in this text box are used as the property value. If the Use as Custom Code Expression checkbox is selected, the value entered in the text box is used as though it were Java code rather than as a static string. So, for example, entering `getTitle()` in an alert's Title property when Use as Custom Code

Expression is selected causes that component to be constructed with code like this:

```
alert = new Alert(getTitle(), "text", get_image2(),
```

You must implement a `getTitle()` method for the component to work properly.

- **Image Dialog.** Used to set the resource path for image resources. The Select Image list contains all images that will appear in the project's distribution JAR file. Selecting an image displays a preview of the image, as well as the dimensions, file size, and name of the file. Clicking OK sets the property to the path of the selected image.
- **Input Constraints.** Selecting the Input Constraints property on TextBox objects opens the input Constraints dialog box. Use the radio button to specify the restrictive constraint setting and the checkboxes to specify any additional constraint modifiers.

- **Font.** Editing the Font property on Font resources opens the Font dialog box. This dialog box allows you to specify which font to use. Selecting Custom from the main radio button enables the dialog box's font checkboxes; otherwise, the default or system font is used.
- **Layout.** Editing the Layout property of any Form item opens the Layout dialog box. This dialog box allows you to specify properties that determine how the item will appear within the form.

Flow Designer

The main screen of the Flow Designer uses drag-and-drop to add and connect components in the MIDlet. All designs contain a Mobile Device element that represents the MIDlet's start point.

Displayable components (screens) can be dragged from the Palette window and dropped onto the Design screen. After a screen has been added, appropriate commands, elements, items, and resources can be added to it by dragging the object from the Palette window and dropping it on the screen.

Transition sources are displayed on the right side of the component to which they are attached. These are used to create transitions to different screens by selecting the transition source and dragging to the target screen. The connection is visualized in the design as a line.

Transitions can also be selected. Doing so paints the transition in a different color and displays the transition sources' properties in the property sheet.

Selecting a screen node highlights all transitions to and from that screen in a different color. Double-clicking a screen opens it

in Screen Design view.

Toolbar

The toolbar of the Flow Designer has four items:

- **Snap to Grid.** Toggles between displaying and hiding a dotted grid on the background of the designer page. Also determines whether screens snap to set locations or can be moved with complete freedom.
- **Realign Components.** Realigns all existing screens to a grid pattern.
- **Zoom in.** Zooms in on a selection.
- **Zoom out.** Zooms out from a selection.

Understanding the Screen Designer

The Screen Designer allows you to customize screens that have been added to the visual design. It contains two main sections: Device Screen and Assigned Commands, which will be discussed here.

Device Screen

The device screen simulates how the screen will appear on a real device. All elements, items, and resources (components) appear in the order in which they are listed in the Inspector view. Components can be dragged to new locations in the Inspector to change display order.

Hovering over a component highlights it with a dashed line. Clicking the component selects and highlights it with a solid line. Clicking editable areas within a selected component (as shown with a moving dashed line) allows inline editing of the selected area. Noneditable sections of components cannot be modified inline but usually can still be selected. As always, properties of the selected object appear in the property sheet.

New components can be dragged and dropped directly to the desired location of the device screen. Only components that are valid for the screen type are added. Appropriate elements and items can be added to existing components (for example, Choice Element objects can be added to existing Choice Group objects). Trying to add a component not supported by the screen/object shows a standard Not Allowed icon.

Assigned Commands

This portion of the Screen Designer shows all transition sources attached to the screen by the user. The box may be selected to see the transaction source's properties in the Properties dialog box. Additionally, you can open the transaction target in the Screen Designer by clicking its name in each of these command boxes.

This list has a subsection for item commands as well. These commands appear only when you are editing a list screen; one Item Command box is displayed for each list element. They function in the same way as normal Assigned Commands boxes.

Toolbar

The Screen Design toolbar has a combo box containing all screens added to the visual design. Selecting a screen from this drop-down menu displays the screen in the Screen Designer.

Also in the toolbar is a Zoom icon. This simply increases the viewable size of all components in the screen within the IDE.

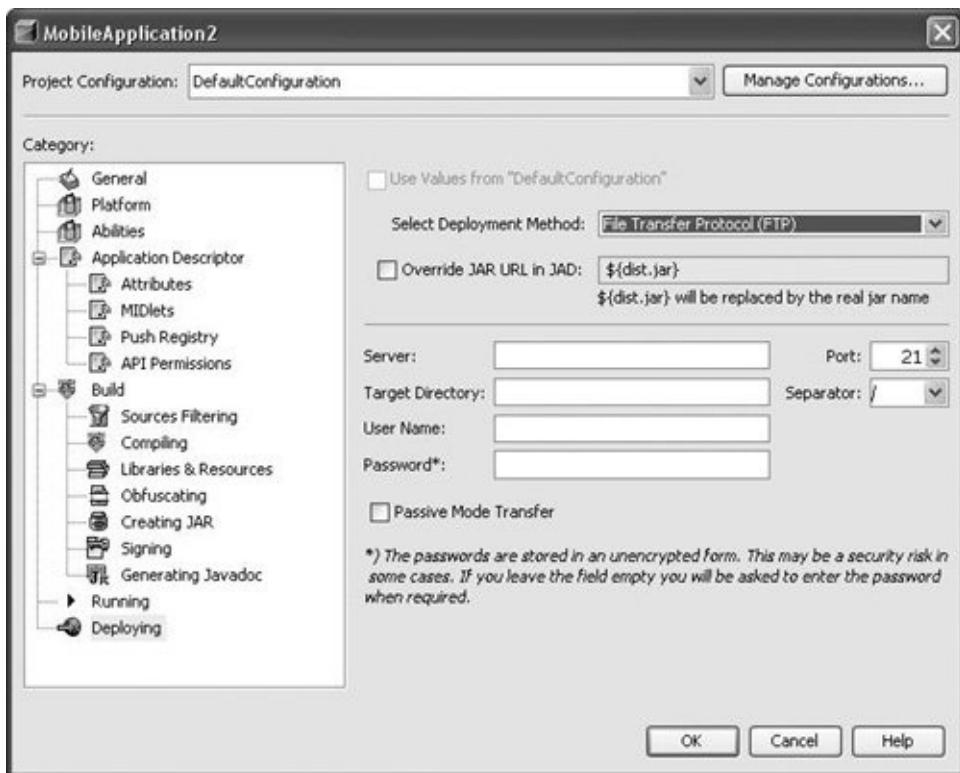
Deploying Your Application Automatically

NetBeans Mobility Pack provides you the capability to deploy your application. There are several methods of deployment, including simply moving the JAR file to a specified location on the local machine or using various protocols to move the file to a remote server. These options are specified in the Deploying panel of the Project Properties dialog box.

1. Right-click your project in the Projects window and choose Properties.
2. Select the Deploying node in the Category tree (as shown in [Figure 14-15](#)).

**Figure 14-15. Project Properties dialog box,
Deploying panel**

[\[View full size image\]](#)



3. In the Select Deployment Method combo box, select the method you would like to use.
4. Optionally, set **MIDlet-Jar-URL** to a value other than the default (which is simply the JAR filename) by deselecting the Override JAR URL in JAD checkbox and entering a new value in the text field.

The remainder of the form is based on which deployment method you have selected above.

File Copy

Use the Target Directory field to specify where the distribution JAR and JAD files should be copied.

FTP

1. Enter the server location in the Server field.
2. Optionally, use the Target Directory field to enter the remote directory to which the JAR file and JAD file should be copied.
3. Enter the remote server's port in the Port field.
4. In the Separator field, specify the path separator used on the server.
5. Enter the remote username in the User Name field.
6. Optionally, select the Passive Mode Transfer checkbox. The Passive Mode Transfer checkbox is used to toggle between passive and active FTP mode. If you are having trouble connecting to your FTP server, and you are behind a firewall, you should select this checkbox.

SCP

1. Enter the server location in the Server field.
2. Optionally, enter the remote directory to which the JAR and JAD files should be copied in the Target Directory field.
3. Enter the remote server's port in the Port field.
4. Enter the remote username in the User Name field.
5. Select the Use Password Authentication radio button or, if you use public/ private key authentication, select the Use Authentication Key radio button. If you select Use Authentication Key, enter the location of your private key in the Key File field.

WebDAV

- 1.** Enter the server location in the Server field.
- 2.** Optionally, enter the remote directory to which the JAR and JAD files should be copied in the Target Directory field.
- 3.** Enter the remote server's port in the Port field.
- 4.** Enter the remote username in the User Name field.

Deployment to Sony-Ericsson Devices

If you are using a Sony-Ericsson SDK as one of your emulator platforms, you can deploy your applications directly to a device that is connected to your computer via the Connection Proxy. Simply select Sony-Ericsson Phone and your application will deploy to the real device.

Once you have finished configuring the options in the Deploying panel, click OK to apply the changes and close the Project Properties dialog box. To deploy the project, right-click your project in the Projects window and choose Deploy Project.

Incrementing the Application's MIDlet-Version Automatically

The IDE has a built-in method that allows you to auto-increment the MIDletVersion attribute in the JAR file each time you deploy the application. This can be useful, as some physical devices have a Check for Updates function for installed applications. If you increment the value, this device function automatically detects a new version when you deploy the application.

Deploying with an incremented value can be done in the following manner:

- 1.** Right-click your project in the Projects window and choose Properties.
- 2.** Select the Attributes node in the Project Properties dialog box.
- 3.** Select the MIDlet-Version record and click Edit.
- 4.** Enter `${deployment.number}` in the Value field.
- 5.** Click OK.

Using Ant in Mobility Projects

As with general Java project types, Mobility project-related commands are controlled by an automatically generated Ant script. This Ant script is named `build.xml` and is located directly in your project's home folder. Examining this file reveals that it, by default, simply imports the project's `build-impl.xml` file. The relationship between these two files is important to understand if you would like to modify the build process in some way. Feel free to modify `build.xml` however you like, but do not change `build-impl.xml`. `build-impl.xml` may be regenerated by the IDE, so changes you make to this file can be lost.



Should your `build-impl.xml` file become corrupted, you can force it to be regenerated by deleting it and closing and then reopening the project.

Adding functionality to the NetBeans build process can be accomplished by overriding the targets defined in `build-impl.xml` file that run both before and after the main build targets. These are described in detail in each project `build.xml` file.

It is also acceptable to override the main project targets if you are interested in completely changing the behavior of the build script. These main targets are invoked by the similarly named project commands within the IDE.

Mobility Ant Library

Mobility Pack ships with a special Ant library responsible for

handling J2MEspecific tasks. This library can be freely shared, and is located at [NetBeansHome/mobility/modules/org-netbeans-modules-kjava-antext.jar](#).

Tasks that exist in the Mobility Library are listed here, along with descriptions of what they can do. Each task is listed along with its attributes and nested elements. Required elements are bolded.

- **ExTRACTask:** extracts specified JAR and .zip files to a given location.
 - ***ClassPath:** the archives to extract as a full path.
 - ***ClassPathRef:** the archives to extract as a reference to an existing Ant object.
 - ***Nested ClassPath:** the archives to extract specified with a nested Classpath element.
- **Dir:** the target directory for extraction.**ExcludeManifest:** specifies if the **META-INF/Manifest.mf** files should be excluded from extraction. Defaults to false.
 - * one of the class path attributes must be defined.
- **JadTask:** support for updating existing JAD file with correct JAR size and URL information. Also provides support for JAR file signing.
 - JadFile:** location of the source JAD file.
 - JarFile:** location of the source JAR file.
 - Output:** destination JAD file. If unspecified, the source location is used.
 - Url:** value to be set for the **MIDlet-Jar-URL** property.

`Encoding`: the encoding used for the JAD file. Defaults to UTF-8.

`Sign`: set to true if signing should be used. Defaults to False.

`*KeyStore`: location of the `KeyStore` file.

`KeyStoreType`: use to set the keystore type explicitly. Valid settings are `Default` and `PKCS12`. If not set, the extension of the keystore file is used to determine the type.

`*KeyStorePassword`: the keystore password.

`*Alias`: the owner of the private key to be used for signing

`*AliasPassword`: the password to access the alias' private key.

`*Required` if `Sign` is True.

- `ObfuscateTask`: support for obfuscation using the ProGuard obfuscator.

`SrcJar`: location of source JAR file to be obfuscated.

`DestJar`: destination of the obfuscated JAR file.

`ObfuscatorType`: specifies which obfuscator to use. If `NONE` is selected, `SrcJar` is simply copied to the `DestJar` location.

`ClassPath`: classpath required by `SrcJar` classes. Can be specified using nested Classpath instead.

`ClassPathRef`: classpath for `SrcJar` classes specified as Ant reference.

`*ObfuscatorClassPath`: classpath for the obfuscator.

`*ObfuscatorClassPathRef`: classpath for obfuscator as Ant

reference.

*`Nested ObfuscatorClassPath`: classpath for the obfuscator as a nested element.

`Exclude`: a comma-separated list of classes that should not be obfuscated.

`ObfuscationLevel`: integer value specifying level of obfuscation. Valid values are 0 through 9, with the default 0.

`ExtraScript`: string containing any additional obfuscation commands.

*`Obfuscator` classpath must be set using one of these methods.

The exact commands used by different obfuscation levels can be seen within the IDE. Simply open the Project Properties dialog box and select the Obfuscating panel. Changing the obfuscation level on the slider displays the script commands in the Level Description panel.

- `PreverifyTask`: support for preverification.

`SrcDir`: location of classes to be preverified.

`DestDir`: destination directory for preverified classes.

`PlatformHome`: home directory of the emulator platform to be used for preverification.

`PlatformType`: controls the format of the preverification command line used. Valid values: `UEI-1.0`, `UEI-1.0.1`, or `CUSTOM`. The default value is `UEI-1.0.1`.

`Configuration`: the configuration to use for preverification. Valid values: `CLDC-1.0` and `CLDC-1.1`. Ignored when `PlatformType` is set to `CUSTOM`. `ClassPath`: classpath for sources in `SrcDir`.

Can also be defined using nested elements.

`ClassPathRef`: classpath for sources in `SrcDir` as an Ant reference.

`CommandLine`: command line to be used for preverification.
Required when `PlatformType` is `CUSTOM`.

- `RunTask`: support for running and debugging an application.

`*JadFile`: location of target application's JAD file. Required when `JadUrl` is not set.

`JadUrl`: URL for the application's JAD file when using OTA execution.

`PlatformHome`: location of emulator platform used for execution.

`PlatformType`: controls the format of the execution command line used. Valid values: `UEI-1.0`, `UEI-1.0.1`, or `CUSTOM`. The default value is `UEI-1.0.1`.

`Device`: target emulator platform device.

`ExecMethod`: determines if OTA or Standard execution is used.
Ignored if `JadUrl` is not set or `PlatformType` is `CUSTOM`.

`ClassPath`: classpath for sources in `SrcDir`. Can also be defined using nested element.

`ClassPathRef`: classpath for sources in `SrcDir` as Ant reference.

`SecurityDomain`: security domain in which execution takes place.

`Debug`: set to `true` to run in Debug mode. The default value is `False`.

`DebugAddress`: should just be the port number to use for attaching to the debugger.

`DebuggerAddressProperty`: the address to which the debugger tries to connect.

`DebugTransport`: specify the transport type to use for debugging. Default is `dt_socket`.

`DebugServer`: specify whether emulator should run in server mode. The default is `TRue`.

`DebugSuspend`: specify whether emulator should wait for connection before starting the application. The default is `TRue`.

`*CommandLine`: command line for running emulator. Required when `PlatformType` is `CUSTOM`.

Using Headless Builds

Because the IDE's project commands are based on Ant scripts and properties files, you can run these targets from outside of the IDE as "headless builds." Headless builds for Mobility projects operate under the same principles as for general Java projects. See [Chapter 3](#), Deploying a Java Project Outside of the IDE for more information.

Computers with NetBeans IDE

Assuming that the project has already been opened in NetBeans on the target computer and no reference problems exist, any Ant target can be invoked from within the project directory. For example, typing `ant jar run` at a command line will compile, package, and execute the project.

As always, Ant properties can be set by using the `-D` switch. Examine your project's `build.properties` file to see which properties can be set. Normally, this is used to activate a certain configuration. For example, the following line will run the project using the `BigScreenConfig`:

```
ant -Dconfig.active=BigScreenConfig run
```

Computers without NetBeans IDE

Running NetBeans projects on computers that have no NetBeans IDE installation is somewhat more complicated. It is strongly recommended that projects be opened within NetBeans

IDE, as then the IDE can be used to configure the project to work on the new machine. But if this is not possible, you should take the following actions to set up the project:

- You must have access to the Mobility Ant Library, as described in Using Ant in Mobility Projects earlier in this chapter.
- Create a `/private` subdirectory in `nbproject` containing a file called `private.properties`. This file should match the properties file located in the NetBeans user's `/private` directory for the same project. Update the hard path references to refer to local locations.
- Create a `build.properties` file somewhere on the local machine. This file should be similar to the one located on the NetBeans user's `{user home}/build.properties` file. It should contain all properties (with correct path information) that refer to libraries or emulator platforms used by the project in question.

Once these files have been created, the headless build can be invoked similarly to how it is started on machines with NetBeans installed. The only difference is that the `user.properties.file` should be set to point to the `build.properties` file created previously. So, for example,

```
ant-Duser.properties.file=C:\{path}\build.properties j
```

will build the distribution JAR files of the project. Make sure to use a fully qualified path to the `build.properties` file.

Using the Wireless Connection Tools

The Wireless Connection Tool is a utility that greatly simplifies the process of developing Java ME applications that connect to third-party web services or other kinds of back-end systems.

The tool can generate two kinds of clients that connect to external services over HTTP. If the device for which you are developing is JSR-172 compliant, you can easily generate a client application from a WSDL file or a URL. If the device does not support JSR-172, the wizard will create a web service proxy in the container, which can then be accessed via a generated servlet over a simple HTTP connection. In addition, you can write your own database proxy and the wizard will also generate a servlet in the container that you can use to access the database via the proxy.

For all the scenarios the wizard will generate a client application that is ready to run on the device and can be edited using the Visual Designer. However, it is a good idea to modify this client application to suite your needs before deploying.

Creating Applications that Consume WS-I Compliant Web Services

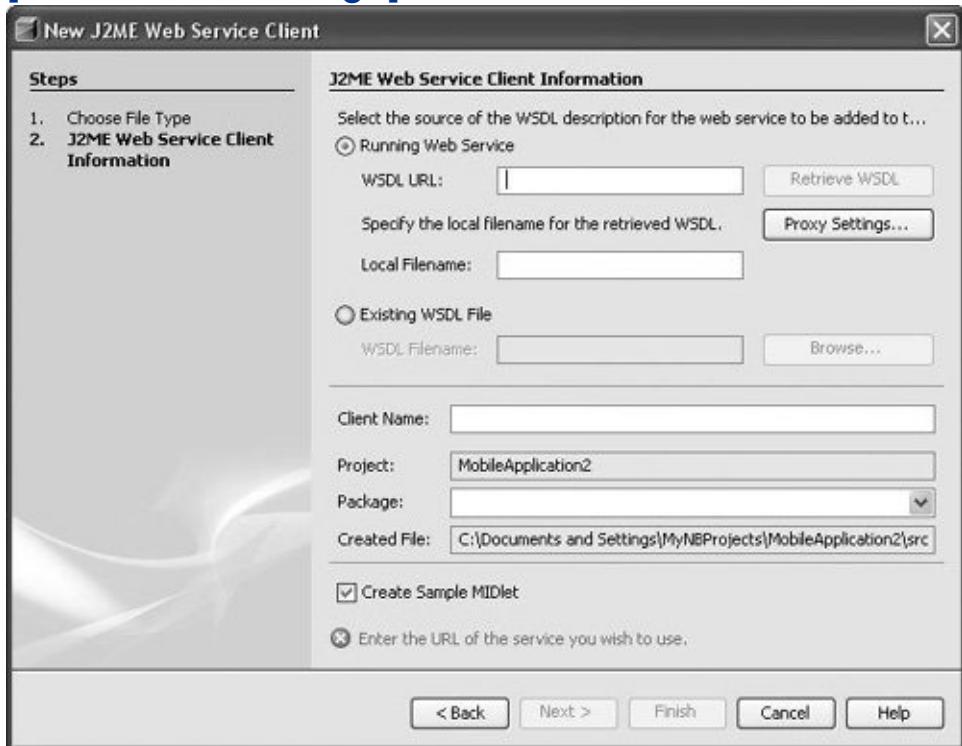
If the device you are developing for is JSR-172 compliant, you can easily create a client application that will consume the web service provided that ***it is WS-I compliant***. To generate a client application:

1. Make sure you have the Sun Java System Application Server installed and properly configured to work with the IDE.

2. Using the New Project wizard, create a new Mobile Application project.
3. Using the New File wizard, create a new J2ME Web Service Client.
4. In the New J2ME Web Service Client wizard (as shown in [Figure 14-16](#)), paste the URL of the WSDL file into the WSDL URL field. Then click Retrieve WSDL.

Figure 14-16. New J2ME Web Service Client wizard

[[View full size image](#)]



5. Make sure the Create Sample MIDlet checkbox is marked and click Finish.

Once the build process finishes, the Visual Designer will open the application for you. You can then edit and

customize the Visual MIDlet to your liking.

Creating Applications Using the Web Service Proxy

At the time of this writing, there are very few available devices that implement JSR-172. Because of this, the tool supports an alternate method of connecting to web services via a web service proxy, which sits in the container and talks to the web service for the device. The device talks to a proxy servlet that acts as a liaison between the web service and the device over a simple HTTP protocol.

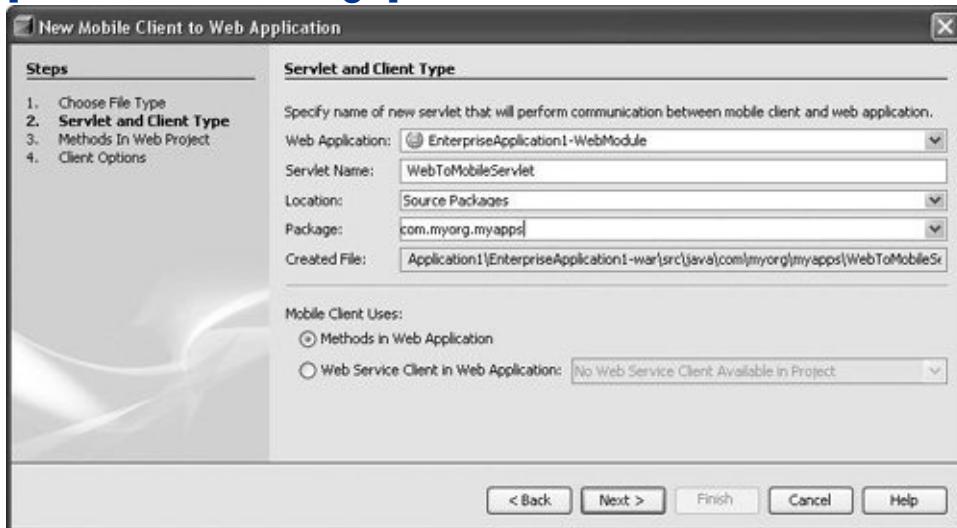
To create an application that connects to a web service via a web service proxy:

- 1.** Using the New Project wizard create a Web Application project.
- 2.** Right-click the web application you have just created, and choose New | Web Service Client.
- 3.** In the New Web Service Client wizard, paste a URL of a WSDL file very much like when you would be creating J2ME Web Service client, and click Retrieve WSDL. Then click Finish.
- 4.** Create a new empty Mobile Application project using the New Project wizard.
- 5.** Right-click the new Mobile Application project, and choose New | Mobile Client to Web Application. (If the Mobile Client to Web Application template does not appear in the New submenu, choose New | File/Folder to access all available templates.)

6. In the New Mobile Client to Web Application wizard (shown in [Figure 14-17](#)), select the web application and the web service client that you have just created.

Figure 14-17. New Mobile Client to Web Application wizard

[[View full size image](#)]



7. In the Package field, name the package into which the classes will be generated.
8. Select the Web Service Client in Web Application radio button, fill in the web service client to which to link, and click Next.
9. Select the services whose results you wish to use in your J2ME client application and click Next.
10. Fill out the client options to your liking and click Finish.

Again, a Visual MIDlet that you can use as a basis for your application will be generated and opened in the Visual Designer. Also, by double-clicking a node with an extension named **wsclient** in your Mobile Application project, you can change or

review the options you have selected previously and regenerate the client.

Using a very similar procedure you can easily connect to any application you might have running in the web container. For example, if you have an application that requires massive processing power to compute a complex mathematical formula, it would be better to perform the calculation on the back end rather than on the device. To do so, you just need to select the Methods in Web Application radio button on the Servlet and Client Type page of the New Mobile Client in Web Application wizard. The wizard will then generate a servlet that will take parameters from the device, pass them to the method, and then send back the result when it becomes available.

Finding More Information

The following online resources might be useful to you if you would like to learn more about NetBeans, NetBeans Mobility Pack, or Java ME technology in general:

- NetBeans download page:

<http://www.netbeans.org/downloads/index.html>

- Mobility Pack home page:

<http://developers.sun.com/prodtech/javatools/mobility/index.html>

- NetBeans IDE articles:

<http://www.netbeans.org/kb/index.html>

- Java ME documentation:

<http://java.sun.com/j2me/docs/index.html>

- Java ME technical articles and tips:

<http://developers.sun.com/techtopics/mobile/reference/technical.html>

- Java Mobility forums:

<http://forum.java.sun.com/wireless/>

- Developer Network Mobility Program:

http://sun.com/developers/mobility_program

Chapter 15. Profiling Java Applications

- [Supported Platforms](#)
- [Downloading and Installing the NetBeans Profiler](#)
- [Profiling Primer](#)
- [Starting a Profiling Session](#)
- [The Profiler Control Panel](#)
- [Monitoring an Application](#)
- [Surviving Generations and Memory Leaks](#)
- [Analyzing Performance](#)
- [Analyzing Code Fragment Performance](#)
- [Analyzing Memory Usage](#)
- [Attaching the Profiler to a JVM](#)

CREATING A ROBUST JAVA APPLICATION that is ready for production requires an implementation that will scale up as the load increases. In other words, your application should continue to perform well as users are added or additional data are processed, etc. Your application will not scale if it over consumes resources such as memory and CPU time. A profiler is a tool that helps you detect whether your application is suffering from performance problems because it is consuming more

resources than it should. Assumptions about Java application performance are frequently wrong, so it is important to use a profiling tool to help find and eliminate problems.

The NetBeans Profiler is a powerful tool that can help you identify performance problems in your application. Since it is integrated into the IDE, profiling becomes a simple task that does not require additional tools. The NetBeans Profiler allows you to monitor your application, determine the time used by specific methods, and examine how your application uses memory. The NetBeans Profiler uses advanced technology that reduces profiling overhead, making it easier for you to learn about the performance of your application. Following are some of the features of the NetBeans Profiler:

- **Low Overhead Profiling.** You control the profiler's performance impact on your application. Based on your selections, the performance impact will range from extensive to none.
- **CPU Performance Profiling.** Time spent in every method of your application or just in selected methods can be reported.
- **Memory Profiling.** You can check for excessive object allocations.
- **Memory Leak Detection.** The profiler's statistical reports make it easy to detect object instances that are leaking.

Supported Platforms

The NetBeans Profiler can profile applications when they are run in a Java Virtual Machine (JVM) that supports the JVM Tool Interface (JVMTI). Sun's JDK 5, update 4 (and higher), supports JVMTI.

Unlike the NetBeans IDE, the NetBeans Profiler includes binary code. This is needed to communicate with the JVMTI support in the JVM. As a result, the NetBeans Profiler support is platform specific. The supported platforms are listed below.

- **Solaris (SPARC and x86)**
- **Windows**
- **Linux**
- **Mac OS X**

Downloading and Installing the NetBeans Profiler

The standard NetBeans IDE download does not include support for profiling applications. You need to download the NetBeans Profiler separately and use its "add-on" installer to integrate the Profiler functionality with the NetBeans IDE installation you already have on your system.

You can find the NetBeans Profiler on the same download site as the NetBeans IDE.

Once you have downloaded the appropriate installer for your platform, launch it (in the same way that you launched the NetBeans IDE installer) and complete the wizard. The wizard will help you identify the installation of NetBeans IDE to which the NetBeans Profiler will be added.

Starting a Profiling Session

A profiling session typically contains the following steps:

- 1.** Select a project.
- 2.** Select a profiling task.
- 3.** Specify options for the selected task.
- 4.** Start the profiler and examine the data it displays.

If your project uses a JDK that is not supported, then the profiler will display the registered JDKs that are supported. To register a JDK with the IDE, choose Tools | Java Platform Manager; for more information, refer to the Changing the Version of the JDK Your Project Is Based On section in [Chapter 3](#).

Profiling Primer

The NetBeans Profiler makes it easy for all programmers to profile their applications, even if you have never used a profiling tool before.

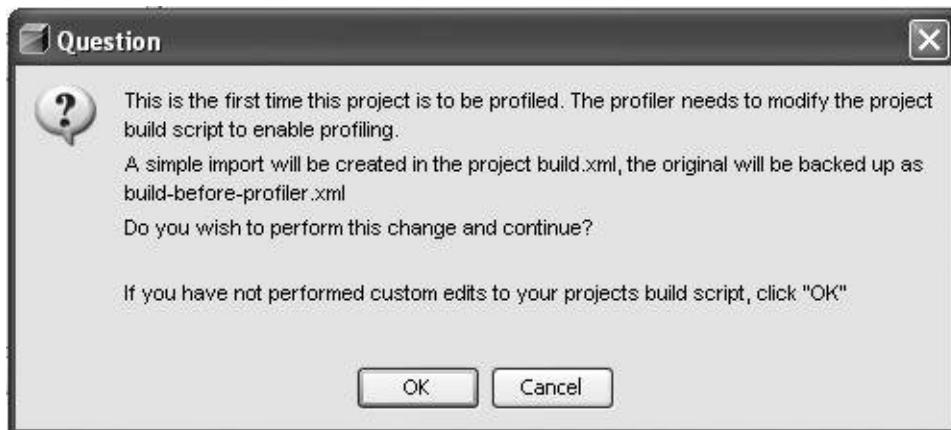
To help you get started, here are some terms that are used throughout this chapter.

- **Profiler.** A tool that shows you the behavior of your application as it runs in the Java Virtual Machine (JVM).
- **Instrumentation.** The insertion of profiling methods (counters, timers, etc.) into the Java bytecode of your application. These methods do not change the logic of your program and are removed when profiling is stopped.
- **Overhead.** The time spent executing profiling methods, instead of your application code.
- **Heap.** The memory pool used by the JVM for all objects allocated in your program by the `new` operator.
- **Garbage Collection.** The removal of objects from memory that your application is no longer using. Garbage Collection is performed periodically by the JVM.
- **Memory Leak.** An object that is no longer in use by your application but that cannot be garbage collected by the JVM because of one or more inadvertent references to it.
- **Self Time.** The amount of time needed to execute the instructions in a method. This does *not* include the time spent in any other methods that were called by the method.
- **Hot Spot.** A method that has a relatively large Self Time.
- **Root Method.** A method selected for performance profiling.
- **Call Tree.** All methods reachable from a root method.

In order to profile a project, the IDE must add targets to the

project's Ant build script. The first time you attempt to profile a project, the IDE will prompt for permission to modify the Ant build script in order to add those targets, as shown in [Figure 15-1](#).

Figure 15-1. Add Ant target dialog box



The simplest way to start using the profiler is to choose **Profile | Profile Main Project**; this command will work with projects from the following New Project wizard categories:

- General
- Web
- NetBeans Plug-in Modules (Module or Module Suite projects only)



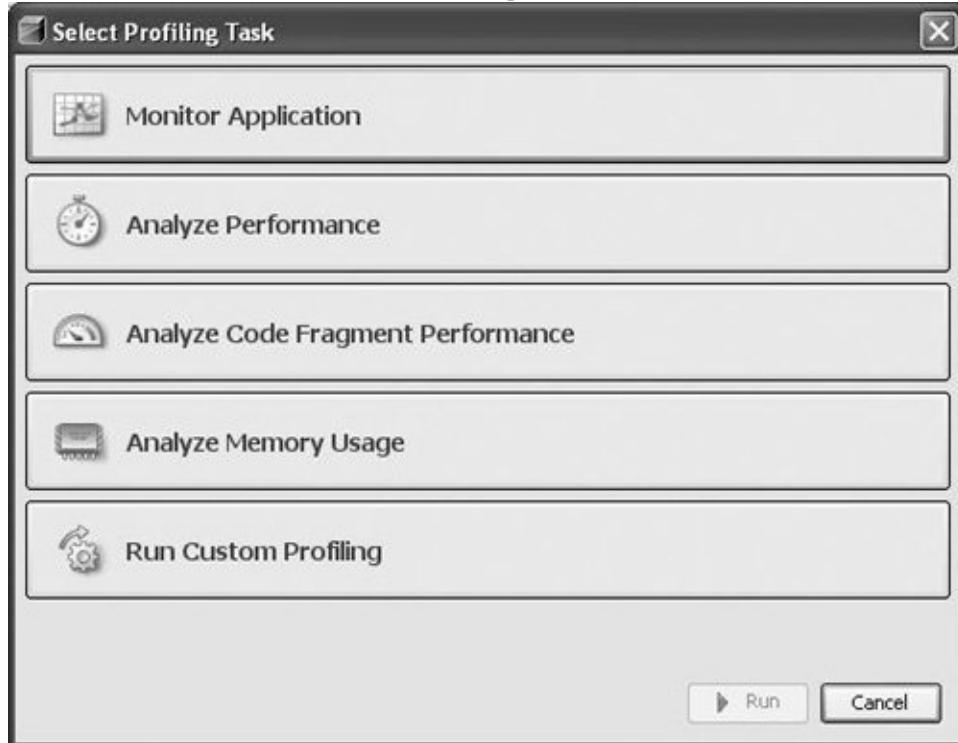
If you have set up a free-form project (one set up using a "with Existing Ant Script" project template), you need to do some extra configuration to get the profiling commands to work. You can either use Attach Profiler or you can add the necessary Ant target, as described in

[Chapter 16.](#)

The Profiler will display the Select Profiling Task dialog box (shown in [Figure 15-2](#)). When you click on a task button it will expand and display task-specific options. After setting the task-specific options, click Run to start the profiling session. If the profiler has never been run before on your system, then it will have to collect calibration data.

Figure 15-2. Profiler Task Selection dialog box

[\[View full size image\]](#)





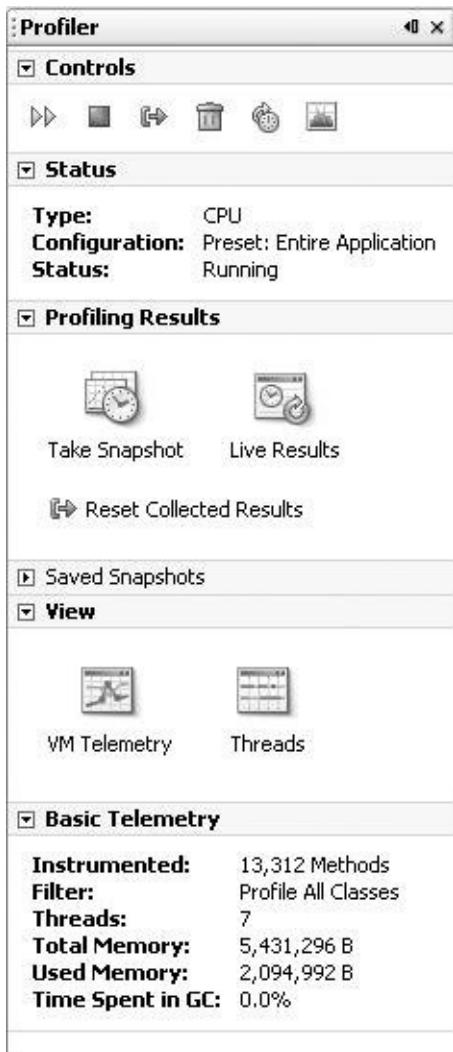
If the profiler's calibration data gets deleted or corrupted, you can recreate it by choosing Profile | Advanced Commands | Run Profiler Calibration.

When you are finished profiling, choose Profile | Stop.

The Profiler Control Panel

Regardless of which profiling task you choose, the IDE will display the Profiler Control Panel (shown in [Figure 15-3](#)).

Figure 15-3. Profiler Control Panel



Each section of the Profiler Control Panel can be expanded or

hidden by clicking the arrow icon next to the name of the section.

Controls

See [Table 15-1](#) for an explanation of the buttons in the Profiler Control Panel Controls section.

Table 15-1. Profiler Control Panel Controls

Component	Description
 ReRun Last Profiling (Ctrl-Shift-F2)	Run the last profiling command again.
 Stop	Stops the current profiling command. Also stops the target application if the application was started by the profiler.
 Reset Collected Results	Discards the already accumulated profiling results.
 Run GC	Runs garbage collection.
 Modify Profiling (Alt-Shift-F2)	Opens the Modify Profiling Task dialog box and allows you to run a new profiling command without stopping the target application.
 VM Telemetry	Opens the VM Telemetry Overview in the Output window of the IDE, displaying smaller versions of the telemetry graphs.

Status

See [Table 15-2](#) for an explanation of the entries in the Profiler Control Panel Status section.

Table 15-2. Profiler Control Panel Status

Component	Description
Type	The type of profiling: Monitor, CPU, or Memory
Configuration	Indicates whether the profiler was started with one of its preset configurations
Status	Running or Inactive

Profiling Results

See [Table 15-3](#) for an explanation of the entries in the Profiler Control Panel Profiling Results section.

Table 15-3. Profiler Control Panel Profiling Results

Component	Description
	Take Snapshot Displays a static snapshot of the profiling results accumulated thus far
	Live Results

Displays the current results of the profiling task



Reset Collected Results

Discards the already accumulated profiling results

Saved Snapshots

Enables you to manage the profiling snapshots associated with your project. When you select an open project in the combo box, the saved snapshots associated with that project are displayed. Double-clicking the name of the snapshot opens the snapshot in the Source Editor window.

View

See [Table 15-4](#) for an explanation of the entries in the Profiler Control Panel View section.

Table 15-4. Profiler Control Panel View

Component	Description
	VM Telemetry Opens the VM Telemetry tab. The VM Telemetry tab displays high-level data on thread activity and memory heap and garbage collection in the JVM.
	Threads Opens the Threads tab. When Enable Threads Monitoring is selected in the Select Profiling Task dialog box, application thread activity is

displayed in the Threads tab.

Basic Telemetry

See [Table 15-5](#) for an explanation of the entries in the Profiler Control Panel Basic Telemetry section. You can see the graphic presentation of some of this information by clicking the VM Telemetry and Threads buttons in the View section.

Table 15-5. Profiler Control Panel Basic Telemetry

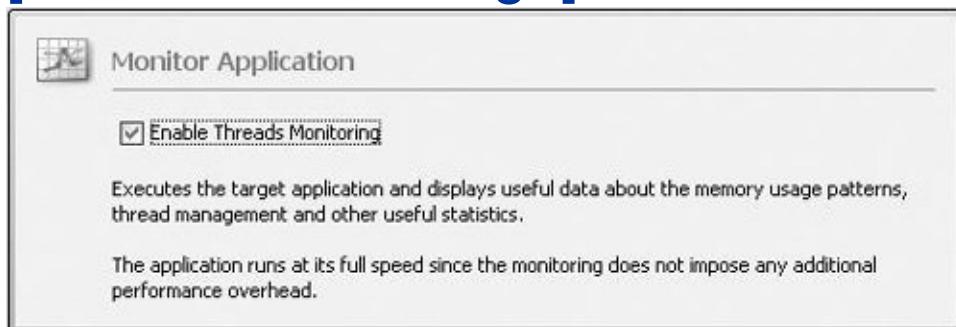
Component	Description
Instrumented	When doing memory profiling, the number of classes with profiler instrumentation; when doing CPU performance profiling, the number of methods with profiler instrumentation.
Filter	Type of filter (if any) that was specified.
Threads	Number of active threads.
Total Memory	Allocated size of the heap.
Used Memory	Portion of the heap that is in use.
Time Spent in GC	Percentage of time spent performing garbage collection.

Monitoring an Application

The Monitor Application task is useful for watching high-level statistics as your application runs. The Monitor Application task does not do any instrumentation and therefore imposes no profiling overhead. To choose it, click Monitor Application (shown in [Figure 15-4](#)).

Figure 15-4. Monitor Application options

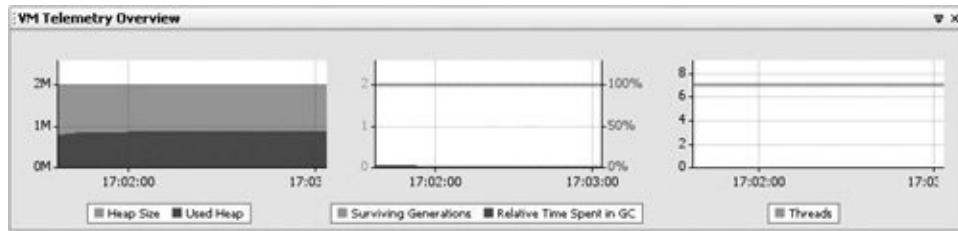
[[View full size image](#)]



The Monitor Application task will display basic information about your application: heap memory allocated, heap memory in use, percentage of time spent doing garbage collection, and number of threads that are running. These values are displayed in both the Profiler Control Panel and in the VM Telemetry Overview window (shown in [Figure 15-5](#)).

Figure 15-5. VM Telemetry Overview window

[[View full size image](#)]

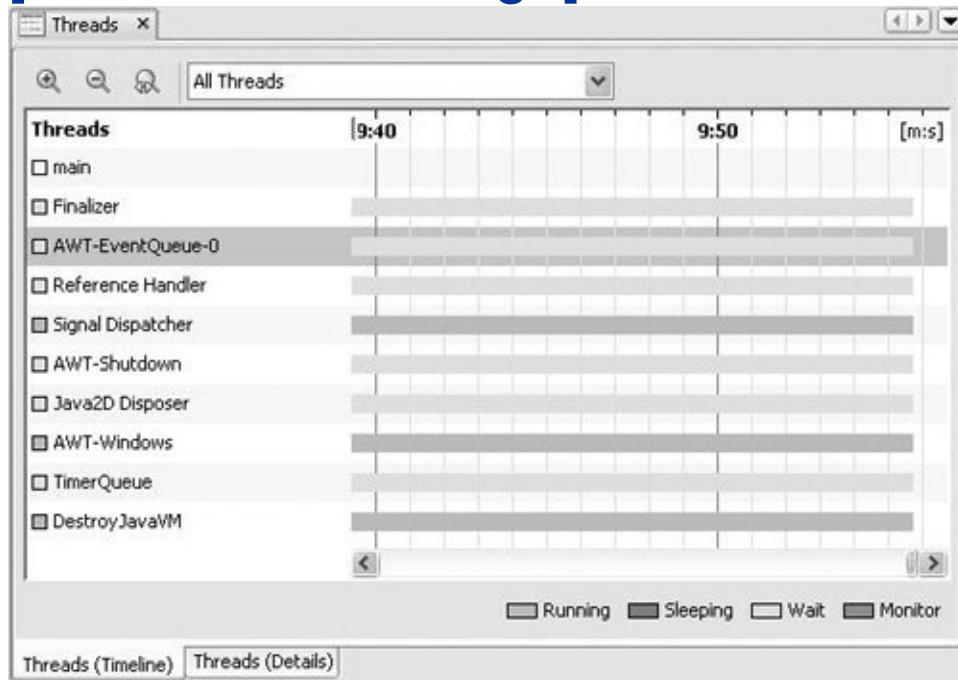


The VM Telemetry Overview window also displays the surviving generations on the heap. For an explanation of surviving generations, refer to the Surviving Generations and Memory Leaks section later in this chapter.

By default, detailed information about the state of each thread is also monitored. It is displayed in the Threads window (shown in [Figure 15-6](#)).

Figure 15-6. Threads window

[[View full size image](#)]



The thread states are shown with color coding:

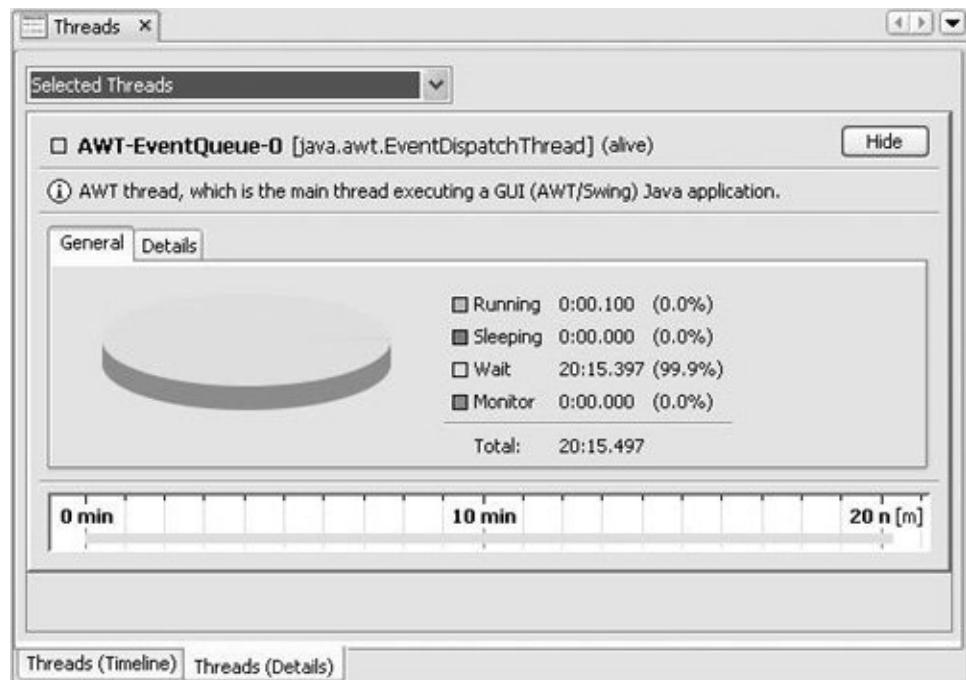
- **Green.** Thread is either running or is ready to run.
- **Purple.** Thread is sleeping in Thread.sleep()
- **Yellow.** Thread is waiting in a call to Object.wait()
- **Red.** The thread is blocked while trying to enter a synchronized method or block

The scroll bar can be used to scroll through time, allowing you to examine thread state going all the way back to when your application started. Click the Zoom In and Zoom Out icons (⊕ ⊖) to control the level of detail displayed by the Threads window.

Double-clicking a thread will switch the Threads window to its Details tab, where more information about the selected thread is displayed (shown in [Figure 15-7](#)).

Figure 15-7. Thread Details

[\[View full size image\]](#)

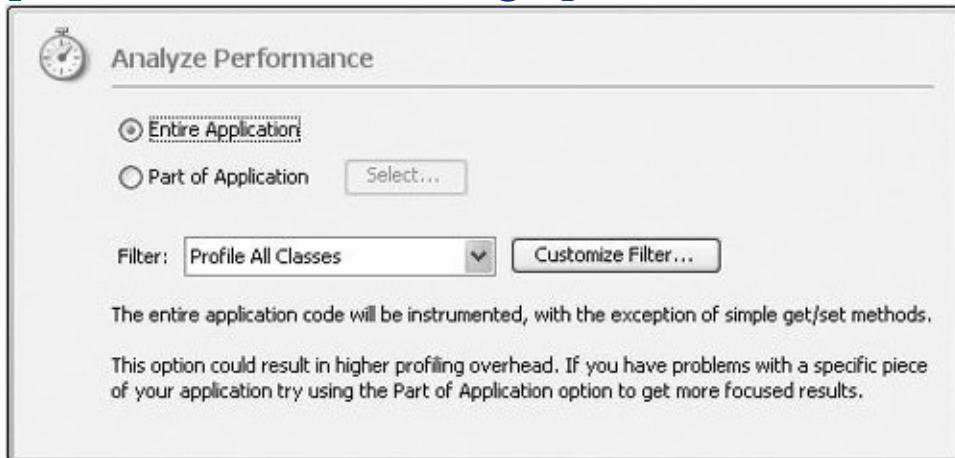


Analyzing Performance

Use the Analyze Performance task when you want to get detailed information about which methods in your application are using the most CPU time. The Analyze Performance task will also tell you how many times each method has been invoked. Clicking the Analyze Performance button expands it (shown in [Figure 15-8](#)).

Figure 15-8. Analyze Performance options

[[View full size image](#)]



Surviving Generations and Memory Leaks

To understand Surviving Generations, you have to think about the JVM's Garbage Collection process. Every time the garbage collector runs, each object either survives and continues to occupy heap memory, or it is removed and its memory is freed. If an object survives, then its age has increased by a value of 1. In other words, the age of an object is simply the number of garbage collections that it has survived. The value of Surviving Generations is the number of *different* object ages.

For example, assume there are several objects that were all allocated when your application first started. Further, there is another group of objects that were allocated at the mid-point of your application's run. And finally, there are some objects that have just been allocated, and have only survived one garbage collection. If the garbage collector has run 80 times, then all of the objects in the first group will have an age of 80, all of the objects in the second group will have an age of 40, and all of the objects in the third group will have an age of 1. In this example, the value of Surviving Generations is 3, because there are three different ages among all the objects on the heap: 80, 40, and 1.

In most Java applications, the value for Surviving Generations will eventually stabilize. This is because the application has reached a point where all long-lived objects have been allocated. Objects that are intended to have a shorter life span will not impact the Surviving Generations count because they will eventually be garbage collected.

If the Surviving Generations value for your application continues to increase as the application runs, it could be an indication of a memory leak. In other words, your application is continuing to allocate objects over time, each of which has a different age because it has survived a different number of garbage collections. If the objects were being properly garbage collected, then the number of different object ages would not be increasing.

You can either profile the entire application or just parts of it. Choosing to profile the entire application means that all called methods get instrumented. A large amount of instrumentation can slow performance dramatically, so this option is best used on smaller applications. An additional factor is that profiling your entire application will create a large amount of profiling information that you will have to interpret.



When profiling web or enterprise applications, an additional option is available when Entire Application is chosen: Profile Application Server Code. It is off by default. Turning it on dramatically increases profiling overhead. Similarly, when profiling a NetBeans module or module suite, an additional option is available: Profile the Entire IDE. It is off by default.

If you suspect that certain parts of your application are causing performance problems, then profiling just those parts may be the best approach. If you choose to profile only part of your application, then you must select one or more root methods. Select root methods by either clicking Select, or prior to starting the profiling session use the Tools | Add As Profiling Root Method menu option.

Regardless of whether you choose to profile your entire application or only parts of it, you can specify filters. Filters specify classes that should be either included or excluded from the instrumentation done by the profiler. The Filters list has three entries:

- **Profile All Classes.** An empty filter, so choose this entry when you want no filtering.
- **Quick Filter.** This entry displays a dialog box where you can specify a simple filter.
- **Exclude Java Core Classes.** This is a predefined custom filter that lists the following packages: `java..`, `javax..`, `sun..`, `sunw..`, `org.omg.CORBA`, `org.omg.CosNaming..`, `COM.rsa`. Select this entry to exclude the methods of all classes in the listed packages and any child packages.

To create your own custom filters, click Customize Filter. Filters are specified by package name; an asterisk (*) is not necessary but if it is used it must be at the end of the package name.

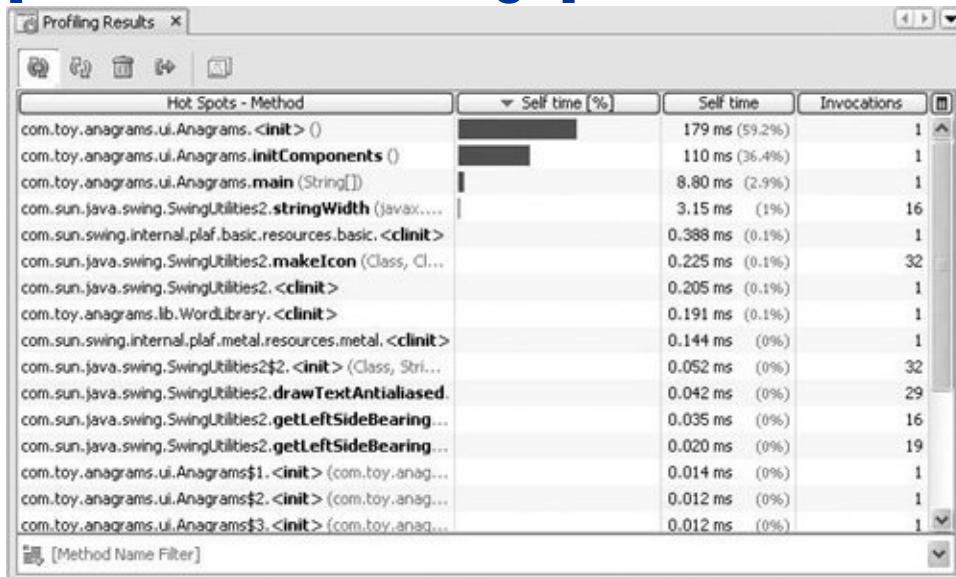
Clicking Run causes the IDE to start your application and then begin the profiling session.

Viewing Live Results

While your application is running you can watch the amount of time used by individual methods. Select Profile | View | Live Results or click the Live Results icon in the Profiler Control Panel to display the Profiling Results window (shown in [Figure 15-9](#)).

Figure 15-9. Live results while analyzing performance

[[View full size image](#)]



This window displays all methods that have been invoked at least once. The default sort order is by descending self time, so the methods in your application that are using the most time are displayed at the top of the list. The amount of time used is displayed in two columns, one with a graph to show the percentage of time spent in each method and the other with text that displays the raw time value and the percentage. The number of invocations is also shown. The profiler will update these values as your application runs.

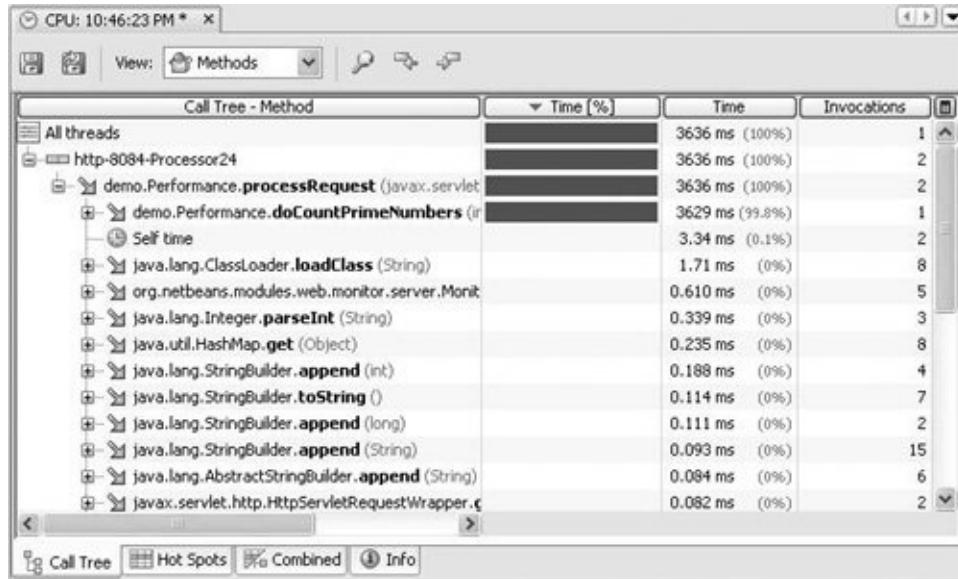
To change the sort order, click a column header. This will sort the table in descending order using the values from the column. Click again to sort in ascending order. Clicking the Hot Spots Method column will sort the table by package, class, and method name. To find a specific method more quickly, click on Method Name Filter at the bottom of the table and then enter the method name.

Taking a Snapshot of Results

To see more detailed information, select Profile | Take Snapshot of Collected Results or click the Take Snapshot icon in the Profiler Control Panel. The CPU snapshot window is displayed, with the time of the snapshot as its title (shown in [Figure 15-10](#)).

Figure 15-10. Results snapshot while analyzing performance

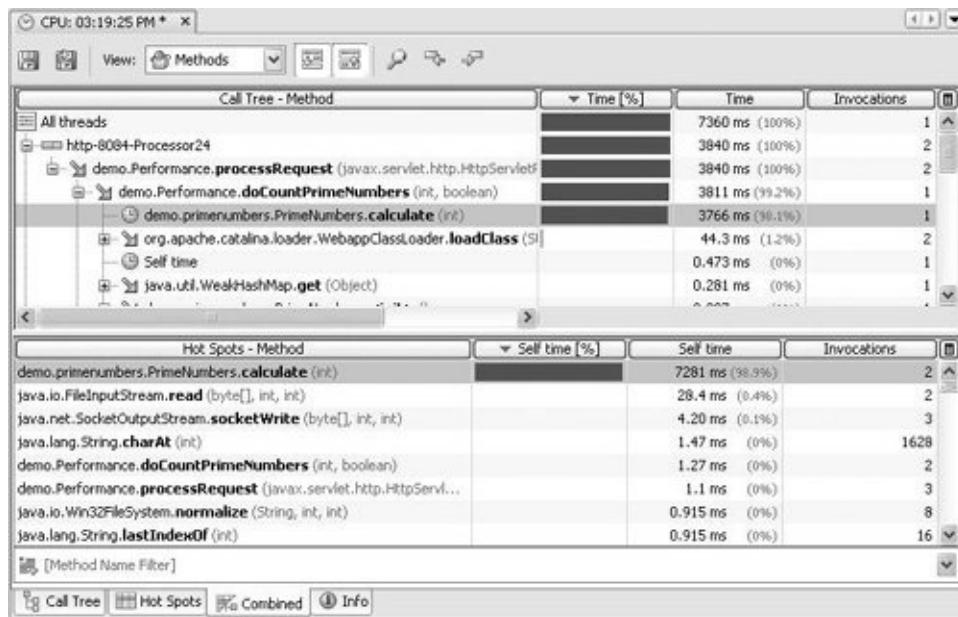
[\[View full size image\]](#)



The CPU snapshot window initially displays its Call Tree tab, which shows the call trees, organized by thread. If root methods were selected, then the Call Tree is displayed expanded so that those root methods can be seen. To switch to the Hot Spots view, just click the Hot Spots tab. It is usually helpful to see the execution path used by your application to get from one or more of the root methods to the hot spots. In order to do that easily, click the Combined tab. This tab shows both the Call Tree and the Hot Spots. Clicking a method in the Hot Spot list will find that method's entry in the Call Tree, making it easy to see the relationship between the root method and the hot spot (shown in [Figure 15-11](#)).

Figure 15-11. Combined view while analyzing performance

[[View full size image](#)]



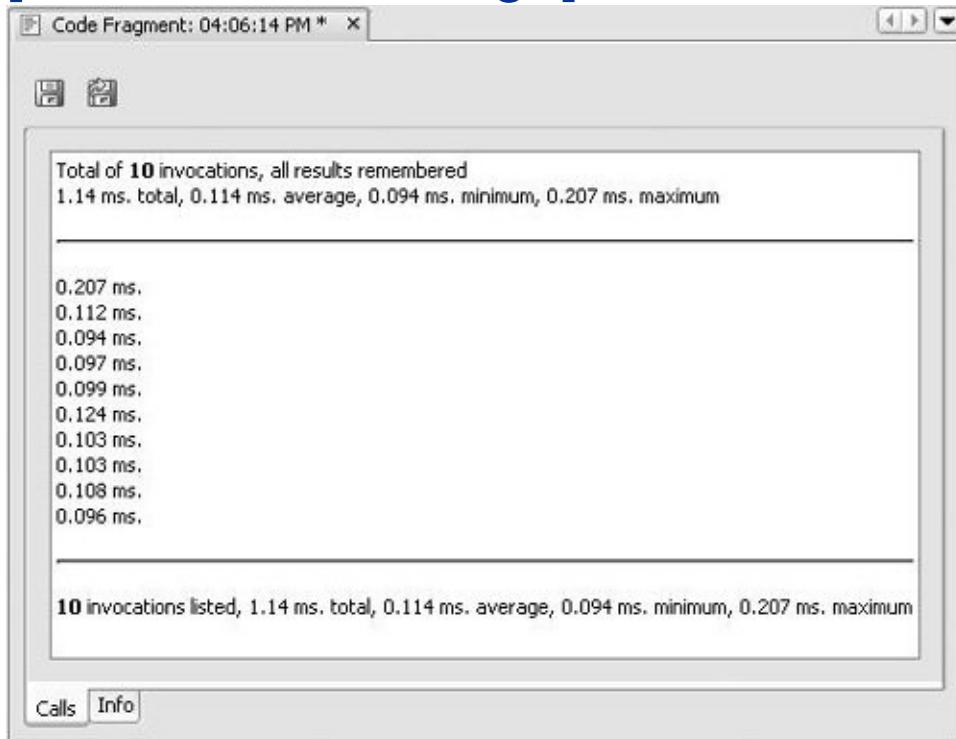
The Info tab displays a summary of the snapshot information: date, time, filter settings, etc. The icons along the top of the snapshot window allow you to save the snapshot, control the granularity of the snapshot (method, classes, or packages), and search the snapshot.

Analyzing Code Fragment Performance

When you want to measure the execution time of a single code fragment or method, use Analyze Code Fragment Performance. The value reported is very high level no details are provided to show self time or other methods that might have been called. The fragment or method to profile can be selected from the Profiling Task dialog box or by selecting Tools | Add as Profiling Code Fragment. Once profiling has begun, you can display either live results or you can look at snapshots of the collected profiling information (shown in [Figure 15-12](#)).

Figure 15-12. Analyzing Code Fragment results

[[View full size image](#)]

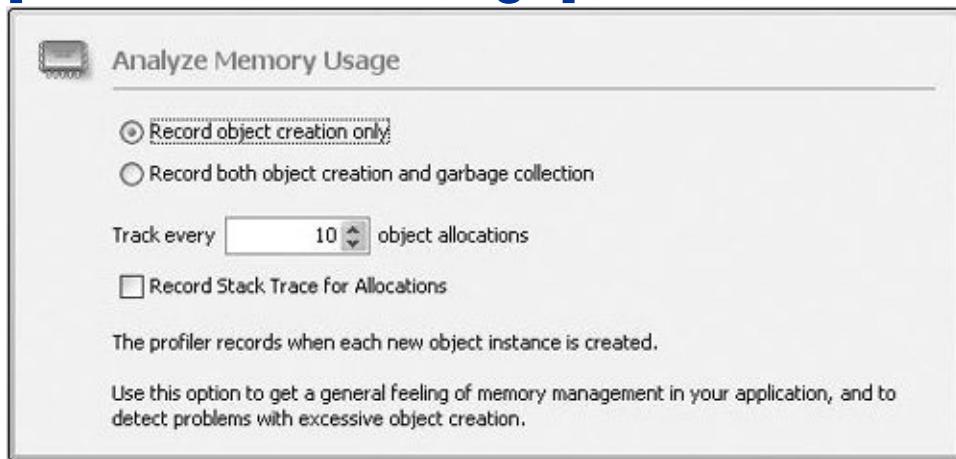


Analyzing Memory Usage

Use the Analyze Memory Usage feature to track the heap memory used by your application. If the JVM is reporting an OutOfMemoryError while running your application, then the profiler can help you determine the cause of the problem. When you click Analyze Memory Usage in the Select Profiling Task dialog box, the Analyze Memory Usage dialog box (shown in [Figure 15-13](#)) presents you with a choice: the profiler can track just object creation, or object creation and garbage collection.

Figure 15-13. Analyze Memory Usage options

[[View full size image](#)]



To get a general feel for object allocation, select object creation since it imposes less overhead. Statistics will be displayed that alert you to problems such as excessive object creation.

To track down potential memory leaks, choose to record both object creation and garbage collection. For each class used by

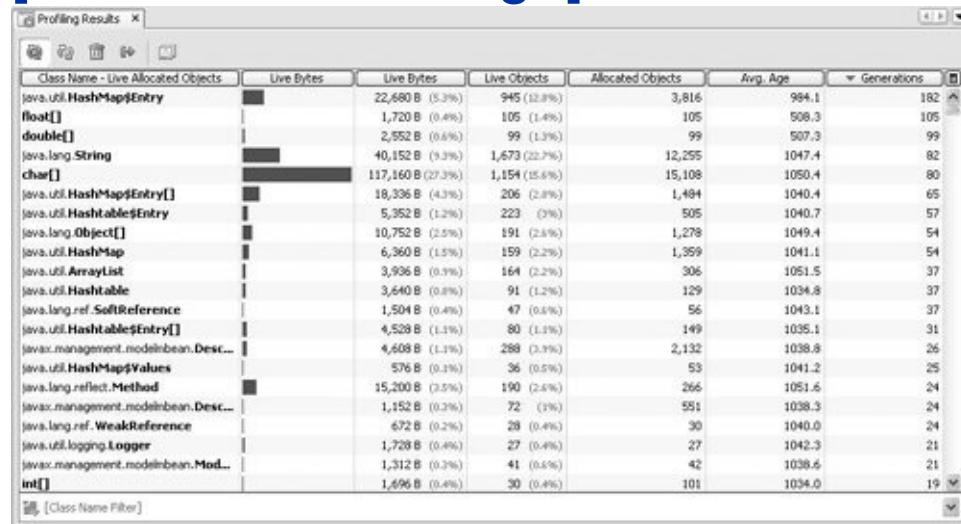
your application, only every tenth allocation will actually be tracked by the profiler. For most applications, this statistical approach dramatically lowers overhead without an impact on accuracy. You can use the spin control to change the number of allocations that are tracked, but keep in mind that lowering the value increases profiling overhead. In order for the profiler to report the methods that performed the allocations, you must select the Record Stack Trace for Allocations option.

Viewing Live Results

Once profiling begins, you can use the Live Results button to open a dynamic display of the heap contents (shown in [Figure 15-14](#)).

Figure 15-14. Live Results while analyzing memory usage

[[View full size image](#)]



The screenshot shows a Windows application window titled "Profiling Results". The main area is a table with the following columns: Class Name - Live Allocated Objects, Live Bytes, Live Objects, Allocated Objects, Avg. Age, and Generations. The table lists various Java classes and their memory usage statistics. The rows are as follows:

Class Name - Live Allocated Objects	Live Bytes	Live Objects	Allocated Objects	Avg. Age	Generations
java.util.HashMap\$Entry	22,680 B (5.3%)	945 (12.8%)	3,816	984.1	182
float[]	1,720 B (0.4%)	105 (1.4%)	105	508.3	105
double[]	2,952 B (0.6%)	99 (1.3%)	99	507.3	99
java.lang.String	40,152 B (9.3%)	1,673 (22.7%)	12,255	1047.4	82
char[]	117,160 B (27.3%)	1,154 (15.6%)	15,108	1050.4	80
java.util.HashMap\$Entry[]	18,336 B (4.3%)	206 (2.8%)	1,494	1040.4	65
java.util.Hashtable\$Entry	5,352 B (1.2%)	223 (3%)	505	1040.7	57
java.lang.Object[]	10,752 B (2.5%)	191 (2.6%)	1,278	1049.4	54
java.util.HashMap	6,360 B (1.5%)	159 (2.2%)	1,359	1041.1	54
java.util.ArrayList	3,936 B (0.9%)	164 (2.2%)	306	1051.5	37
java.util.Hashtable	3,640 B (0.8%)	91 (1.2%)	129	1034.8	37
java.lang.ref.SoftReference	1,504 B (0.4%)	47 (0.6%)	56	1043.1	37
java.util.Hashtable\$Entry[]	4,528 B (1.1%)	80 (1.1%)	149	1035.1	31
java.management.modelbean.Desc...	4,608 B (1.1%)	288 (3.9%)	2,132	1038.8	26
java.util.HashMap\$Values	576 B (0.1%)	36 (0.5%)	53	1041.2	25
java.lang.reflect.Method	15,200 B (3.5%)	190 (2.6%)	266	1051.6	24
java.management.modelbean.Desc...	1,152 B (0.3%)	72 (1%)	551	1038.3	24
java.lang.ref.WeakReference	672 B (0.2%)	29 (0.4%)	30	1040.0	24
java.util.logging.Logger	1,728 B (0.4%)	27 (0.4%)	27	1042.3	21
java.management.modelbean.Mod...	1,312 B (0.3%)	41 (0.6%)	42	1038.6	21
int[]	1,696 B (0.4%)	30 (0.4%)	101	1034.0	19

The columns displayed are:

- **Allocated Objects.** The number of objects that the profiler is tracking. In this example, there are 105 instances of float[] that are being tracked. By default this number will be approximately ten percent of the objects actually allocated by your application. By monitoring only a subset of the created objects the profiler is able to dramatically reduce the overhead it places on the JVM, which then allows your application to run at close to full speed.
- **Live Objects.** The number of the Allocated Objects that are currently on the heap and are therefore taking up memory.
- **Live Bytes.** Shows the amount of heap memory being used by the Live Objects. One column displays a graph, the other displays text.
- **Avg.Age.** Average age of the Live Objects. The age of each object is the number of garbage collections that it has survived. The sum of the ages divided by the number of Live Objects is the Avg. Age.
- **Generations.** Calculated using the Live Objects. The age of an object is the number of garbage collections it has survived. The Generations value is the number of different ages for the Live Objects. It is the same concept as the surviving generations, only applied to a single class; see the Surviving Generations and Memory Leaks section earlier in this chapter.

To change the sort order, click a column header. This will sort the table in descending order using the values from the column. Click again to sort in ascending order. Sorting the table by Generations can frequently help identify classes that are the

source of memory leaks. This is because an increasing value for Generations typically indicates a memory leak.



Once you have the display sorted so that the classes of interest are at the top, if you chose to track object creation and garbage collection then you can right-click an entry and choose Stop Profiling Classes below this Line to reduce profiling overhead.

Taking a Snapshot of Results

In order to see which methods in your application are allocating objects, you must take a snapshot. Use the Take Snapshot button in the Profiler Control Panel or choose Profile | Take Snapshot of Collected Results. The resulting window has a tab labeled Memory Results which contains the same information as the Live Results window. Right-click a class and then select Show Allocation Stack Traces to switch to the Allocation Stack Traces tab. Its display is similar, only the first column displays method names (shown in [Figure 15-15](#)).

Figure 15-15. Results snapshot while analyzing memory usage

[[View full size image](#)]



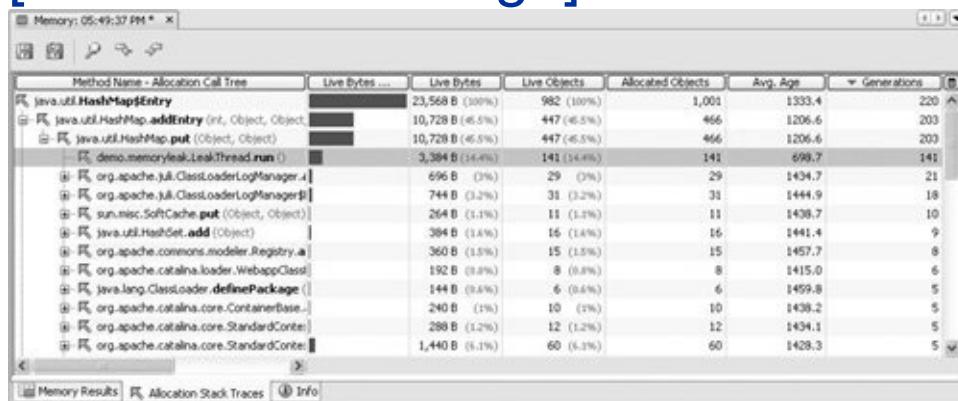


You can right-click an entry in the Live Results window and select Take Snapshot and Show Allocation Stack Traces to quickly open Memory Results with the Allocation Stack Traces tab displayed.

The listed methods allocated one or more instances of the selected class. You can use the displayed statistics to help narrow down which of the methods is allocating class instances that are causing memory leaks. In the example shown, the `addEntry()` and `createEntry()` methods are both allocating instances of `HashMap$Entry`. Note that the Generations value for the allocations done by `addEntry()` is much higher than that for `createEntry()`. This indicates that `addEntry()` is where leaking instances of `HashMap$Entry` are being allocated. You can click the icon next to a method to see the different execution paths that called that method (shown in [Figure 15-16](#)).

Figure 15-16. Execution paths for a method

[[View full size image](#)]



The `addEntry()` method was called by `put()`, which in turn was called by several different methods. The calls from one of those methods, `LeakThread.run()`, resulted in allocations with a very high Generations value, indicating that it is a likely source of a memory leak. It should be inspected to see if perhaps it is adding entries to a `HashMap` that are never being removed.

Attaching the Profiler to a JVM

The NetBeans Profiler can profile applications that are not started by the IDE. In other words, it can be attached to a JVM. To use this feature, select Profile | Attach Profiler. The Profiling Task dialog box is displayed with a list of projects at the top. To profile an application that does not have a corresponding project, select the <Global Attach> entry from the list and click Attach Wizard. This will start the Attach Wizard (shown in [Figure 15-17](#)).

Figure 15-17. Attach Wizard dialog box



The Attach Wizard will step you through the process of attaching the profiler to a JVM that was started from outside the IDE.

The Attach Profiler feature can also be used to profile free-form web applications (those that are based on a build script created

and maintained outside of the IDE). If your project is deployed to the Tomcat or Sun Java System Application Server, use the following steps to profile it:

- Open the Runtime window, right-click the server instance's node and then select Start in Profile Mode.
- Select Profile | Attach Profiler.
- Choose a project and a profiling task, then click Attach.

If your free-form web application is deployed to another type of server, the Attach Wizard can modify the script used to start that server or it can provide documentation that shows you how to make the changes. To use the Attach Wizard for profiling a free-form web application, select the free-form project from the list of projects displayed at the top of the Profiling Task dialog box. When prompted by the Attach Wizard to select a Target Type, select J2EE Web/App Server.

Chapter 16. Integrating Existing Ant Scripts with the IDE

- [Creating a Free-Form Project](#)
- [Mapping a Target to an IDE Command](#)
- [Setting up the Debug Project Command for a General Java Application](#)
- [Setting up the Debug Project Command for a Web Application](#)
- [Setting up Commands for Selected Files](#)
- [Setting up the Compile File Command](#)
- [Syntax for Mapping Targets to Commands](#)
- [Setting up the Run File Command](#)
- [Setting up the Debug File Command](#)
- [Setting up the Debugger's Apply Code Changes Command](#)
- [Setting up the Profile Project Command for a General Java Application](#)
- [Changing the Target JDK for a Free-Form Project](#)
- [Making a Custom Menu Item for a Target](#)

- [Debugging Ant Scripts](#)

THE USER INTERFACE FOR STANDARD PROJECTS IN NETBEANS IDE is designed to handle common development scenarios, to be easy to use, and to encourage good programming practices (such as modular design with no circular dependencies). It is particularly well suited to creating projects from scratch.

However, the standard user interface does not cover all scenarios, particularly for projects originally developed in other environments. If this is your case, you can take advantage of the IDE's tight integration with Ant to customize the IDE to work with your existing Ant build script.

If you already have your own build script and do not want to (or cannot) re-create it through the IDE, you can set up the IDE to use that build script by creating a free-form project. Free-form projects also might be preferable to standard projects if you create multiple outputs from individual source roots or if there is anything too restrictive in standard projects.



Before committing to using your existing build script with the IDE, carefully consider whether you really need to use your own Ant script, and make sure that it is not possible to replicate your existing build processes with a combination of standard IDE projects, because standard IDE projects will likely be easier to maintain on a long-term basis.

The IDE's project system is based on Ant to the degree that even incremental commands (such as for compiling a single file) and other commands that you might specifically associate with IDE use (such as debugging) are defined in the build script. Build targets for these commands are generated by default in

standard projects but not in free-form projects. In free-form projects, it is left up to you to write the targets in whatever way will work with your project.

Using a build script that was created outside of the IDE entails the following steps:

- Creating a project in the IDE using one of the free-form (With Existing Ant Script) templates.
- Mapping key existing build targets to the IDE commands that correspond to them (such as for compiling and running applications and running tests). You can create these mappings in the New Project wizard as you set up the project or later in the Project Properties dialog box.
- Registering classpath items (such as external source roots or JAR files) in the New Project wizard (when creating the project) or in the Project Properties dialog box (after creating the project) so that IDE-specific features such as code completion and refactoring work correctly.
- Creating new build targets for commands that are IDE-specific and creating mappings to these targets in the project's `project.xml` file. For some commands, the IDE helps you by offering to generate a target (and the mapping in the `project.xml` file) when you first run the given command in the IDE, but you might have to modify the target to get it to work correctly for your project. For other commands, you might have to write the target from scratch and manually create the mapping in the project's `project.xml` file. In NetBeans IDE 5.0, the IDE offers to generate targets and mappings for the Debug Project and Compile File commands, but leaves it to you to create targets for commands such as Run File, Test File, and Debug File. Most likely, post-5.0 versions of the IDE will offer target generation for the latter commands.

Creating a Free-Form Project

- 1.** Choose File | New Project.
- 2.** Select a project category (such as General, Web, or Enterprise) and then select the With Existing Ant Script template for that category.
- 3.** In the Name and Location page of the wizard, fill in the Location field with the folder that contains the various elements of your project, such as your source folder, test folder, and build script.

If the build script is at the top level of the folder that you have specified, the rest of the fields are filled in automatically. If the build script is not found, fill in the Build Script field manually.

Note that for NetBeans IDE 5.0, the folder for your compiled classes should also be in this folder. If it isn't, some editing features, such as refactoring, might not work correctly. This should be fixed in a later IDE release.

If you wish, you can change the other fields, such as Project Folder (which determines where the IDE stores metadata for the project).

- 4.** In the Build and Run Actions page, specify targets for the listed IDE commands so that the IDE knows which target in your script to run when you choose the command in the IDE. You can click the combo box arrow next to each command to select from a list of all targets in the build script, or you can type a target in the combo box manually. If the IDE finds a likely target for the command, it is filled in automatically, although you can change it if you like. If you leave any of the commands blank, you can later fill them in

manually outside of the wizard.

If the build script imports targets from other build scripts, those targets are not shown in the combo box list, although you can type one of those targets manually.

Not all available IDE commands are given here. You can provide mappings for other IDE commands (such as Compile File) directly in the `project.xml` file. See Mapping a Target to an IDE Command later in this chapter.

5. (For Web projects only) In the Web Sources page, specify the folder that contains your web pages, fill in the context path for the application, and mark the J2EE Specification level.
6. In the Source Package Folders page, specify all of the folders that contain your top-level packages. For example, if the package structure of one of your source roots begins with `com`, choose the folder that contains `com` (such as `src`). Similarly, if you have any test packages, you can specify them here in the Test Package Folders area.

On this page, also be sure to set the Source Level to the appropriate JDK version. Even if this is already accounted for in your Ant script, you need to set the source level here so that IDE-specific features, such as proper Source Editor syntax highlighting and code completion, work correctly.

7. In the Java Sources Classpath page (and, for Web projects, also in the Web Sources Classpath page), specify any libraries or sources that each source root is compiled against. Doing this hooks up IDE features such as code completion and refactoring to your project. You do not have to specify the JDK on this page.



in the Project Properties dialog box, so you do not have to fill in each value immediately. For example, if you still have to write a target for an IDE command, you can later map the target to the command in the Build and Run page of the Project Properties dialog box. To open the Project Properties dialog box, open the Projects window, right-click the project's main node, and choose Properties.

Mapping a Target to an IDE Command

When you use the Java Project With Existing Ant Script project template, the New Project wizard enables you to map specific build targets to IDE commands, including the following:

- Build Project
- Clean Project
- Generate Javadoc
- Run Project
- Test Project

You can also use the Project Properties dialog box (Build and Run page) to map targets to these commands. Other commands (such as Debug Project, Compile File, Run File, Debug File, and Apply Code Changes) need to have targets created for them and then be mapped in the `project.xml` file if you want them to work in the IDE.

If you let the IDE generate an Ant target for a command, the IDE handles this mapping automatically. In NetBeans IDE 5.0, the IDE offers to generate targets for the Debug Project and Compile File commands the first time you run those commands. In future versions of the IDE, target generation will be offered for other commands as well.

See [Table 16-1](#) for a list of commands that you can map to your build script. The IDE Action column gives the code name for the command that you use when mapping a build target to the command.

Table 16-1. IDE Commands and Corresponding Action Names Used in the `project.xml` File

Command	IDE Action
Build Project	<code>build</code>
Clean and Build Project	<code>rebuild</code>
Compile Selected Files	<code>compile.single</code>
Clean Project	<code>clean</code>
Run Project	<code>run</code>
Run Selected File	<code>run.single</code>
Redeploy Project (for web applications)	<code>redeploy</code>
Test Project	<code>test</code>
Test File	<code>test.single</code>
Debug Test For File	<code>debug.test.single</code>
Debug Project	<code>debug</code>
Debug File	<code>debug.single</code>
Apply Code Changes	<code>debug.fix</code>
Step Into	<code>debug.stepinto</code>
Generate Javadoc	<code>javadoc</code>

The next several topics provide examples of how to create Ant targets for specific commands and then map them to the IDE.

The IDE comes bundled with XML schemas for the `project.xml` file and automatically validates them every time you edit and save them. If you make invalid changes to a `project.xml` file, the IDE reports the errors in the Output window.

If you would like to inspect the schemas yourself, you can view them online. See [Table 16-2](#) for a list of the schemas used.

Table 16-2. Free-Form Project Schema

Schema	Description
www.netbeans.org/ns/freeform-project/1.xsd	Defines the <code><general-data></code> part of the <code>project.xml</code> file for all free-form project types.
www.netbeans.org/ns/freeform-project-java/1.xsd	Defines the <code><java-data></code> part of the <code>project.xml</code> file for all free-form project types.
www.netbeans.org/ns/freeform-project-web/1.xsd	Defines the <code><web-data></code> part of the <code>project.xml</code> file for web applications.

Setting up the Debug Project Command for a General Java Application

To get debugging to work with a free-form project, you need to:

- Make sure that the target you use for compiling specifies `debug="true"` when calling the `javac` task.
- Create a mapping in the IDE between the project's sources and the project's outputs so that the debugger knows which sources to display when you are stepping through the running program.
- Add a target to your Ant script for the command, making sure that all necessary path elements for the command are defined, either in the build script or in a `.properties` file that is called by the script.

Mapping the Project's Sources to Its Outputs

In NetBeans IDE 5.0 free-form projects, the IDE does not automatically know which sources are associated with compiled classes that you run. Therefore, to get the IDE's debugging features to work, you need to create this mapping between the sources and the outputs.

To map a free-form project's sources to its outputs:

1. In the Projects window, right-click the project's node and choose Properties.
2. In the Project Properties dialog box, select the Output node.

3. In the right pane, click the Add JAR/Folder button and navigate to the folder or JAR file that contains the compiled classes corresponding to the source root selected in the Source Packages Folder field.

If you have multiple source roots, repeat this step for each source root listed in the Source Packages Folder field.



When you create your debug target, the outputs you specify here will need to be referenced as part of the `classpath` attribute of the `jpdastart` (or `nbjpdacconnect`) task.

Creating the Debug Target

You can have the IDE generate a debug target for you by running the Debug Main Project command (assuming you have set the free-form project as your main project) or Debug Project command and then clicking the Generate button in the dialog box that appears.

When you generate the target, it appears in a file called `ide-file-targets.xml`, which imports your main build script. This enables you to have IDE-only targets separate from other targets but still allow the IDE-specific targets to reference targets and properties in your main build script.

If you already have the Run Project command mapped, the generated Debug Project target should look something like the following:

```
<target name="debug-nb">
    <nbjpdastart addressproperty="jpda.address" name="jpda">
```

```

        transport="dt_socket">
        <classpath path="build"/>
</nbjpdastart>
<java classname="MainClass" classpath="build" fo
        <jvmarg value="-Xdebug"/>
        <jvmarg value="-Xnoagent"/>
        <jvmarg value="-Djava.compiler=none"/>
        <jvmarg value="-Xrunjdwp:transport=dt_socket
                        address=${jpda.address}"/>
    </java>
</target>

```

In many cases, the generated target will work without modification. If the target does not work or does not work the way you want it to, you can modify it by hand. See [Table 16-3](#) for further details and some things to look out for.

Table 16-3. Details of the Debug Target for a General Java Application

Target, Task, Attribute, or Property	Description
<code>depends</code>	An optional attribute of the target, where you specify other targets that need to be run before the current target is run. This attribute is not specified in the generated target, but you might want to add it.
<code>netbeans.home</code>	Ant property that is loaded by any instance of Ant that runs inside of the IDE. The <code>if="netbeans.home"</code> attribute ensures that the target is run only if it is called from within the IDE. This attribute is not included in the generated debug target, but might be useful in other targets that you include directly in your build script.
<code>nbjpdastart</code>	A special task bundled with the IDE to debug programs within the JPDA debugger.

<code>addressproperty</code>	An attribute of <code>nbjpdastart</code> that defines the property that holds the port that the debugger is listening on. (The IDE automatically assigns the port number to the property.) The value of the property that is defined there (in the case of the examples given on the previous page, <code>jpda.address</code>) is passed as the value for the <code>address</code> suboption of the <code>-Xrunjdwp</code> option.
<code>transport</code>	An attribute specifying the debugging transport protocol to use. You can use <code>dt_socket</code> on all platforms. On Windows machines, you can also use <code>dt_schem</code> .
<code>classpath</code>	An attribute of both the <code>nbjpdastart</code> and the <code>java</code> tasks that represents the classpath used for debugging the application. When generating the debug target, the IDE fills in the classpath provided by the Run Project target, if possible.
<code>sourcepath</code>	An optional attribute of <code>nbjpdastart</code> used to specify the explicit location of source files that correspond to JAR files in your classpath. If you have associated your sources with JAR files in the Output panel of the project's Project Properties dialog box or in the IDE's Library Manager, you should not need to set this attribute. This attribute is not included in the generated target.
<code>fork</code>	Attribute of the <code>java</code> task that determines whether the debugging process is launched in a separate virtual machine. For this target, the value must be <code>True</code> .
<code>classname</code>	Attribute of the <code>java</code> task. It points to the class that the debugger executes. For the Debug Project target, this attribute should be the fully qualified name of the main class of the project. When generating the debug target, the IDE fills in the classname provided by the Run Project target, if possible.
<code>jvmarg</code>	Parameter of the <code>java</code> element for providing arguments to the JVM. The arguments provided in the example are typical for debugging Java SE applications with the JPDA debugger.

If you have generated the debug target without having previously designated a run target in the project, the generated target will have some gaps that you need to fill in. Such a target might look something like the following:

```
<target name="debug-nb">
    <path id="cp">
        <!-- TODO configure the runtime classpath for -->
        <!-- -->
    </path>

    <nbjpdastart addressproperty="jpda.address" name="Note|
        transport="dt_socket">
        <classpath refid="cp"/>
    </nbjpdastart>
    <!-- TODO configure the main class for your project he
    <java classname="some.main.Class" fork="true">
        <classpath refid="cp"/>
        <jvmarg value="-Xdebug"/>
        <jvmarg value="-Xnoagent"/>
        <jvmarg value="-Djava.compiler=none"/>
        <jvmarg value="-
Xrunjdwp:transport=dt_socket,address=${jpda.address}"/
    </java>
</target>
```

The `TODO` comments mark places where you need to fill in the runtime classpath and the project's main class. For the main class, enter the fully-qualified classname. For the classpath, you could place `pathelement` elements within the provided `path` element. For example, you could use the `location` attribute of

`pathelement` to specify the location of folders relative to your project directory (usually the one that contains the `build.xml` file) as in the sample below:

```
<path id="cp">
    <pathelement location="libs">
        <pathelement location="classes">
</path>
```

If your project is more complex, you can nest `path` elements, where the `path` attribute of `pathelement` references a different `path` element, as in the following example:

```
<path id="run.classpath">
    <pathelement path="${javac.classpath}">
        <pathelement path="${build.classes.dir}">
</path>
```

Setting up the Debug Project Command for a Web Application

To get debugging to work with a free-form web project, you need to:

- Make sure that the target you use for compiling specifies `debug="true"` when calling the `javac` task.
- Create a mapping in the IDE between the project's sources and the project's outputs so that the debugger knows which sources to display when you are stepping through the running program.
- Add a target to your Ant script for attaching the debugger to a running web application, making sure that all necessary path elements for the command are defined, either in the build script or in a `.properties` file that is called by the script.
- Make sure your server is started in debug mode.
- Make sure your web application is already deployed. (To be able to deploy your application with the IDE's Run Project command, you need to have a target in your build script for deploying your web application and have that target mapped to the Deploy command. You can provide this mapping in the wizard when creating the project or on the Build and Run page of the Project Properties dialog box.)

You can have the IDE generate a debug target for you by running the Debug Main Project command (assuming you have set the free-form project as your main project) or Debug Project command and then clicking the Generate button in the

dialog box that appears.

When you generate the debug target, it appears (along with four supporting targets) in a file called `ide-file-targets.xml`, which imports your main build script. This enables you to have IDE-only targets separate from other targets but still allow the IDE-specific targets to reference targets and properties in your main build script. You can find this file in the project's `nbproject` folder, which you can access through the Files window.

The generated targets should look something like the following:

```
<target name="-load-props">
    <property file="nbproject/debug.properties"/>
</target>
<target name="-check-props">
    <fail unless="jpda.session.name"/>
    <fail unless="jpda.host"/>
    <fail unless="jpda.address"/>
    <fail unless="jpda.transport"/>
    <fail unless="debug.sourcepath"/>
    <fail unless="client.url"/>
</target>
<target depends="-load-props, -check-props" name="-ini">
<target depends="-init" if="netbeans.home" name="debug">
    <nbjpdacconnect address="${jpda.address}" host="${jpda.host}" port="4444" name="${jpda.session.name}" transport="${jpda.transport}"/>
    <sourcepath>
        <path path="${debug.sourcepath}"/>
    </sourcepath>
</nbjpdacconnect>
    <antcall target="debug-display-browser"/>
</target>
<target name="debug-display-browser">
    <nbbrowse url="${client.url}"/>
</target>
```

In addition, a file called `debug.properties` is generated. This file should look something like the following:

```
jpda.session.name=My_Project
jpda.host=localhost

# Sun Java System Application Server using shared memo
# jpda.address=localhost4848
# jpda.transport=dt_shmem

# Sun Java System Application Server using a socket
# jpda.address=9009
# jpda.transport=dt_socket

# Tomcat using shared memory (on Windows)
# jpda.address=tomcat_shared_memory_id
# jpda.transport=dt_shmem

# Tomcat using a socket
jpda.address=11555
jpda.transport=dt_socket

src.folders=src
web.docbase.dir=web

# you can change this property to a list of your source
debug.sourcepath=${src.folders}:${web.docbase.dir}

# Client URL for Tomcat
client.url=http://localhost:8084/myproject

# Client URL for Sun Java System Application Server
# client.url=http://localhost:8080
```

You should be able to get the generated target to work merely by making appropriate edits to the `debug.properties` file. After being customized for your environment, the generated target can be used in your project. See [Table 16-4](#) for further details on the `debug.properties` file and [Table 16-5](#) for details on the debug targets.

Table 16-4. Details of the `debug.properties` File for a Web Application

Debug Property	Description
<code>jpda.session.name</code>	The name that appears in the Sessions window when you debug the application.
<code>jpda.host</code>	The hostname of the machine that the debugged application is running on.
<code>jpda.address</code>	The port that the debugger is listening on.
<code>jpda.transport</code>	The JPDA debugging transport protocol to use. You can use <code>dt_socket</code> on all platforms. On Windows machines, you can also use <code>dt_schem</code> , though the IDE and the debugged application would both have to be running on the same machine.
<code>src.folders</code>	The location of your Java source files.
<code>web.docbase.dir</code>	The location of your web root.
<code>debug.sourcepath</code>	The location of the sources to be referenced when debugging. By default, the value is a reference to the <code>src.folders</code> and <code>web.docbase.dir</code> properties.
<code>client.url</code>	The web page to be opened in the default browser that is specified by the IDE.

Table 16-5. Details of the Debug Target for a Web Application

Target, Task, Attribute, or Property	Description
<code>depends</code>	Attribute where you specify targets that need to be run before the current target is run.
<code>netbeans.home</code>	Ant property that is loaded by any instance of Ant that runs inside of the IDE. The <code>if="netbeans.home"</code> attribute ensures that the target is run only if it is called from within the IDE.
<code>nbjpdaconnect</code>	A special task bundled with the IDE to enable attaching the JPDA debugger to a running application.
<code>host</code>	An attribute of <code>nbjpdaconnect</code> that specifies the hostname of the machine that the debugged application is running on. In this example, the <code>jpda.host</code> property is used. The value of this property is defined in the <code>debug.properties</code> file that is referenced by the build script.
<code>address</code>	An attribute of <code>nbjpdaconnect</code> that specifies the port that the debugger is listening on. In this example, the <code>jpda.address</code> property is used. The value of this property is defined in the <code>debug.properties</code> file.
<code>transport</code>	An attribute specifying the JPDA debugging transport protocol to use. You can use <code>dt_socket</code> on all platforms. On Windows machines, you can also use <code>dt_schem</code> , although the IDE and the debugged application would both have to be running on the same machine.
<code>classpath</code>	An optional attribute of <code>nbjpdaconnect</code> that represents the classpath used for debugging the application.
<code>sourcepath</code>	

An attribute of `nbjpdaconnect` used to specify the explicit location of source files that correspond to JAR files in your classpath.

nbbrowse

Element that specifies a web page to be opened in the default browser that is specified by the IDE.

In this example, the `client.url` property is used. The value of this property would need to be defined elsewhere in the build script or in a `.properties` file that is referenced by the build script.

Setting up Commands for Selected Files

To get file-specific commands (such as Compile File, Run File, and Debug File) to work in the IDE, you need to do the following:

- Add a target to your Ant script for the command, making sure that all necessary path elements for the command are defined, either in the build script or in a `.properties` file that is called by the script. In NetBeans IDE 5.0, you can have the IDE generate a target for the Compile File command, but you have to write the others from scratch.
- Map the target to the IDE through the project's `project.xml` file, and include a `context` element to provide the IDE a way of passing the currently selected files to the Ant script. If the IDE generates the target for you, it performs this step for you as well.
- Define any properties that are needed in the project's `project.xml` file, either in the `project.xml` file or in a `properties` file that is referenced from the `project.xml` file.

See the next few topics for examples of how to set up the commands.

Setting up the Compile File Command

To be able to compile selected files in a free-form project in the IDE, you need to create an Ant target for the command.

Generating a Skeleton Target for Compile File

You can create this target by right-clicking a `.java` file and choosing Compile File. The IDE then offers to create a target for you. If you click Generate, the IDE generates a skeleton target in the `ide-file-targets.xml` file and creates a mapping in the `project.xml` file between the target and IDE command. The target might look something like this:

```
<target name="compile-selected-files-in-src">
    <fail unless="files">Must set property 'files'</f
    <mkdir dir="build"/>
    <javac destdir="build" includes="${files}" source=
        srccdir="src">
        <classpath path="resources"/>
    </javac>
</target>
```

In this example, the `files` property picks up the files that you have selected in the IDE. The value of `files` is passed from the `project.xml` file when you choose the Compile File command. If no files are selected in the IDE when Compile File is chosen (and, therefore, no value is passed to `files` from the `project.xml` file), the target does not complete successfully.



In NetBeans IDE 5.0, the IDE also offers to generate the Compile File target for you the first time you try to choose

Compile Package on a package. However, the generated target does not actually work on packages. To compile a package, you can use the Ctrl or Shift key to select all of the classes in the package and then right-click and choose Compile Files.

Syntax for Mapping Targets to Commands

Targets in your build script are mapped to IDE commands in the `project.xml` file using the `<action>` element. These mappings are entered into the `project.xml` file automatically when you specify targets for commands (in the New Project wizard or the Project Properties dialog box) or when you have the IDE generate a target for you. If you write a target from scratch, you have to enter the mapping manually.

See [Table 16-6](#) for a description of the parts of the `<action>` element in the `project.xml` file.

Table 16-6. Details of the `project.xml <action>` Element

Target, Task, or Property	Description
<code>context</code>	Parameter that the IDE uses to collect information about the files that the command is to be run on.
<code>property</code>	Parameter that defines the name of the property that is passed the names of the currently selected files in the IDE when the command is chosen. A target can then reference this property to determine what files to run the command on. For example, the target that you can have the IDE generate for the Compile File command references the <code>files</code> property to determine which files are to be compiled.
<code>folder</code>	Parameter that enables you to specify the directory in which the target is enabled. In the <code>compile-selected-files-in-src</code> example on the previous page, the value is provided as a reference to the <code>src</code> property.
<code>pattern</code>	Parameter that contains a regular expression to limit the kinds of files that the target can be run

on. In this example, only files with the `.java` extension are passed to the target.

format

Parameter that specifies the form in which the selected files are passed to the target. Possible values for this element are

`relative-path`passes the filename with its path relative to the folder specified by the `folder` element

`relative-path-noext`like `relative-path` except that the filename is passed without its extension

`java-name`like `relative-path-noext` except that periods (.) are used instead of slashes to delimit the folders in the path

`absolute-path`passes the filename with its absolute path

`absolute-path-noext`like `absolute-path` except that the filename is passed without its extension

arity

Parameter that specifies whether single or multiple files can be passed to the target. Possible values are

`<separated-files>delimiter</separated-files>`
Multiple files can be passed.

`<one-file-only>`Only one file can be passed.

Handling Properties in the `project.xml` File

In the example in the preceding section, the `src` property is referenced from the `project.xml` file. This property needs to be defined, either in a file referenced by the `project.xml` file or

directly in the `project.xml` file.

In the `project.xml` file, properties are defined in the `<properties>` element, which belongs between the `<name>` and `<folders>` elements. Within the `<properties>` element, use the `<property>` element and its `name` attribute to define an individual property, or use the `<property-file>` element to designate a `.properties` file. Note that this syntax is different from Ant's syntax for defining properties.

After completing the New Project wizard, where you have specified the build script to use, something like the following is generated in your `project.xml` file:

```
<properties>
    <property name="project.dir">C:\MyNBProjects\SampleF
    <property name="ant.script">${project.dir}/build.xml
</properties>
```

You can add more property references or property file references within the `<properties>` element. Because the `src` property in this example is likely a property that can also be used in your build script, it might be useful to set the property in one place and let both the build script and `project.xml` file use it. A reference to a `.properties` file from the `project.xml` file would look something like the following line:

```
<property-file>${project.dir}/MyProject.properties</property-file>
```



File paths that are referenced from the `project.xml` file are relative to the project folder. For path references to work the same for both the `project.xml` file and the build script, the build script needs to be in the project folder (which is actually the folder that *contains* the `nbproject` folder).

If the build script is in a different folder, you might solve the path discrepancy by moving the `project.dir` property in the example above to a properties file that is common for both the `project.xml` file and build script, and use that property in the values of other properties that you define for your classpath, source path, and so on. For example, you could create a property to specify the location for compiled class files and give it the value

```
 ${project.dir}/build/classes.
```

Setting up the Run File Command

To be able to run a selected file in a free-form project in the IDE, you need to create an Ant target for the command and then map that target in the project's `project.xml` file.



If you are using a post-5.0 version of NetBeans IDE, try to run the Run File command before writing a target. The IDE might offer to generate the target for you, which could save you some time.

Creating the `run-selected-file` Target

Following is a sample target for running selected files:

```
<target name="run-selected-file"
depends="compile-selected-files-in-src"
description="Run Single File">
<fail unless="selected-file">Must set
    property 'selected-file'</fail>
<java classname="\$\{selected-file\}">
    <classpath refid="run.classpath"/>
</java>
</target>
```

In this example, the `selected-file` property picks up the file that you have selected in the IDE. The value of `selected-file` is passed from the `project.xml` file when you choose the Run File

command.

This example also assumes that you have a working `compile-selected-files-in-src` target (like the example in Setting up the Compile File Command earlier in this chapter), although it is also possible to have the target depend on a different compile target you have set in your script.

The example uses the `refid` attribute to reference a run classpath that must be defined elsewhere in the script, with `run.classpath` specified as the `id` attribute of a `path` element. For example, `run.classpath` could be defined as in the following snippet:

```
<path id="run.classpath">
    <pathelement path="${javac.classpath}">
        <pathelement path="${build.classes.dir}">
    </path>
```

The paths for `javac.classpath` and `build.classes.dir` would need to be defined in their own path elements. If the path element needs to reference a physical location, the `location` attribute could be used to specify a directory relative to the Ant project's base directory. For example:

```
<path id="build.classes.dir">
    <pathelement location="classes">
</path>
```

Mapping the `run-selected-file` Target to the IDE Command

1. In the Files window, expand the project's folder, expand the `nbproject` folder, and then open the `project.xml` file.
2. Within the `<ide-actions>` element, add a mapping for the Run File command. The mapping might look something like the following example:

```
<action name="run.single">
    <target>run-selected-file</target>
    <context>
        <property>selected-file</property>
        <folder>${src.dir}</folder>
        <pattern>\.java$</pattern>
        <format>java-name</format>
        <arity>
            <one-file-only/>
        </arity>
    </context>
</action>
```

See [Table 16-6](#) for a description of the parts of the `<action>` element in the `project.xml` file. See Handling Properties in the `project.xml` File earlier in this chapter for information on calling properties from the `project.xml` file.

Setting up the Debug File Command

To be able to debug a selected file in a free-form project in the IDE, you need to write an Ant target for the command and then map that target in the project's `project.xml` file.

Creating the debug-selected-file Target

Following is a sample target for debugging a selected file:

```
<target name="debug-selected-file"
    depends="compile-selected-files-in-src" if="netbeans"
    description="Debug a Single File">
    <fail unless="selected-file">Must set
        property 'selected-file'</fail>
    <nbjpdastart name="${selected-file}" addressproper
        transport="dt_socket">
        <classpath refid="run.classpath"/>
        <sourcepath refid="debug.sourcepath"/>
    </nbjpdastart>
    <java fork="true" classname="${selected-file}">
        <jvmarg value="-Xdebug"/>
        <jvmarg value="-Xnoagent"/>
        <jvmarg value="-Djava.compiler=none"/>
        <jvmarg
            value="-Xrunjdwp:transport=dt_socket,address=
            <classpath refid="run.classpath"/>
        </java>
    </target>
```

In this example, the `selected-file` property picks up the file that you have selected in the IDE. The value of `selected-file` is

passed from the `project.xml` file when you choose the Debug File command.

This example also assumes you have a working `compile-selected-files-in-src` target (like the example in Setting up the Compile File Command earlier in this chapter), although it is also possible to have the target depend on a different compile target you have set in your script.

The example uses the `refid` attribute to reference two path elements (`run.classpath` and `debug.sourcepath`) that need to be defined elsewhere in your build script. For example, `run.classpath` could be defined as in the following snippet:

```
<path id="run.classpath">
    <pathelement path="${javac.classpath}">
        <pathelement path="${build.classes.dir}">
    </path>
```

The paths for `javac.classpath` and `build.classes.dir` would need to be defined in their own path elements. If the path element needs to reference a physical location, the `location` attribute could be used to specify a directory relative to the Ant project's base directory. For example:

```
<path id="build.classes.dir">
    <pathelement location="classes">
</path>
```

Refer to [Table 16-3](#) for a description of the various parts of the target.

Mapping the `debug-selected-file` Target to

the IDE Command

1. In the Files window, expand the project's folder, expand the `nbproject` folder, and then open the `project.xml` file.
2. Within the `<ide-actions>` element, add a mapping for the Debug File command.

The mapping might look something like the following example:

```
<action name="debug.single">
    <target>debug-selected-file</target>
    <context>
        <property>selected-file</property>
        <folder>${src.dir}</folder>
        <pattern>\.java$</pattern>
        <format>java-name</format>
        <arity>
            <one-file-only/>
        </arity>
    </context>
</action>
```

See [Table 16-6](#) for a description of the parts of the `<action>` element in the `project.xml` file. See Handling Properties in the `project.xml` File earlier in this chapter for information on calling properties from the `project.xml` file.

Setting up the Debugger's Apply Code Changes Command

To be able to use the debugger's Apply Code Changes feature in a free-form project, you need to write a special Ant target for the command and then map that target in the project's `project.xml` file. The Ant target needs to call the IDE's custom `nbjpdareload` task, which the IDE uses to reload the fixed code into the debugged program's JVM. See Fixing Code During a Debugging Session in [Chapter 7](#) for information on using the Apply Code Changes command.

Creating the `debug-fix` Target

Following is a sample target for running the Apply Code Changes command:

```
<target name="debug-fix" description="Reload Fixed Code  
Debugger">  
    <javac srcdir="${src.dir}" destdir="${classes.dir}"  
           classpath refid="javac.classpath"/>  
    <include name="${selected-file}.java"/>  
  </javac>  
  <nbjpdareload>  
    <fileset dir="${classes.dir}">  
      <include name="${selected-file}.class"/>  
    </fileset>  
  </nbjpdareload>  
</target>
```

In this example, the `selected-file` property picks up the file that you have selected in the IDE. The value of `selected-file` is

passed from the `project.xml` file when you choose the Run | Apply Code Changes command.

The example uses the `refid` attribute to reference the `javac.classpath` path element, which needs to be defined elsewhere in your build script. For example, `javac.classpath` could be defined as in the following snippet:

```
<path id="javac.classpath">
    <pathelement location="libs">
</path>
```

Mapping the debug-fix Target to the IDE Command

1. In the Files window, expand the project's folder, expand the `nbproject` folder, and then open the `project.xml` file.
2. Within the `<ide-actions>` element, add a mapping for the Apply Code Changes command.

The mapping might look something like the following example:

```
<action name="debug.fix">
    <target>debug-fix</target>
    <context>
        <property>selected-file</property>
        <folder>${src.dir}</folder>
        <pattern>\.java$</pattern>
        <format>relative-path-noext</format>
        <arity>
            <one-file-only/>
        </arity>
    </context>
```

</action>

Setting up the Profile Project Command for a General Java Application

If you have the NetBeans Profiler installed, you can use it to profile free-form projects. (See [Chapter 15](#) for more information on installing and using the Net-Beans Profiler.)

In order for the Profile Main Project command to work with a free-form project, you must add an Ant target to your project's `build.xml` file. This Ant target will be very similar to the target you use to run your application, with some additional tasks and parameters. The minimum changes are:

- A call to the custom `nbprofiledirect` task, with a parameter that specifies your application's CLASSPATH.
- An extra argument to the `java` task to specify a JVM that supports profiling (`${profiler.info.jvm}`).
- An extra JVM argument that signals the JVM that it is being profiled (`${profiler.info.jvmargs.agent}`).

For this example run target

```
<target name="run" description="Runs my Application">
  <depends>compile, init</depends>
  <java jar="myApplication.jar"
        fork="true">
    </java>
</target>
```

the profiling target would be

```
<target name="profile" description="Profiles my App
depends="compile, init">
    <nbprofiledirect>
        <classpath>
            <pathelement location="myApplication.jar"
            </classpath>
    </nbprofiledirect>
    <java jar="myApplication.jar"
        fork="true" jvm="${profiler.info.jvm}">
        <jvmarg value="${profiler.info.jvmargs.agr
    </java>
</target>
```

After you add the target, select Profile | Profile Main Project. The IDE will prompt you for the target that should be used for profiling.

Changing the Target JDK for a Free-Form Project

If you want to set a target JDK for your project that differs from the JDK that the IDE is running on, you must specify the JDK version in *both* of the following places:

- The Ant script for any pertinent tasks, such as `javac`. This ensures that the build targets (and the IDE commands that call them) work correctly.
- The Project Properties dialog box for the project. This ensures that IDE-specific functions, such as code completion and the Javadoc popup, work correctly.

For the `javac` task, you could do this by including the `source` and `target` options when you call `javac`. For example, the call to the task might look something like the following example. The `javac.source` and `javac.target` properties would need to be specified elsewhere in the script or in a `.properties` file with the values set to the appropriate JDK version (for example, 1.3, 1.4, or 1.5).

```
<javac srcdir="${src.dir}" destdir="${classes.dir}"
       debug="true" source="${javac.source}"
       target="${javac.target}"
       <classpath refid="javac.classpath"/>
</javac>
```

To change the target JDK in the project's properties:

1. In the Projects window, right-click the project's node and

choose Properties.

2. In the Project Properties dialog box, select the Java Sources node and select the target JDK from the Source Level combo box.



If you want code completion and other Source Editor features to work exactly according to the target JDK, make sure that you have the target JDK on your system and registered in the IDE. Choose Tools | Java Platform Manager. If the JDK version that you want to use is not listed in the Platforms list, click Add Platform and navigate to the folder for the JDK that you want to use.

Making a Custom Menu Item for a Target

If your build script has a target that does not exactly correspond to any of the available menu items, you can create a custom menu item for it. The menu item is then available when you right-click the project's node in the Projects window.

To create a custom menu item for a target:

1. In the Projects window, right-click the project's node and choose Properties.
2. In the Project Properties dialog box, select the Build and Run node.
3. Click the Add button next to the Custom Menu Items table to add a blank row to the table.
4. Fill in the target name in the Ant Target column and the name for the menu item in the Label column.

Debugging Ant Scripts

If you need to troubleshoot an Ant script, or you just would like a tool to help you make sense of a working script, you can use the IDE's Ant Debugger module. The module essentially plugs into the IDE's visual debugging framework, provides most of the debugging features you are used to for Java files, and applies those features to Ant scripts. For example, you can

- Use the Step Into, Step Over, Step Out, and Continue commands to trace execution of the script or a specific target. This feature is particularly useful to help you untangle the order in which nested targets (and even nested scripts) are called.
- Use the Call Stack window to monitor the current hierarchy of nested calls.
- Set breakpoints.
- View the values of properties in the Local Variables window.
- Set a watch on a property.

Figure 16-1. Ant Debugger

The screenshot shows the NetBeans IDE interface. The top half displays a portion of a build script (build.xml) with several targets defined. The bottom half shows the 'Local Variables' window, which lists various environment variables and their values.

```
<target name="-do-compile" depends="init,deps-jar,-pre-pre-c
    <j2seproject2:javac/>
    <copy todir="${build.classes.dir}">
        <fileset dir="${src.dir}" excludes="${build.classes.}</copy>
</target>
<target name="-post-compile">
    <!-- Empty placeholder for easier customization. -->
    <!-- You can override this target in the ../build.xml fi
</target>
<target name="compile" depends="init,deps-jar,-pre-pre-compi
<target name="-pre-compile-single">
    <!-- Empty placeholder for easier customization. -->
    <!-- You can override this target in the ../build.xml fi
</target>
<target name="-do-compile-single" depends="init,deps-jar,-pr
    <fail unless="javac.includes">Must select some files in
```

Root	Type	Value
default.javac.source		1.5
file.encoding.pkg		sun.io
javac.source		1.5
Env-TMP		C:\DOCUME~1\PATRIC...
netbeans.system_http		http://webcache.uk.sun.c...
java.home		E:\Program Files\Java\jdk...
test.src.dir		test
javadoc.version		false
Env-USERNAME		Patrick Keegan
env-userdomain		PKEEGAN-LAPTOP
java.endorsed.dirs		E:\Program Files\Java\jdk...

To start debugging a build script:

1. Open the Files window and navigate to the build script.
2. Right-click the build script, and choose Debug Target and then the name of the target you want to debug.

The program counter goes to the line where the target is declared and stops. Then you can step through the target with

any of the normal debugging commands.

Chapter 17. Developing NetBeans Plug-in Modules

- [Plug-in Modules](#)
- [Rich-Client Applications](#)
- [Extending NetBeans IDE with Plug-in Modules](#)
- [Setting up a Plug-in Module](#)
- [Using the NetBeans APIs](#)
- [Registering the Plug-in Module](#)
- [Adding a License to a Plug-in Module](#)
- [Building and Trying Out a Plug-in Module](#)
- [Packaging and Distributing a Plug-in Module](#)
- [Packaging and Distributing a Rich-Client Application](#)
- [Finding Additional Information](#)

BEGINNING WITH THE 5.0 RELEASE, comprehensive support for plug-in module development was introduced for the first time in NetBeans IDE. Even though it had been possible to create plug-in modules for NetBeans IDE with previous releases, and even though many developers had been doing so, NetBeans IDE did not provide user interface or project system support for developing NetBeans plug-in modules. However, in NetBeans

IDE 5.0, a plug-in module development environment was created, aiming to get you started as quickly as possible and guiding you from start to finish. Wizards were provided for setting up the basic framework of a plug-in module project. Further wizards were provided for adding the specific items that you might want to add to the IDE. Finally, a broad range of other user interface features and usability enhancements were added, aiming to simplify and streamline plug-in module development.

Plug-in Modules

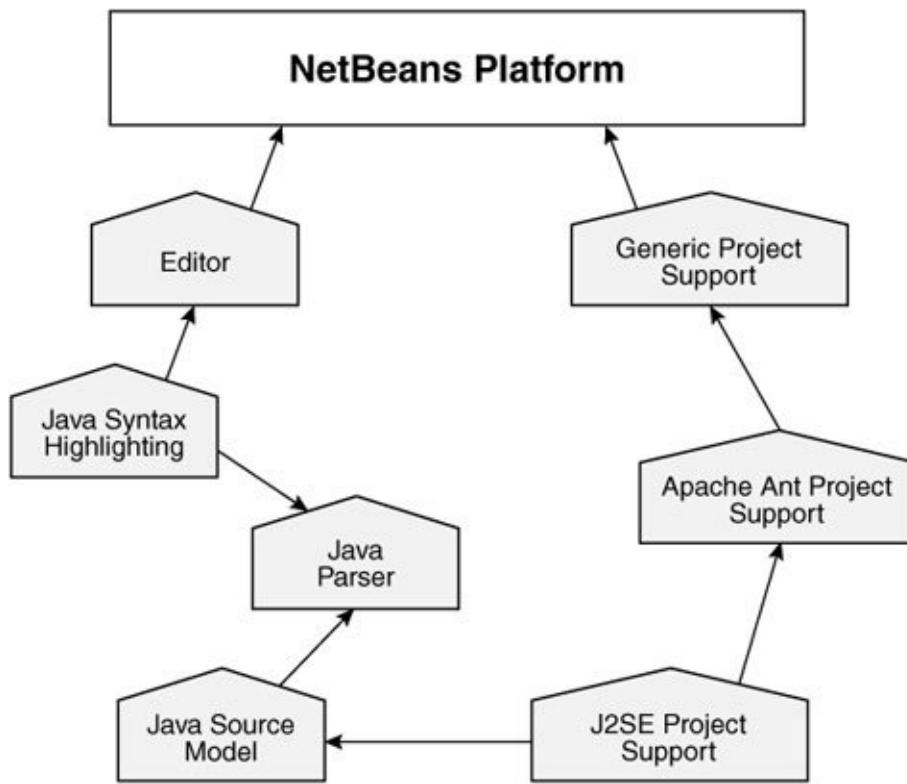
By creating NetBeans plug-in modules, you can extend NetBeans IDE with your favorite tools and technologies. Whether you want to make functionality from a third-party library available to NetBeans IDE, or whether you want to add user interface support for the latest cutting-edge technology, the NetBeans plug-in module development environment in NetBeans IDE 5.0 is tailored to your specific needs. For example, you can use wizards to very quickly add skeletons of menu items, toolbar buttons, window components, editors, and more, to NetBeans IDE, and thereby provide a solid basis for extending the IDE's usefulness to you, your colleagues, and customers.

Rich-Client Applications

Apart from plug-in modules, you can create complete, functioning, standalone rich-client applications. The NetBeans open source project provides an application framework called the NetBeans Platform. The NetBeans Platform has all the basics that every desktop application needs, such as menus, toolbars, and a windowing system. In fact, NetBeans IDE itself is built on top of the NetBeans Platform and serves as an example NetBeans rich-client application.

NetBeans rich-client applications are built out of a series of NetBeans plug-in modules added to the NetBeans Platform. For example, the NetBeans IDE is itself made up of a large collection of plug-in modules that have been added to the NetBeans Platform, as shown in [Figure 17-1](#).

Figure 17-1. NetBeans IDE: a rich-client application, assembled from plug-in modules



Read this chapter if you want to develop either NetBeans plug-in modules or rich-client applications, because the general process of creating a rich-client application in NetBeans IDE 5.0 is similar to that of creating NetBeans plug-in modules. Where there are differences between developing NetBeans rich-client applications and NetBeans plug-in modules in the context of the topics discussed in this chapter, they are mentioned.

A comprehensive guide to developing NetBeans plug-in modules is not provided in this chapter; that subject deserves its own separate book. Instead, this chapter focuses on the development process and the user interface that was created to assist you in developing plug-in modules. After reading this chapter, you will have the basics to get started. You will also be given some pointers for further exploration and helpful tips for developing plug-in modules.

Extending NetBeans IDE with Plug-in Modules

Once you have decided on the functionality you want to add to NetBeans IDE or assemble as a NetBeans rich-client application, it is useful to do some planning to decide how to best design and implement your plug-in module, along with how to structure your plug-in modules using the plug-in module development support available in NetBeans IDE 5.0.

Many of the best design practices associated with software development are applicable when developing NetBeans plug-in modules, such as the use of software patterns, package naming, etc. In addition, there are other considerations to take into account when creating NetBeans plug-in modules. The following are some of the things you may want to think about when deciding how to structure your plug-in modules.

- What kind of functionality do you want to add to NetBeans IDE?

Your plug-in module can be as extensive as your imagination and skills allow. For example, you can create plug-in modules for new functionality by implementing menu items, toolbar buttons, custom components, and editor functionality such as syntax highlighting, code completion, code folding, and multi-view editors.

- Are there any third-party libraries that you plan to use with the plug-in module?

If you plan to use third-party libraries with your plug-in module, NetBeans IDE provides a convenient way to incorporate third-party library access and distribution, via a plug-in module project template called "library wrapper module project." In addition, a library wrapper module

project also provides the capability to include a third-party license, should it be required for its redistribution with your plug-in module.

- How should you manage dependencies between multiple plug-in modules?

NetBeans IDE requires that you do not use circular dependencies in your plug-in modules. A circular dependency is a situation in which a Java class in plug-in module A is dependent on a Java class in plug-in module B, and plug-in module B has a Java class that is dependent on a Java class in plug-in module A. NetBeans IDE will automatically detect circular dependencies, should you attempt to use one or you inadvertently add one.

Performing some analysis and design on the structure of your plug-in module(s) will greatly help in avoiding circular dependencies and result in a much easier to maintain plug-in module.

- Are the behavior and user interface consistent with NetBeans design guidelines?

If you want the plug-in module to have the NetBeans "look and feel," this is an important aspect to consider.

- How do you want to distribute your plug-in module?

If you are building a plug-in module to extend the functionality of NetBeans IDE, you can easily distribute it as a NetBeans Module binary file, which has an `.nbm` extension and is very similar to a JAR or ZIP file. You can create an NBM file from your plug-in module project with one mouse click. The resulting NBM file contains all the necessary information to integrate and run your plug-in module in an installation of NetBeans IDE.

Any user interested in using your plug-in module can easily add it to his or her installation of NetBeans IDE by using the Tools | Update Center menu item. In the Update Center wizard, your plug-in module is added by selecting the NBM file from a local file system or from an Update Center you have populated with your NBM file. At some point you will want to choose between whether to distribute your NBM file via an Update Center or by selecting an NBM file from a local file system. This is not necessarily a decision that needs to be made prior to beginning development of your plug-in module. However, it is something you should be considering.

Setting up a Plug-in Module

NetBeans IDE provides three project templates that you use when you begin creating plug-in modules. They are as follows:

- **Module.** Use this to provide the functionality and actual business logic of your plug-in module. For example, the menu items, dialog boxes, window components, and so on, as well as all the code for interaction between these items, are defined in one or more module projects.
- **Library Wrapper Module.** Use this to put a third-party library on a plug-in module's classpath. For example, a plug-in module that makes use of the third-party library JDOM would only be able to use JDOM's classes after the library provided by JDOM is wrapped in a library wrapper module. By itself, the library wrapper module project does nothing. The library wrapper module project must be deployed together with its related module projects via a module suite project.
- **Module Suite.** Use this to group a set of interdependent modules and library wrappers. This allows them to be deployed together. For rich-client applications, module suites provide additional featuresa splash screen, a launcher, and a title for the titlebar. In addition, module suite projects include functionality that enables rich-client applications to be bundled into a ZIP file or provided as a JNLP (web-startable) application.



If you intend to build a simple plug-in module without dependencies, you will not need a module suite.

Creating a Module Project

To create a plug-in module, start by using a module project template. To use a module project template:

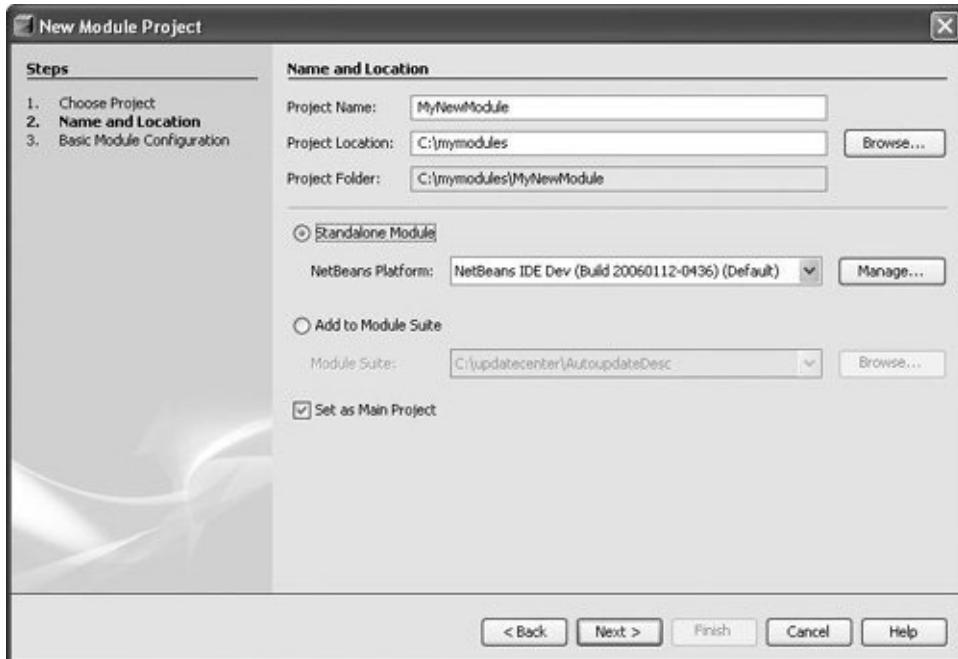
1. Choose File | New Project (Ctrl-Shift-N). In the Categories tree, choose NetBeans Plug-in Modules. In the Projects tree, choose Module Project. Click Next.

The New Module Project wizard appears.

2. Type a project name and specify where it should be stored in your file system.
3. Select whether the module project will be standalone or part of a module suite project. If you specify that the module project will be standalone (as shown in [Figure 17-2](#)), you must also specify the platform it will be built against.

Figure 17-2. New Module Project wizard

[[View full size image](#)]



By default, a module project is built against the NetBeans IDE installation you are running. If you want to build against a different version or installation (such as a stripped-down copy of the NetBeans Platform with no IDE modules at all), you can configure the IDE to do so. Depending on the platform you choose, you may or may not have the modules that you need available to you. For example, a stripped-down version of a platform might not have the modules that relate to editor functionality. Therefore, make sure that the platform you choose includes the modules you need.

If a suitable platform is not available, click **Manage** and then use the NetBeans Platform Manager to choose a different platform.

If you specify that the module project will be standalone, you can change your mind later and add it to a module suite project. You can only add your module project to a module suite project if one already exists.

4. Specify whether the module project will be the IDE's main

project. The main benefit of this is that there is only one main project in the IDE, and certain keyboard shortcuts are specifically geared toward the main project. For example, F6 is Run Main Project.

5. Click Next. Here you specify the code name base. The code name base uniquely identifies a plug-in module. By convention, it is typically the name of the base package, such as `org.netbeans.modules.mymodule`.

In Module Display Name, you can specify a more user-friendly name that is displayed in the IDE.

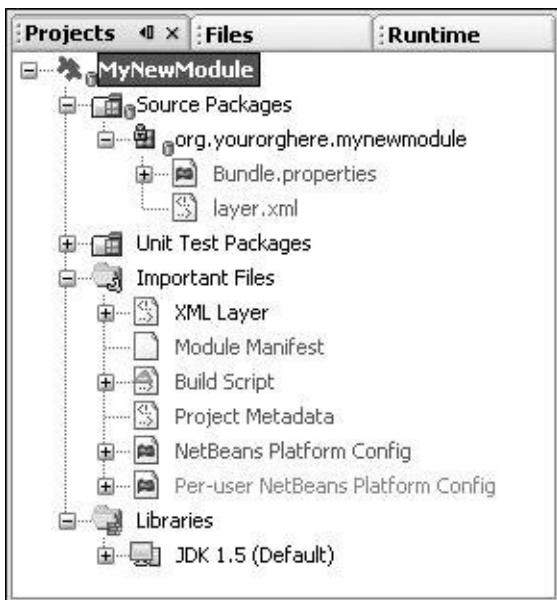
6. Finally, you specify the location of the localizing bundle and the XML layer file.

The former will contain name-value keys for localization, while the latter will register each item that the plug-in module will add to the IDE. The XML layer file is discussed in Registering the Plug-in Module later in this chapter. Normally, you would not change the default locations.

7. Click Finish.

The project opens in the IDE. Nodes representing the module project are shown in the Projects window. The Projects window hides the complexity of all the files in the module project and structures them in a helpful way, as shown in [Figure 17-3](#).

Figure 17-3. A module project in the Projects window



When you switch to the Files window (Ctrl-2), you can see all the files that make up the module project.

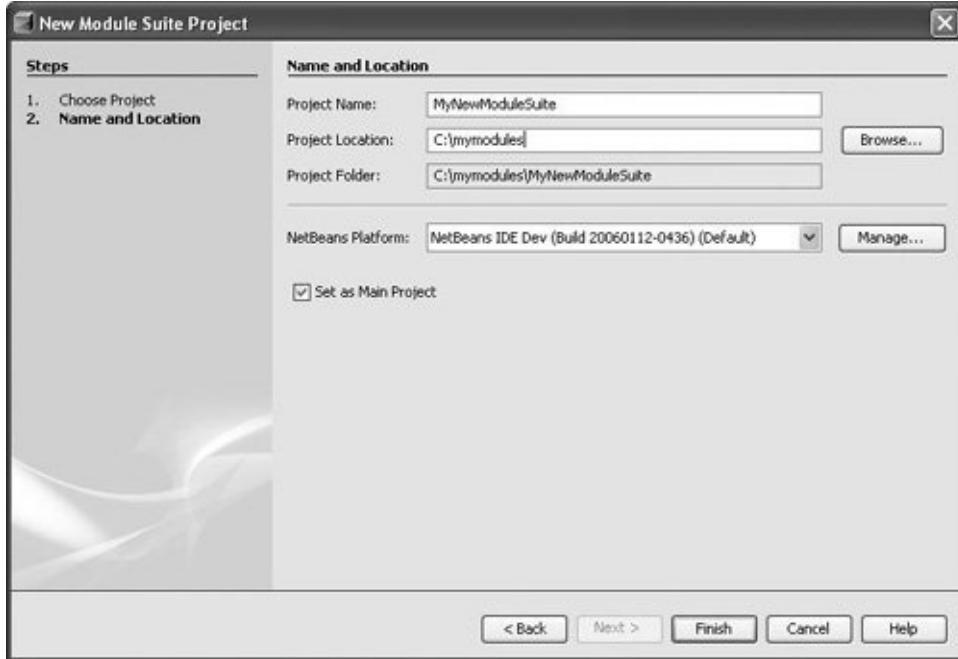
Creating a Module Suite Project

When developing a plug-in module, you often need to make use of a third-party library. Since the third-party library will be deployed together with the plug-in module, you will need to group them together via a module suite project. Therefore, you use the module suite project template before using the library wrapper module project template. To use a module suite project template:

1. Choose File | New Project (Ctrl-Shift-N). In the Categories tree, choose NetBeans Plug-in Modules. In the Projects tree, choose Module Suite Project. Click Next.
2. Type a project name and specify where it should be stored in your file system, as shown in [Figure 17-4](#).

Figure 17-4. New Module Suite Project wizard

[[View full size image](#)]



3. Specify the platform against which your module suite project will be built.

By default, a module suite project is built against the NetBeans IDE installation in which you are creating it. Depending on the platform you choose, you may or may not have the modules that you need available to you. For example, a stripped-down version of a platform might not have the modules that relate to editor functionality. Therefore, make sure that the platform you choose includes the modules you need. If a suitable platform is not available, click Manage and use the NetBeans Platform Manager to choose a different platform.

4. Specify whether the module suite project will be the IDE's main project. Click Finish.

The project opens in the IDE. When you expand the project node, a Modules node appears. Right-click this node if you

need to add module projects that are not currently assigned to any other module suite.

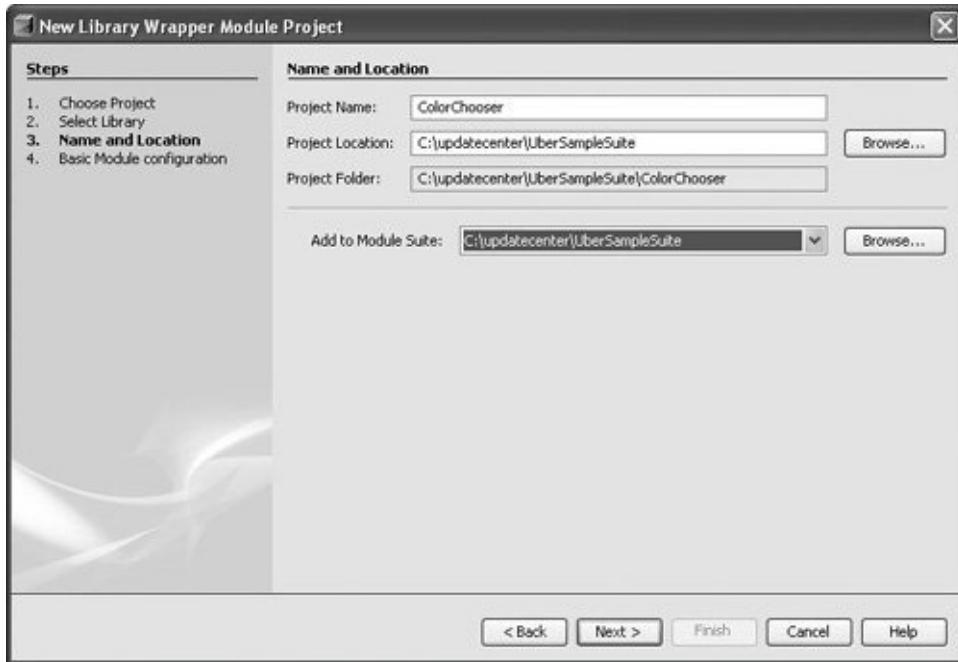
Creating a Library Wrapper Module Project

NetBeans IDE 5.0 lets you wrap one or more third-party libraries with the module project. To use the library wrapper module project template:

1. Choose File | New Project (Ctrl-Shift-N). In the Categories tree, choose NetBeans Plug-in Modules. In the Projects tree, choose Library Wrapper Module Project. Click Next.
2. Browse to the third-party library that you need to make available to your module project.
3. Optionally, browse to the third-party library's license. While doing this is optional in the IDE, adding a third-party library's license will probably not be optional if you want to make the plug-in module publicly available. Click Next.
4. Specify a project name and a location on your file system where you want to store it.
5. Next, specify the module suite to which the library wrapper module will belong, as shown in [Figure 17-5](#). The library wrapper module project will be deployed along with the module suite. Click Next.

Figure 17-5. New Library Wrapper Module Project wizard

[\[View full size image\]](#)



6. Specify the code name base. The code name base uniquely identifies a plugin module. By convention, it is typically the name of the base package, such as `org.netbeans.modules.mymodule`.

In Module Display Name, you can specify a more user-friendly name that is displayed in the IDE.

Finally, you specify the location of the localizing bundle. This will contain name-value keys for localization.

Using the NetBeans APIs

Once you have set up a plug-in module, you need to start coding. The NetBeans APIs provide interfaces to the standard plug-in modules that make up the NetBeans system. The NetBeans APIs enable you to integrate your own functionality with the IDE. For example, if you want the IDE to access user files on disk, you use the NetBeans Filesystems API. For the NetBeans API List, see

<http://www.netbeans.org/download/dev/javadoc>.

Each NetBeans API may have mandatory methods as well as optional methods. For example, the NetBeans Actions API, which you use when you define menu items, requires that you specify the label that appears in the menu item. It leaves it up to you to decide whether the menu item should include an icon.

There are many NetBeans APIs. However, the most commonly used are:

- **NetBeans Actions API.** Defines global singleton actions (such as Open, Cut, Paste) that your module may use.
- **NetBeans Filesystems API.** An API for "virtual files." NetBeans uses this API to access user files on disk, files inside JARs, and configuration data for the IDE.
- **NetBeans Loaders API.** Provides DataObjects that wrap FileObjects and provide programmatic access to their contents. Each file type NetBeans recognizes (such as Java files or HTML files) has a corresponding DataObject subclass provided by the module that adds support for that file type.
- **NetBeans Nodes API.** Generic hierarchy and action

context. Nodes are similar to TreeNodes (from Swing), but can be used in more than trees. Nodes add user-visible components such as pop-up menu actions, localized display names, and icons to DataObjects, but can also be used without DataObjects, with other data models.

- **NetBeans Windows API.** Allows modules to provide window-like components, mainly through embeddable visual components known as "top components."
- **NetBeans JavaHelp Integration API.** Adds HTML help files to the IDE or an application built on the NetBeans Platform.

The IDE provides several ways of simplifying the process of working with the NetBeans APIs. Most importantly, the IDE assists you by providing several wizards that guide you through the initial phase of working with an API. For example, the New Action wizard provides the basis of an implementation of the NetBeans Actions API, the New File Type wizard provides the basis of an implementation of the NetBeans Loaders API, etc.

In the following sections you will look at several of these wizards and how they relate to their APIs. In the process, you will learn how to add menu items, toolbar buttons, loaders, and custom components to the IDE.

Adding a Menu Item and a Toolbar Button to the IDE

Menu items, toolbar buttons, and keyboard shortcuts are implemented via the NetBeans Actions API. They are "presenters" that may display the same action. What distinguishes the presenters from each other is how they are displayed and how they are invoked a menu item is displayed in

the menu bar, in a file or folder's pop-up menu, or in an editor's pop-up menu. A toolbar button is normally displayed in the main toolbar, while a keyboard shortcut isn't displayed anywhere but its underlying action is invoked by pressing one or more keys.

NetBeans IDE includes a template that kickstarts your work with the NetBeans Actions APIthe New Action wizard. Once you've worked through the New Action wizard, you have a Java source file that provides the skeleton of an implementation of the NetBeans Actions API.

To use the New Action wizard:

1. Right-click the module project's node in the Projects window and choose New | Action. If Action is not included in the list, choose File/Folder and then select it from the NetBeans Module Development category and click Next.
2. In the Action Type panel, specify the conditions under which the user will be able to invoke the action.

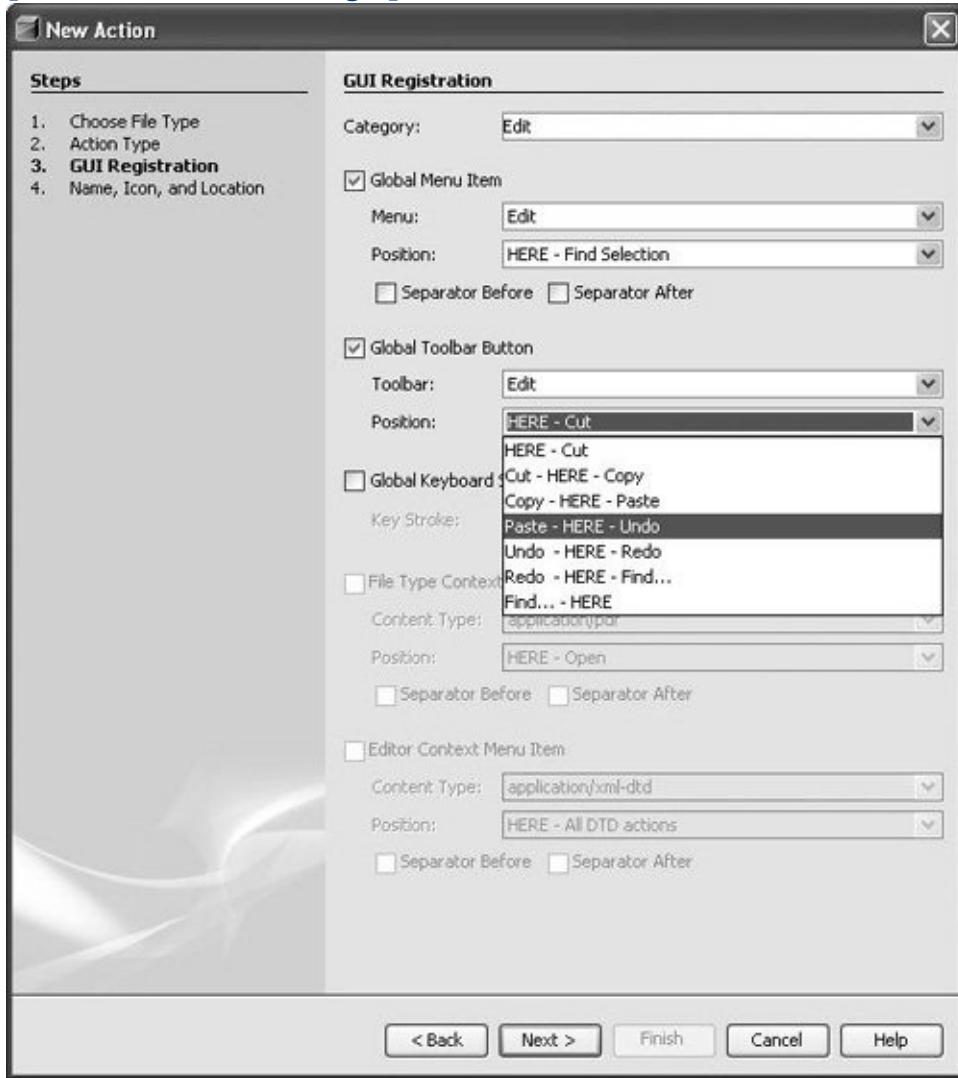
If the action is set to be Always Enabled, you can implement it as a menu item, toolbar button, or keyboard shortcut.

If the action is set to Conditionally Enabled, it is context sensitive. This means that you will be able to implement it so that it can be invoked from a node such as a JSP file's pop-up menu or the Java editor's pop-up menu. In addition, if the action is conditional, you can specify whether it will be activated when more than just the node to which it is assigned is selected.

3. In the GUI Registration panel, specify where the plug-in module will position the action in the IDE, as shown in [Figure 17-6](#).

Figure 17-6. New Action wizard, GUI Registration page

[View full size image]



For example, if you select the Global Menu Item checkbox, the wizard will add appropriate entries to the main menu bar.

4. Select Global Menu Item, if you want the action to be displayed as a menu item in the main menu bar. You might have to wait a few seconds while the IDE fills the Menu and Position drop-down lists with all the menus and related

menu items provided by all the modules that are currently registered in the platform.

5. In the Position drop-down list, select the appropriate place where the menu item will be positioned within the selected menu.
6. Select Separator Before and Separator After if you want the wizard to add separators before and after your menu item.
7. Select Global Toolbar Button, if you want the action to be displayed as a toolbar button in the main toolbar.
8. In the Position drop-down list, select the appropriate place where the toolbar button item will be positioned within the selected toolbar.
9. If the action is conditionally enabled, you can specify that it should be displayed in a file type's pop-up menu or in an editor's pop-up menu, or both. Again, you can specify whether there should be separators before and after the item in the pop-up menu.
10. Type the class name and the name that will be displayed in the menu item. If you specified that you want to create a toolbar button, you must specify an icon. Not all menu items need an icon.
11. Finally, specify a package where the action class will be created.

When you click Finish, the action class opens in the IDE. The basic methods that the NetBeans Actions API requires are included in the class. For example, the `performAction()` method is the hook for the code that is invoked when the action is called from the menu item or toolbar button.

Even though you yourself have added no code whatsoever, the plug-in module has all the content it needs to be built, installed,

and distributed.

Creating an Editor for a New File Type

The centerpiece of NetBeans IDE is its Source Editor. In it, you can edit a wide variety of file types. Supporting you in the Source Editor are various features, such as syntax highlighting, code completion, code folding, and drag-and-drop functionality. Each of these features is tailored toward a specific file type. Therefore, when you want to create an editor for a new file type, the first thing you must do is ensure that NetBeans is able to distinguish your file type from all other file types. Only once it is able to do that does it make sense to provide features such as syntax highlighting for the file type.

File types that are recognized in the IDE have their own icons, menu items, and behavior. The files that are shown in the IDE are FileObjects, which are generally wrappers around `java.io.File`. What the user sees are Nodes, which provide functionality such as actions, as defined in the previous section, and localized names to objects such as files.

In between Nodes and FileObjects are DataObjects. A DataObject is like a FileObject, except that it knows what kind of file is being shown. There are usually different types of DataObjects for files with different extensions and for XML files with different namespaces. Each DataObject is provided by a different module, each implementing support for one or more file types. For example, the Image module makes it possible to recognize and open `.gif` and `.png` files.

A plug-in module that recognizes a file type installs a DataLoader, which is a factory for a file-type-specific DataObject. When a folder is expanded, the IDE asks each known DataLoader, "Do you know what this is?" The first DataLoader that says "Yes" creates the DataObject for the file.

In order to actually display something for each file, the underlying NetBeans IDE subsystem calls `DataObject.getNodeDelegate()` for each DataObject and the Nodes are what you actually see in the IDE.

A template is included in the NetBeans IDE for kickstarting your work with the NetBeans Loaders APIthe New File Type wizard. Once you've worked through the New File Type wizard, you have a basic loaderall the classes that are needed for the IDE to recognize the file type as distinct from all other file types. Once you have a newly recognized file type, you can add a lot of functionality to it, such as syntax coloring, a multi-view editor, code completion, and so on.

To use the New File Type wizard:

1. Right-click the module project's node in the Projects window and choose New | File Type. If File Type is not included in the list, choose File/Folder and then select it from the NetBeans Module Development category and click Next.

The File Recognition panel appears.

2. In the File Recognition panel, specify how the IDE should recognize your new file type.

A file is recognized by a combination of its MIME type and either its file extension or namespace. The latter is for XML documents only.

For example, you could type `text/x-java-jar-manifest` in MIME Type and then type `.mf .MF` in Extension(s). This would register a new MIME type and assign it to all files that have the extension `mf` or `MF`.

3. Click Next. Type the class name prefix. This determines the prefix of all the Java source files that the wizard will create to handle the new file type.

4. Optionally, you can specify an icon.

This is useful because it allows you to see at one glance whether your plug-in module is working or not. If the file type is displayed with your icon, the plug-in module is working, if it's not there's something wrong.

When you click Finish, the following files are created by the wizard:

- The node that provides the user interface in the IDE
- The loader that recognizes the MIME type and functions as a factory for the data object
- The data object that wraps the file object
- The BeanInfo object that handles the display in the Options window (Advanced Options | IDE Configuration | System | File Types)
- The resolver XML file that maps the MIME type to the file extension
- A sample file of the type that you defined

Even though you have added no code whatsoever, the plug-in module has all the content it needs for the IDE to distinguish files with the `mf` or `MF` extension from all other file types. After you try out the plug-in module, you can extend it further by adding code completion, syntax coloring, and other typical editor features for the file type.

Adding a Component to the Main Window

The user interface of a plug-in module often makes use of one or more panels. Whether you are creating a wizard, a GUI editor, or a toolbar, your plug-in module will frequently need to make use of a windowing component. In the NetBeans APIs, a visual component called "top component" is provided, which you can embed in one of many places in the IDE. A top component can correspond to a single window or a tab in a window. It can provide a variety of functionality, including its own nodes and actions.

NetBeans IDE includes a template that kickstarts your work with the NetBeans Window System APIthe New Window Component wizard. Once you've worked through the New Window Component wizard, you have a Java source file that provides the skeleton of an implementation of the NetBeans Window System API.

To use the New Window Component wizard:

1. Right-click the module project's node in the Projects window and choose New | Window Component.

If Window Component is not included in the list, choose File/Folder and then select it from the NetBeans Module Development category and click Next.

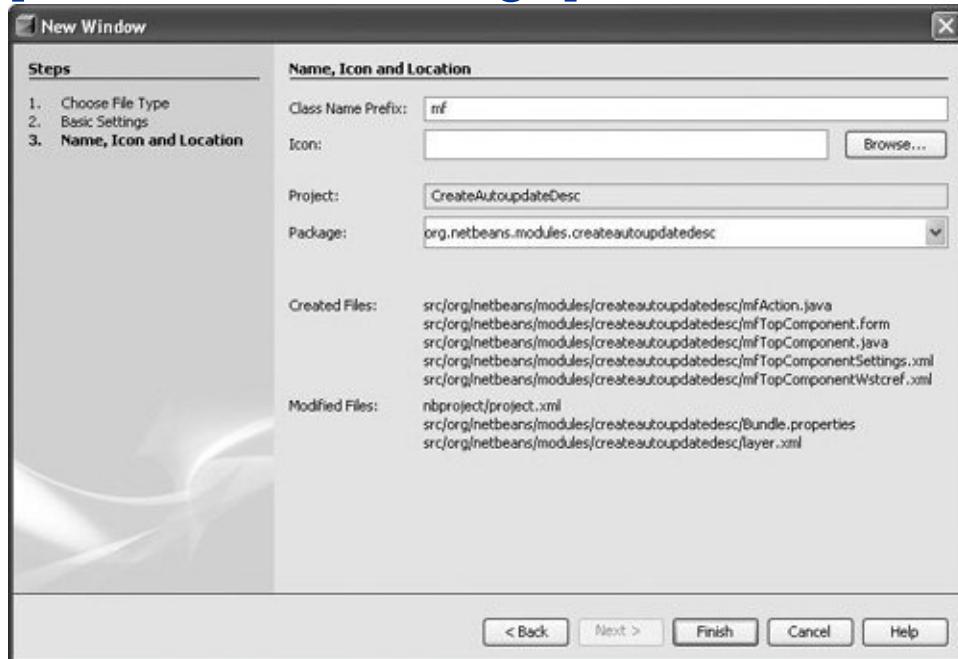
2. In the Basic Settings panel, specify the place in the IDE where you want the window component to be positioned. For example, if you want to display the window component on the left side of the IDE, where the Projects and Files windows are also displayed, choose "explorer" in the drop-down list. On the other hand, you might want the window component to be displayed in the same location as the Source Editor. You can pick from a wide variety of positions in the IDE, depending on the modules that are available to the platform against which your development IDE builds.
3. Next, specify whether the new window component will be

opened when the IDE starts. If you leave this checkbox unselected, the window component will be opened only when the user invokes the action that opens the window component. The wizard will create this action for you. Click Next.

4. In the Name, Icon and Location panel, type the class name prefix, as shown in [Figure 17-7](#).

Figure 17-7. New Window Component wizard

[[View full size image](#)]



This determines the prefix of all the Java source files that the wizard will create. Optionally, you can specify an icon. This icon will be displayed in the tab of the window component.

When you click Finish, several files are created. Some of them `TopComponentWstcref.xml` and `TopComponentSettings.xml` are used for

serializing the window component. When you open the Java source file that subclasses `TopComponent` in the Form Editor, you can use the Free Design layout (as explained in [Chapter 6](#)) for the design and layout of the window component.

However, without adding any additional code, you can build your plug-in module. When you install it, a new menu item is added to the Window menu. When you select it, the empty window component will be displayed in the location that you specified in the wizard.

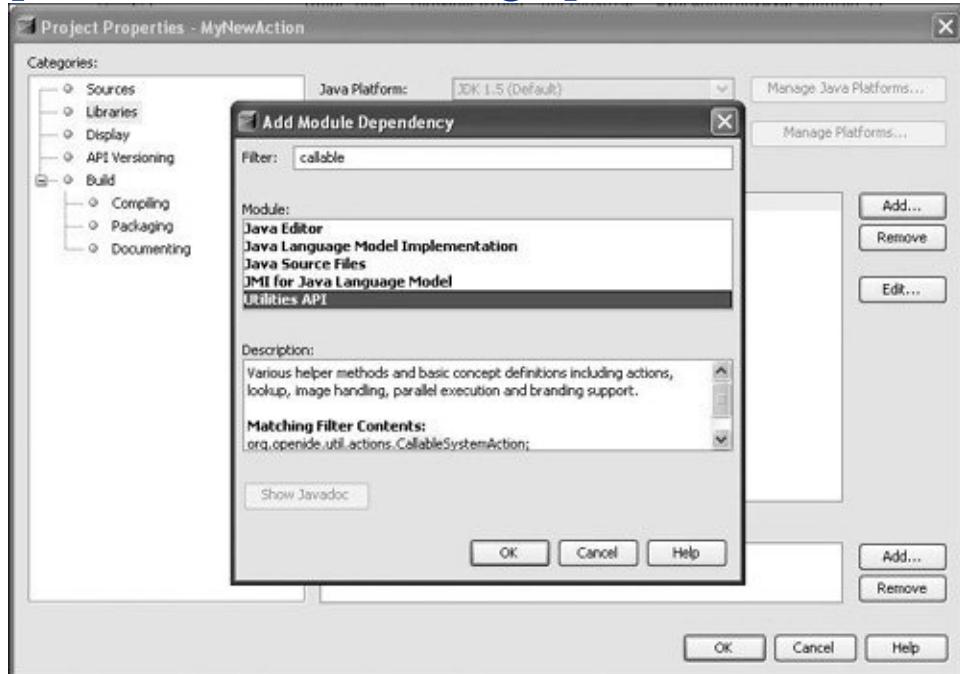
Finding the NetBeans APIs That You Need

After you have used one or more wizards to create items such as actions and windowing components, you begin coding. One way in which the IDE helps you at this stage is by letting you find the NetBeans APIs that you need very quickly and easily. If you know the name of the class you need to work with, but not the NetBeans API that it belongs to, the IDE provides a search facility for you. To search for a NetBeans API, do the following:

1. Right-click a project node, choose Properties, and click Libraries in the Project Properties dialog box.
2. Click Add at the top of the panel. The Add Module Dependency dialog box appears.
3. Begin typing the class that you need to use, and notice that the module list narrows, showing only the modules that contain the content of the filter, as shown in [Figure 17-8](#).

Figure 17-8. Add Module Dependency dialog box

[\[View full size image\]](#)



The module list continues to narrow until only the modules that can provide the class remain. You can then select the module. It is added to the list in the Libraries panel. When you click OK, the module dependency is declared in the plug-in module's metadata, in the project.xml file.

Registering the Plug-in Module

When you use the New Action wizard to add a menu item to the IDE, the wizard registers the menu item in the plug-in module's `layer.xml` file (bold font added for readability):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC "-//NetBeans//DTD Filesys
/www.netbeans.org/dtds/filesystem-1_1.dtd">
<filesystem>
    <folder name="Actions">
        <folder name="Edit">
            <file name="modules-mynewmodule-myNewAction">
                <attr name="instanceClass"
                    stringvalue="mynewmodule.myNewAction"
                </file>
            </folder>
        </folder>
        <folder name="Menu">
            <folder name="Edit">
                <file name="modules-mynewmodule-myNewAction">
                    </file>
            </folder>
        </folder>
    </filesystem>
```

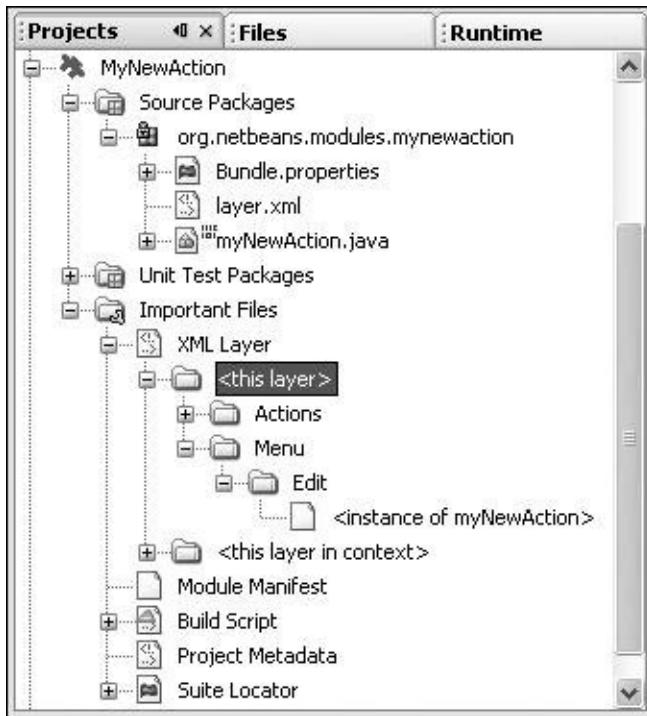
Similarly, when you use the New File Type wizard to create a loader for a new file type, the wizard creates `layer.xml` entries that register a set of default menu items that will appear in the file's pop-up menu.

So, what is the `layer.xml` file? The `layer.xml` file is the plug-in module's configuration file. Items that you want displayed in the IDE need to be registered in `layer.xml`. If you use a wizard, the wizard registers the items for you. However, you can tweak the

registered items later in the `layer.xml` file. For example, you may want to rearrange a menu item so that it will be displayed lower down in the list of items belonging to a menu.

A special tree view containing a node for each element is provided to simplify modifications to the `layer.xml` file. In [Figure 17-9](#), you can see the tree view of the `layer.xml` file shown above.

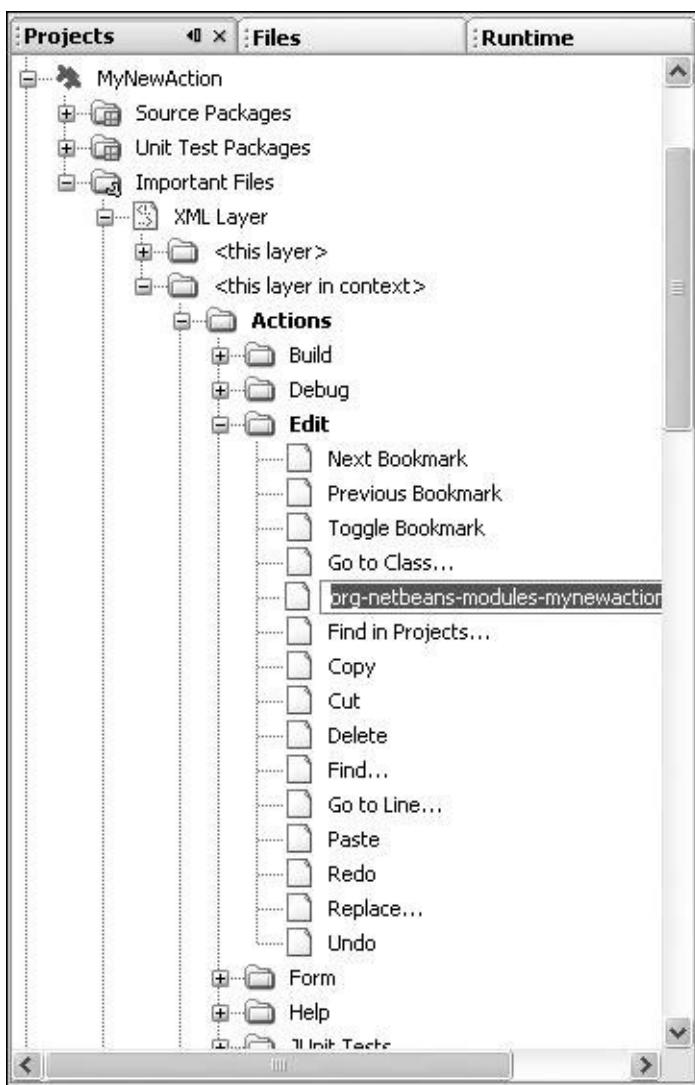
Figure 17-9. Browsing the current module project's `layer.xml` file



You can right-click the node for the new menu item standard items such as Cut, Copy, and Paste are available. In addition, you can localize the menu item or pick an icon that will be displayed in the menu item.

The next tree hierarchy, named "<this layer in context>," shows you all the items registered by all the modules that are loaded in the IDE. The items in bold are those that your plug-in module will contribute to the IDE, assuming you install the plug-in module, as shown in [Figure 17-10](#).

Figure 17-10. Browsing all the `layer.xml` files registered in the current platform



Adding a License to a Plug-in Module

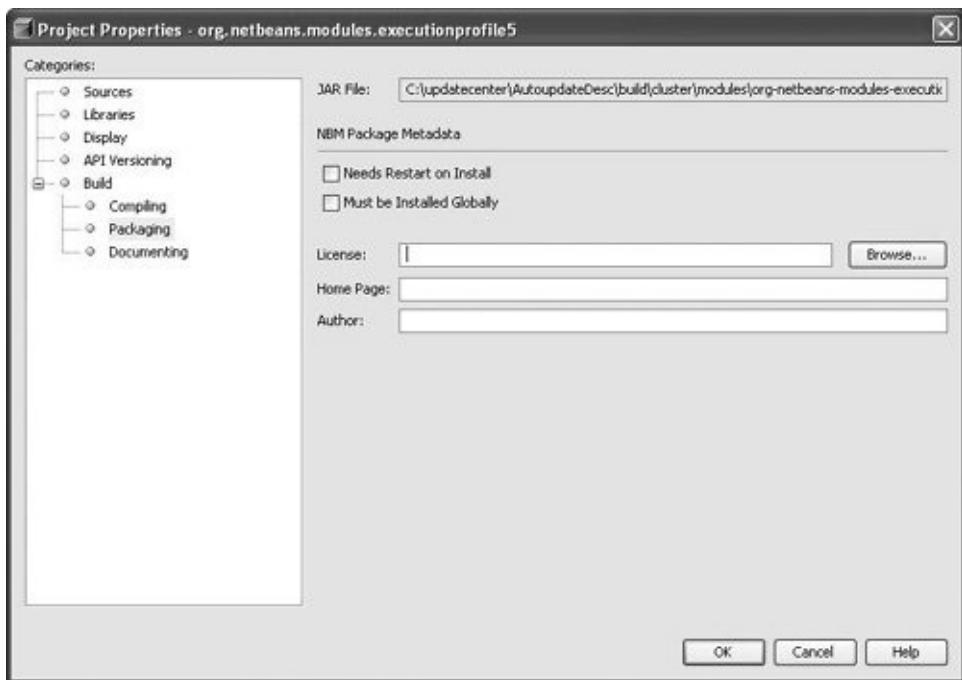
With the creation of any software, there is often a need for that software to have a software license. NetBeans IDE support for creating plug-in modules allows you to add a license to the plug-in module that is being created. The Packaging panel of the module's Project Properties dialog box makes provision for this type of packaging information.

To add a license to a plug-in module:

- 1.** Right-click the module project node and choose Properties.
- 2.** In the Project Properties dialog box, click Packaging.
- 3.** Specify the packaging information, such as the license and author details, as shown in [Figure 17-11](#).

Figure 17-11. Packaging a plug-in module

[\[View full size image\]](#)



Information specified in this panel is packaged with the NBM file, as described in the next section. And, when you make the plug-in module available via an Update Center, the information in this panel needs to be included in the autoupdate descriptor, which is described later in this chapter.

Building and Trying Out a Plug-in Module

After you have coded your plug-in module, you can build it and try it out within the IDE instance where you are working. The IDE uses an Ant script (`build.xml` file) to build plug-in modules. The Ant script is created when you create the module project. You can change the Ant script manually. It is located within the Important Files node. You can also use the Project Properties dialog box to modify settings in the Ant script.

You set and modify the plug-in module's dependencies, versioning, and packaging information in the Project Properties dialog box. Right-click the module project's node in the Projects window and choose Properties to access the Project Properties dialog box.

The build script that you use to build the plug-in module is called via a menu item in the IDE.

To build a plug-in module:

- 1.** In the Projects window, right-click the node of the module project you want to build.
- 2.** Choose Build Project.

After you build a plug-in module, you can try it out from inside the IDE. When you run the plug-in module from the IDE, you can specify that the run process should start another copy of the IDE with the plug-in module loaded in it.

To try out a plug-in module:

- 1.** In the Projects window, right-click the node of the module project you want to build.
- 2.** Choose Install/Reload in Target Platform to load the plug-in

module in a freshly started copy of the IDE.

After developing the plug-in module further, you can repeat the process above, because when you try out the plug-in module, the IDE unloads the previous version of the plug-in module and replaces it with the version you are currently trying out.

You can debug a plug-in module just as you would any other application in NetBeans IDE, by simply selecting Debug Main Project from the Run menu or by right-clicking on a module project in the Projects window and choosing Debug Project. The same is true for profiling.

Packaging and Distributing a Plug-in Module

Plug-in modules are packaged for delivery in binary format. The file extension of plug-in modules is `.nbm`. An NBM file is very similar to a JAR file, with only a few differences. The main differences between NBM files and JAR files are:

- An NBM file can contain more than one JAR filemodules can package any libraries they use into their NBM file.
- An NBM file contains metadata that can be used to display information about the module in an update center, such as the manifest contents, the license, etc.
- An NBM file can be signed for security purposes.

The IDE uses an Ant script to create NBM files. Therefore, you do not need to worry about the exact contents of an NBM filejust let the standard Ant build script for NBM creation take care of it for you. To create an NBM file:

1. In the Projects window, right-click the node of the module project you want to build.
2. Choose Create NBM.

If you are developing a module suite project, you have several additional choices to make. You can upload an image that will be used as the application's splash screen. In addition, you can set the name of the executable, which is the binary that the user actually starts, and the application's title bar. You can also let the IDE create a ZIP file of your application. When the IDE creates the ZIP file, it also creates a launcher, which it includes in the ZIP file. Finally, you can let the IDE create a JNLP file so that the application can be started over the Internet.

Once you have packaged a plug-in module, you can distribute it via an Update Center. To put together an Update Center, you must create an autoupdate descriptor that provides information about the plug-in module, such as the location of the NBM file and its module dependencies. An autoupdate descriptor is an XML file with a specific structure and content.

After placing the autoupdate descriptor and the NBM files on a server, you must make the URL to the autoupdate descriptor available to your target audience.



The autoupdate descriptor can be hosted on a different server from where the NBM files are found. Use the distribution element in the autoupdate descriptor to specify the location of the NBM file.

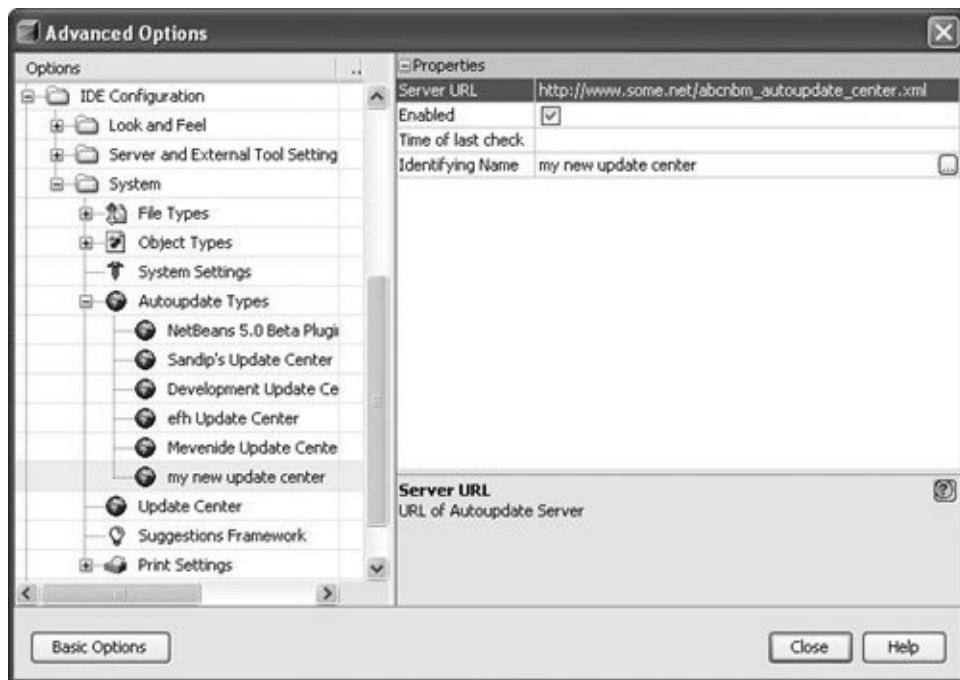
The URL looks similar to the following:

`http://www.netbeans.org/my_autoupdate_center.xml`

The target audience must then register the new autoupdate center by specifying the URL to the autoupdate descriptor in the Options window, as shown in [Figure 17-12](#).

Figure 17-12. Registering an autoupdate descriptor in the Options window

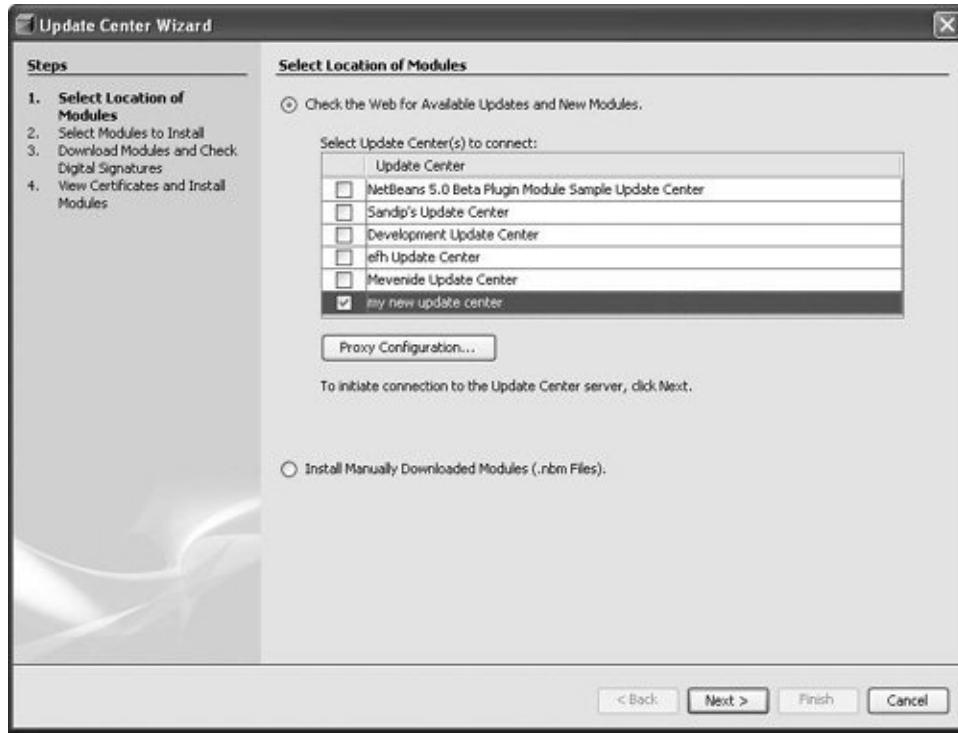
[\[View full size image\]](#)



Once NetBeans IDE knows the URL to the autoupdate center, you use the Update Center wizard to access the NBM files, as shown in [Figure 17-13](#).

Figure 17-13. Getting a plug-in module from an update center

[\[View full size image\]](#)



Updates to plug-in modules can be distributed in the same way. When you have developed a new version of a plug-in module, you can make it available via your Update Center. The same approach can be used for distributing modifications and patches to rich-client applications.

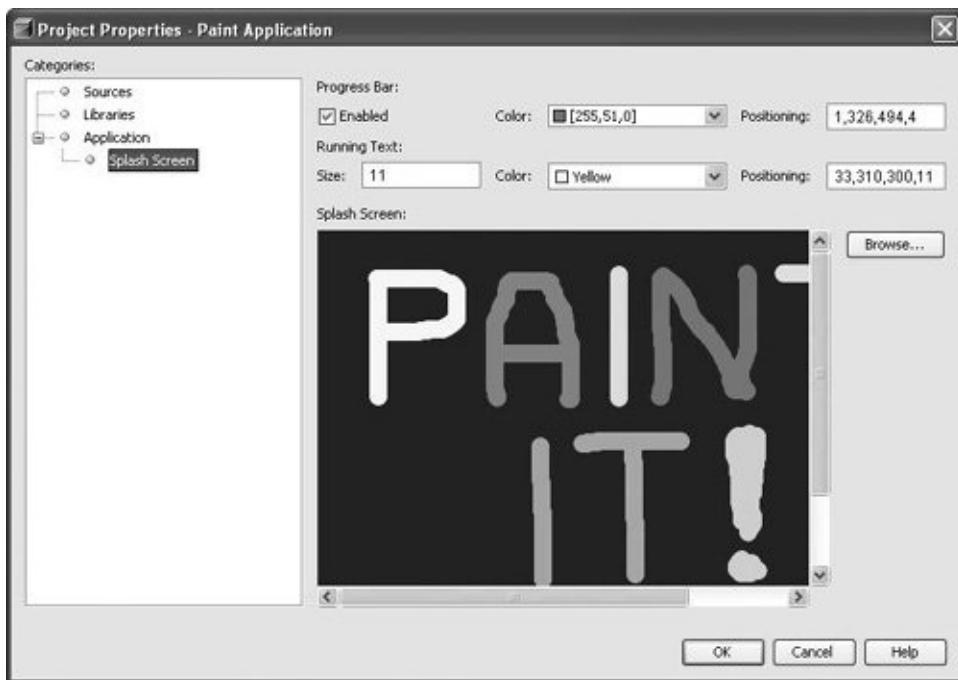
Packaging and Distributing a Rich-Client Application

Rich-client applications are complete, functioning, standalone desktop applications. NetBeans IDE provides the application framework on top of which you build your application. Each distinct part of the application is provided by a separate module project. For example, if your rich-client application is an editor, you might have one module project that provides syntax highlighting, while another provides file templates. Assembling the module projects within a module suite project allows you to package, distribute, and deploy them as a single unit.

Before you package a rich-client application, you need to consider whether you want to have it resemble NetBeans IDE. For example, your rich-client application uses the NetBeans IDE splash screen by default. Branding, the final stage before packaging, involves deciding things such as what the splash screen should look like and whether the application will include a progress bar during startup. In the module suite project's Project Properties dialog box, you define such settings, as shown in [Figure 17-14](#).

Figure 17-14. Branding support in NetBeans IDE

[\[View full size image\]](#)



When you right-click the module suite project's node in the Projects window, you can package the application by building a ZIP distribution, a JNLP application, or both. When the IDE creates your ZIP distribution, it also creates a launcher for the application. You can specify the name of the launcher in the Application panel of the module suite project's Project Properties dialog box. Also, you can use the same panel to define the title that will appear in the application's titlebar and the icon that will appear in the Help | About dialog box.

Once a rich-client application is branded and packaged, you can distribute it over the web as a web-startable JNLP application. Alternatively, you can distribute the ZIP file. Updates to the plug-in modules that make up a rich-client application can be distributed via the Update Center, as explained in the previous section.

Finding Additional Information

There are many resources available to find additional information on creating NetBeans plug-in modules and NetBeans rich client applications. The information presented in this chapter is intended to give you enough information to get you started developing NetBeans plug-in module and NetBeans rich client applications. Additional information on creating NetBeans plug-in modules and rich client applications can be found online.



Some of the links below are specific to module development support that was developed for NetBeans IDE 5.0. By the time you read this, there will probably be more features added to support module development and more documentation as well.

Basic Information

- <http://platform.netbeans.org>Project home page for the NetBeans Platform
- <http://platform.netbeans.org/tutorials/quickstart-nbm.html>NetBeans IDE 5.0 Plug-in Module Quick Start Guide
- <http://platform.netbeans.org/tutorials/nbm-paintapp.html>NetBeans IDE 5.0 Rich-Client Application Quick Start Guide

Tutorials and FAQs

- <http://platform.netbeans.org/tutorials>Tutorials for NetBeans Module (Plug-in) and Rich Client Application Development
- <http://platform.netbeans.org/faqs/index.html>NetBeans Developer FAQ
- <http://www.netbeans.org/kb/faqs/licence.html>NetBeans License FAQ
- <http://wiki.java.net/bin/view/Netbeans/DeveloperDocumentation>Module & Platform Development Work-in-Progress Page

Contributing

- http://www.netbeans.org/download/source_browse.htmlNet source code repository
- There are many NetBeans plug-in modules in the `contrib` directory of the NetBeans source code repository. Looking at some of the existing plug-in modules in the `contrib` directory may give you some ideas on how to implement your plug-in module. You can access the `contrib` directory of NetBeans source at <http://www.netbeans.org/source/browse/contrib>. You can also download the `contrib` directory with the IDE's CVS support. (Set your CVS root to `:pserver:anoncvs@cvs.netbeans.org:/cvs` and specify the `contrib` module.)

Mailing Lists

- nbdev@netbeans.orgthe NetBeans development mailing list, used by the developers of the NetBeans IDE and the NetBeans Platform
- dev@openide.netbeans.orgthe NetBeans APIs mailing list, used by developers who are creating plugin-modules for the NetBeans IDE and NetBeans rich-client applications

Chapter 18. Using NetBeans Developer Collaboration Tools

- [Getting the NetBeans Developer Collaboration Tools](#)
- [Configuring NetBeans IDE for Developer Collaboration](#)
- [Creating a Collaboration Account](#)
- [Managing Collaboration Accounts](#)
- [Logging into a Collaboration Server](#)
- [Collaborating and Interacting with Developers](#)

MANY SOFTWARE DEVELOPERS WORK ON DISTRIBUTED DEVELOPMENT TEAMS. Developing software with distributed teams can be challenging since developer interactions are not as simple as walking down the hall to another developer's office. To help address some of these challenges, NetBeans IDE offers an optional set of Developer Collaboration tools. These tools are available as NetBeans plug-in modules and are available on the NetBeans IDE Update Center. The Developer Collaboration tools can greatly enhance the productivity of distributed software development teams by allowing developers to instant message and share their IDE projects with each other. The NetBeans Developer Collaboration tools are so advanced that you can grant a remote developer permission to view, edit, compile, and run your project source code all from within your IDE. You can share an IDE project by merely dragging and dropping the project or a selection of source files into a shared collaboration window. All developers participating in the collaborating session can see the same source files once

they are shared. Any modifications made to the shared project are saved to the shared IDE project.

This chapter describes how to install the NetBeans Developer Collaboration modules and how to take advantage of those modules to increase productivity when working with remote team members.

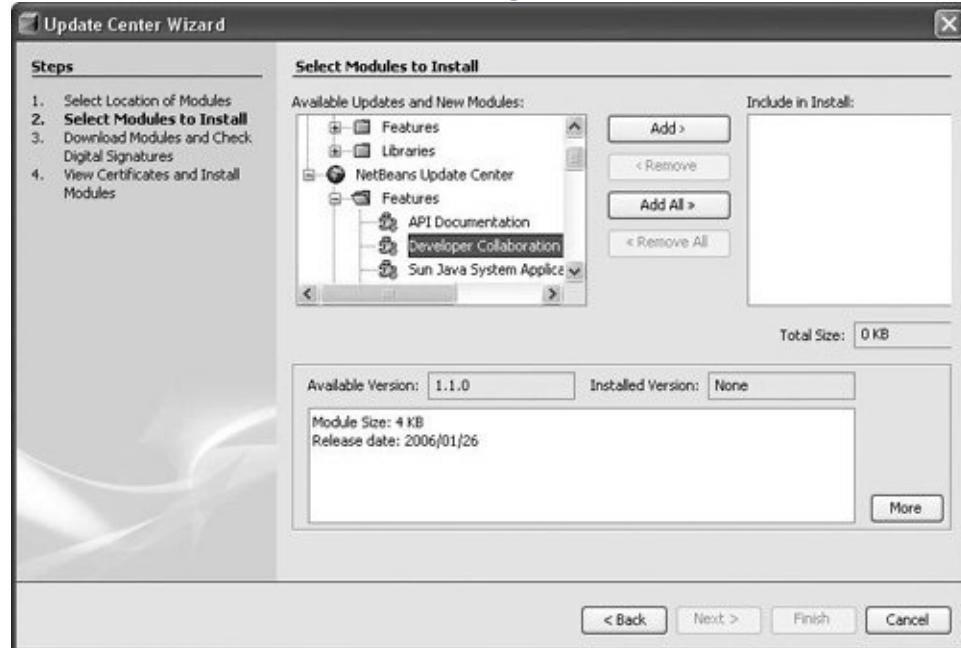
Getting the NetBeans Developer Collaboration Tools

The NetBeans Developer Collaboration tools come in the form of optional modules for NetBeans IDE 5.0.

To download and install the Developer Collaboration modules, launch NetBeans IDE and select the Tools | Update Center menu item. In the Update Center wizard, select Developer Collaboration from the Update Center as shown in [Figure 18-1](#).

Figure 18-1. Update Center panel

[[View full size image](#)]



To continue with the installation of the Developer Collaboration modules, click through the Update Center wizard pages to

accept the license agreements and to download and install the necessary modules.

Once you have downloaded and installed the Developer Collaboration modules, you will notice a new menu called Collaborate and two new icons in the main toolbar.

Configuring NetBeans IDE for Developer Collaboration

Before you can use NetBeans Developer Collaboration, you need a collaboration account on a collaboration server or collaboration service. There is a free public collaboration service available from www.java.net. This service is hosted on a collaboration server called share.java.net.

If you find you have constraints that prohibit the use of a public collaboration server, you have the following alternatives.

- Install and configure the Sun Java System Messaging Server.

The Sun Java System Messaging Server is free and can be downloaded at:

http://www.sun.com/software/products/messaging_srver/index.html

You will need to follow the Sun Java System Messaging Server instructions to successfully install and configure Sun Java System Messaging Server so it may be used as a collaboration server for NetBeans Developer Collaboration.

- Install a JXTA peer-to-peer Collaboration plug-in module into NetBeans IDE.

At the time of this writing, a peer-to-peer JXTA-based plug-in is being planned to be made available that allows you to use NetBeans Developer Collaboration without requiring a dedicated collaboration server such as Sun Java System Messaging Server.

The installation and configuration for the JXTA based plug-in requires only a download of the plug-in and installation into

NetBeans IDE. The installation and configuration of the JXTA plug-in will be significantly less than required by the Sun Java System Messaging Server.

However, the JXTA-based plug-in limits developer conversations to two participants, whereas a collaboration server, such as Sun Java System Messaging Server, allows many more than two developers to participate in a collaboration session or conversation.

The NetBeans IDE development team has been testing the Developer Collaboration plug-in modules with open source messaging server software called jabberd and is actively contributing fixes to the open source jabberd project. At the time of this writing, it is not recommended to use jabberd as a collaboration server with NetBeans Developer Collaboration due to stability reasons. But, this may change as additional contributions are made to the jabberd project.

Creating a Collaboration Account

This section describes the task of creating a collaboration account. A collaboration account must be created in order to collaborate with other developers using the NetBeans Developer Collaboration modules. The example collaboration account information presented in this section uses a free collaboration service available on the Internet site java.net. The collaboration server at java.net is identified as share.java.net.

If you are using an alternative collaboration server solution, much of the account creation described here will be the same. The major exception will be the collaboration server name and possibly the port number offering the collaboration service.

To create a collaboration account you can either choose **Collaborate | Login** from the main menu or click on the **Login** icon in the main toolbar of NetBeans IDE. This will launch the Collaboration Login window as shown in [Figure 18-2](#).

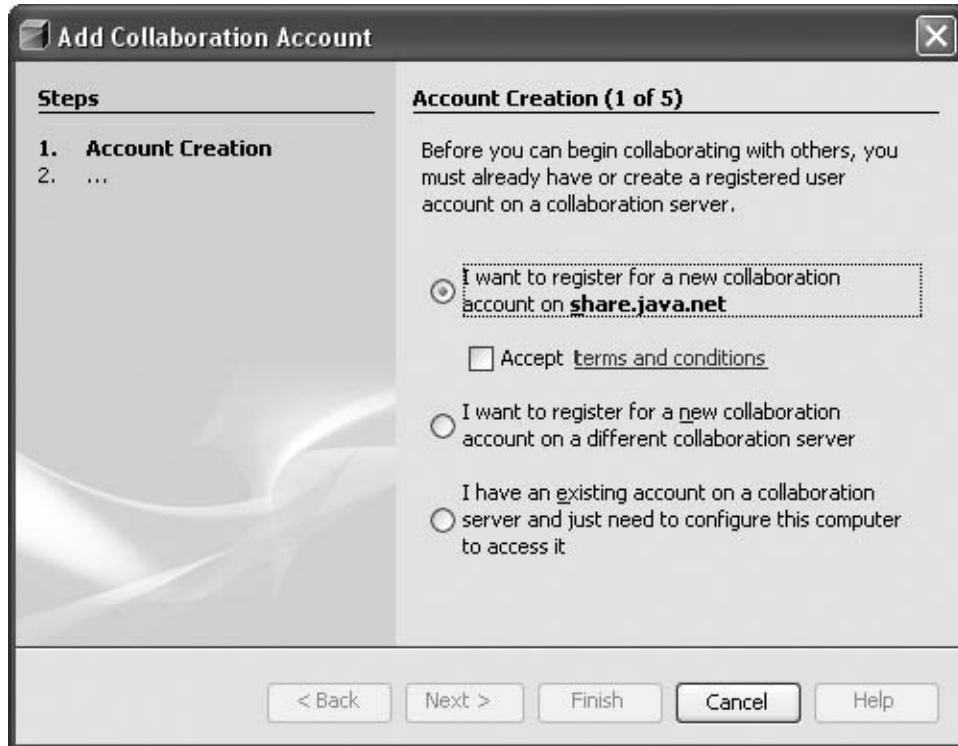
Figure 18-2. Collaboration Login window



The default location of the Collaboration Login window is the right-hand side of the IDE.

When a collaboration account has not been created, the only button available in the Collaboration Login window is Add Account. Click Add Account in the Collaboration Login window to add a collaboration account. This launches the Add Collaboration Account wizard as in [Figure 18-3](#).

Figure 18-3. Collaboration Account Creation



There are several options for creating a collaboration account:

- Use the free public share.java.net collaboration service.

To register for a new collaboration account on share.java.net, you must also accept the terms and conditions of the share.java.net collaboration service.

- Register for a new collaboration account on a different collaboration server.

You would choose this option if you have installed an alternative collaboration service such as one of those identified in the section Configuring NetBeans IDE for Developer Collaboration above and wish to create a login account on that alternative collaboration server.

- Use an existing account on a collaboration server.

Use this option if you already have an account on a collaboration server. For instance, if you already have a collaboration account on share.java.net, you would use this option to configure the NetBeans Developer Collaboration modules to use that collaboration account. Likewise, if you have a collaboration account on an alternative collaboration server, you would also use this option.

If you need to create a new collaboration account, regardless of whether it is on share.java.net, you must provide the following information:

- A display name that will identify you to others using NetBeans Developer Collaboration.
- An account name that is the name you use to login into the collaboration server.
- The collaboration server name and whether the Developer Collaboration modules should use a proxy to connect to the collaboration server.

For example, to create a collaboration account on share.java.net, the Add Collaboration Account wizard in the IDE will ask for a display name that will identify you to other developers on share.java.net. As shown in [Figure 18-4](#), John Smith is the display name that other developers who are using the share.java.net collaboration server will see.

Figure 18-4. Collaboration Account Name

[\[View full size image\]](#)



In the next panel of the Add Collaboration Account wizard, be sure to append the port number if you have a dedicated port number defined for the collaboration service on the target collaboration server. If you are creating an account on the free collaboration server at share.java.net, the wizard will automatically fill in the Server Hostname field with `share.java.net:5222`.

- Proxy information, if applicable.

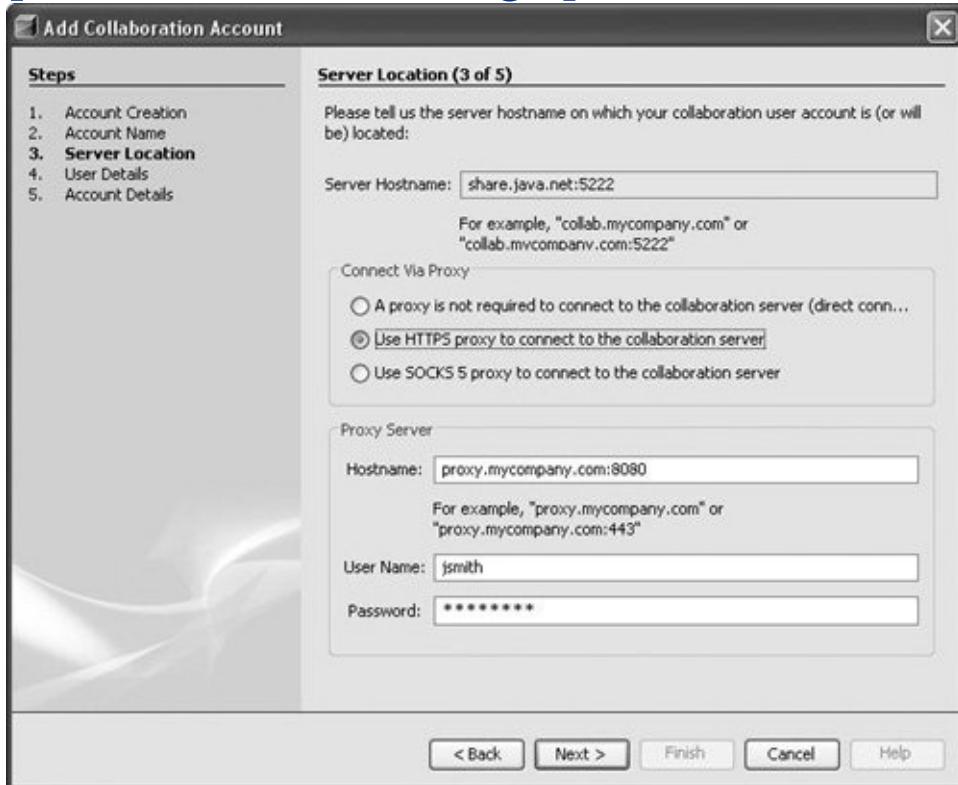
If you require an HTTPS or SOCKS 5 proxy to connect to your collaboration server, select the appropriate option to specify an HTTPS or SOCKS 5 proxy and provide the necessary proxy server host name.

Also specify a username and password for the proxy server if one is required. If a username and password is not required for the proxy server, you can leave those fields blank. Shown in [Figure 18-5](#) is the Server Location portion of the Add Collaboration Account wizard for a collaboration account being created on share.java.net that requires an HTTPS proxy with a host name of proxy.mycompany.com

using port 8080 along with a user name `jsmith` and `jsmith`'s password.

Figure 18-5. Collaboration Server Location

[View full size image]



After specifying the collaboration server location and proxy server if needed, you must provide additional information about yourself, including your first name, last name, and an email address where the collaboration server can notify you of planned collaboration service maintenance such as a server upgrade. Following the example used above, you would enter John as the first name and Smith as the last name, along with providing John's email address.

- On the Account Details page of the Add Collaboration Account wizard, you specify a user name and password to

use with the collaboration server. For example, for a user name jsmith and a chosen password, the last page of the Add Collaboration Account wizard would look as shown in [Figure 18-6](#).

Figure 18-6. Collaboration Account Details



After clicking on the Finish button to complete the Add Collaboration Account wizard, the Collaboration Login window will be updated with the account information you have just added.



If you would like to automatically be logged into your collaboration server whenever the IDE is launched, you can select the Login Automatically checkbox in the Collaboration Login window.

Managing Collaboration Accounts

Once you have created a collaboration account, you can use the Account Management window to create an additional collaboration account, modify an existing collaboration account, set a given collaboration account as the default collaboration account, or remove a collaboration account.

To open the Account Management window:

1. Open the Collaboration window.

You can open the Collaboration window by choosing Window | Collaboration or by pressing Control-Shift-L.

2. Click the Manage Accounts link in the Collaboration window as shown in [Figure 18-7](#).

Figure 18-7. Manage Collaboration Accounts



Once you have clicked on the Manage Accounts link, you have the capability to add additional collaboration accounts, modify an existing collaboration account, set an existing account as the default collaboration account, or remove an existing collaboration account. Each of these tasks is described in further detail in the following sub-sections.

Adding Additional Collaboration Accounts

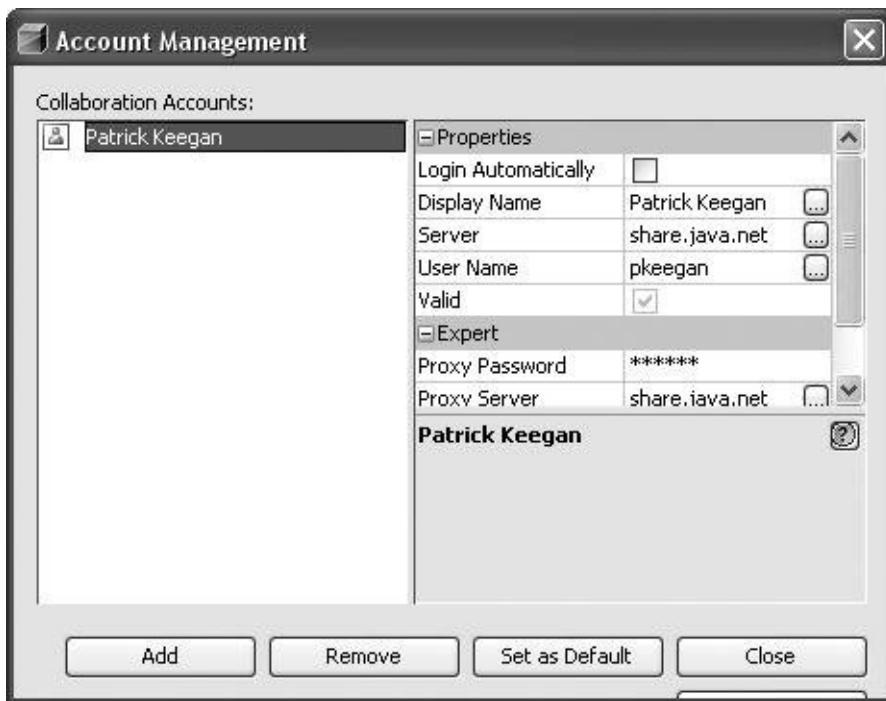
The NetBeans Developer Collaboration modules allow you to have multiple collaboration accounts. Multiple collaboration accounts can be useful if you have multiple projects with constraints that do not allow all the developers you interact with to share the same collaboration server. For example, you may contribute to an open source project such as NetBeans in addition to working with a team of developers at your company on an internally-used application. In this scenario, it makes sense to have two different collaboration accounts.

To add an additional collaboration account:

1. In the Collaboration Login window, click Manage Accounts.

The Account Management dialog box appears like the one shown in [Figure 18-8](#).

Figure 18-8. Collaboration Account Management



2. In the Account Management window, click the Add button.

This will launch the Add Collaboration Account wizard you saw when you created your first collaboration account.

3. Follow the instructions provided in the wizard to add an additional collaboration account.

Once you have provided that information, you will be returned to the Account Management dialog. The Account Management dialog will now be updated with your additional collaboration account.

Setting a Default Collaboration Account

If there is a collaboration account you tend to use more frequently than others, you may find it useful to set that collaboration account as the default collaboration account. To set a default collaboration account:

1. In the left-hand side of the Collaboration Account Management window, select a collaboration account.
2. Click the Select as Default button.

The default collaboration account is identified in the Collaboration Account Management window with a little green colored check next to the collaboration account name. The default collaboration account is also the one shown as the default collaboration login account in the drop-down list on the Collaboration Login window.

Modifying a Collaboration Account

There are times when you need to make modifications to the configuration information for your collaboration accounts. For instance, the system administrator responsible for maintaining a collaboration server you use may decide to change the host name of the collaboration server. Or, you may be a user who travels with a laptop and as a result may need to update how you access a collaboration server.

To update a collaboration account:

- In the Collaboration Login window, click Manage Accounts to launch the Collaboration Account Management dialog box.

From the Account Management dialog you can modify properties and expert settings for a selected collaboration account. For example, to update your user name you can click in the box to the right of the User Name Property or click on the ellipsis and update your username.

To update an expert setting such as the proxy to use to connect to the collaboration server, click on the drop-down box for the proxy expert setting and select the appropriate

option for connecting to your collaboration server.

Removing a Collaboration Account

As you complete development on projects or leave a development project, you may have the need to remove a collaboration account.

To remove a collaboration account:

- 1.** In the Collaboration Login window, click Manage Accounts.
- 2.** In the Collaboration Account Management dialog, select the collaboration account that you want to remove and click Remove.

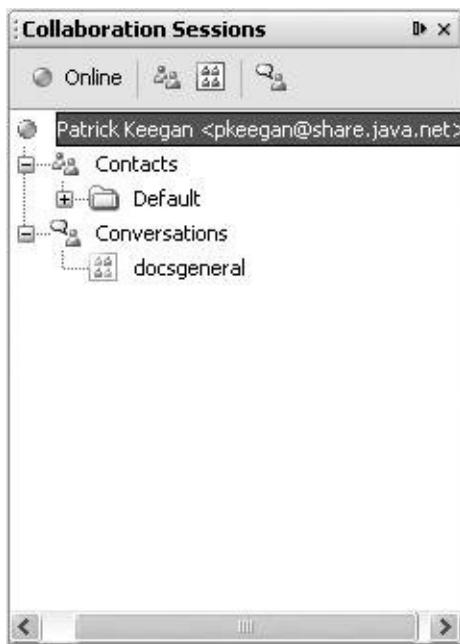
Logging into a Collaboration Server

Once you have created a collaboration account, you may log into a collaboration server. To log into a collaboration server, you select the collaboration account you wish to use in the Collaboration Login window and click the Login button in the Collaboration Login window.

You can configure the IDE to automatically log you into a collaboration server any time you launch the IDE by checking the Login Automatically checkbox in the Collaboration Login window.

When you press the Login button, the IDE will attempt to log into the selected collaboration server. If the login is not successful, you will receive an error dialog indicating the type of error encountered and a suggested corrective action. If the login attempt is successful, the Collaboration Login window will be updated with information retrieved from the collaboration server such as that shown in [Figure 18-11](#) for John Smith.

Figure 18-9. Collaboration Sessions



You can also log into multiple collaboration accounts at the same time.

Collaborating and Interacting with Developers

Once you have logged into a collaboration account, you can begin to interact with other developers. Before interacting with other developers you will need to add those developers as contacts with whom you can collaborate.

Contacts and Contact Groups

When a collaboration session is first started you will see the Contacts and Conversations list in the Collaboration Sessions window. Contacts are developers with whom you wish to collaborate.

You can organize contacts by creating folders under the Contacts list to categorize the developers you collaborate with. For instance, you might choose to create a contact folder based on a project name.

To create an additional contact folder:

- 1.** In the Collaboration Sessions window, right-click Contacts and choose Add Contact Group.
- 2.** Type the name for a new contact group and click OK.

You can also rename a contact group once it is created by right-clicking on the contact group folder and choosing Rename. When you no longer have a need for a contact group you can remove it by right-clicking on the contact group and choosing Delete.

To add a developer to a contact list:

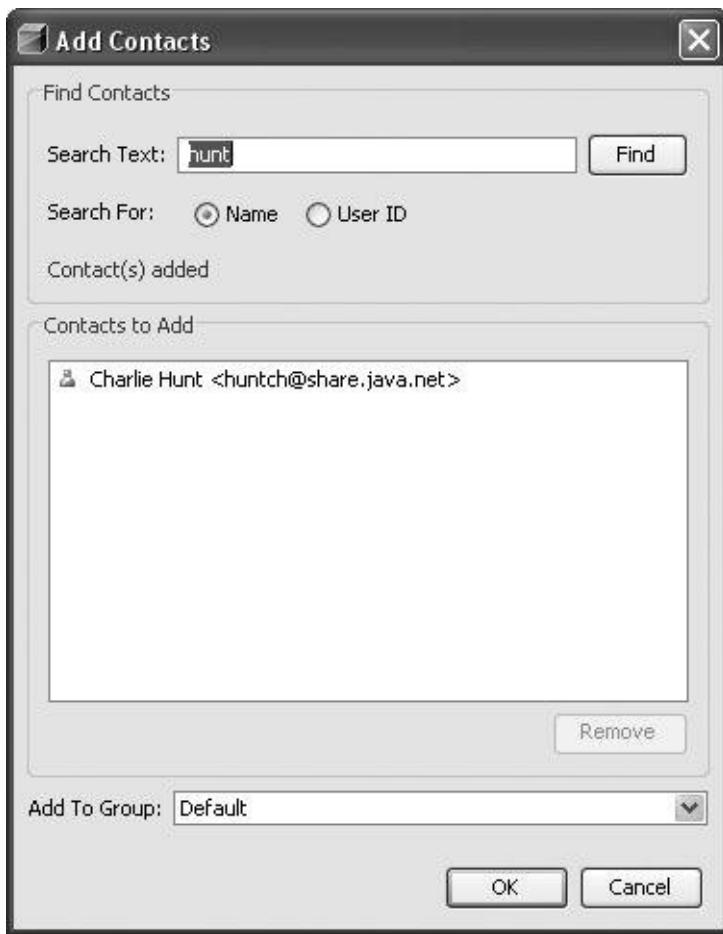
- 1.** In the Collaboration Sessions window, right-click the

Contacts node or the contact folder that you wish to add developers to and choose Add Contacts.

2. In the Add Contacts dialog, search for the developer by his or her display name or by user ID.
3. Once you have found the developer, use the Add to Group drop-down list to select the group to which you want to add that developer.

For example, if you add a developer who has a username of Charlie Hunt on the share.java.net collaboration server to a contact group named default, the Add Contacts dialog would look like the dialog shown in [Figure 18-10](#).

Figure 18-10. Add Contacts



4. Click OK.

Conversation Groups

To help facilitate and manage collaboration activities, the NetBeans Developer Collaboration feature provides conversation groups. Conversation groups are a good way to organize a group of developers in a meeting-like setting. For example, if you would like to schedule a code review with several developers, you can create a conversation group for the code review and ask those developers to join the conversation.

To create a conversation group:

1. In the Collaboration Sessions window, right-click

Conversations and choose Create Public Conversation.

2. In the Create Conversation dialog box, enter a conversation name and click OK.

After you have created a conversation group, the conversation group will be displayed in the Collaboration Session window under Conversations. For example, after creating a conversation called "codereview," the Collaboration Session window will show "codereview" as a conversation as shown in [Figure 18-11](#).

Figure 18-11. Adding conversations



You can add as many conversations to the conversation list as you wish.

Additional developers can be invited to participate in the conversation once a conversation group has been created.

To invite developers to participate in a conversation group:

1. In the Collaboration Sessions window, right-click on a conversation group and choose Manage.
2. In the Manage Public Conversation dialog box, click Add.
3. In the Add Users dialog box, enter the developer's name or user ID of who you wish to invite to the conversation and click Find. You can search for a developer's name or user ID if you do not recall his or her name or user ID.
4. Select the developer to add to the conversation group.
5. Grant access rights for the developer you are adding to your conversation group and click OK.



You can add as many developers to a conversation group as you wish.

Starting and Joining Conversations

There are several ways to initiate a conversation with other developers. One way is to set up a conversation group, which was described above. You can also join an existing conversation group or initiate a new conversation directly with another developer.

To join an existing conversation, you first must find the conversation group you wish to join. You do this by right-clicking Conversations in the Collaboration Sessions window and choosing Subscribe to Public Conversation. In the Subscribe to Public Conversation dialog, you either search or browse for the conversation you wish to join. Then select the conversation you

want join and click OK to subscribe to the conversation. Once you have subscribed to a conversation, the subscribed conversation will show up in the Conversations lists in the Collaboration Session window.

To join a conversation you are already subscribed to, right-click the conversation name in the Collaboration Sessions window and choose Open.

To initiate a conversation with an individual developer, right-click the developer's username in the Contacts list and choose Start Conversation.

Once you have started or joined a conversation, the IDE will open a set of windows in the editor area detailing the participants of the conversation along with status information on who has joined the conversation, etc.

Interacting with other Developers

The NetBeans Developer Collaboration modules provide rich capabilities to communicate and share artifacts with other developers within the conversation window. The conversation window provides information on the participants in the conversation and their status (e.g., online, idle, or away).

To send an instant message to the developers who are participating in a collaboration conversation, you enter text in the lower window and press the Send button. After you have sent an instant message, you and the other participants see your message in the window just above where you entered your text.

As a sender of a message, you have options to format your text as formatted text, Java source code, HTML or XML. You can also insert smiley faces.

In addition, you can have the instant message sending window allow you to use the Enter and Tab characters in the message. (By default, when you press Enter, the text you have entered will be sent.) These options are particularly useful when copying source code, HTML, or XML to the instant message sending window.

Sharing IDE Projects

One of the most useful features of the NetBeans Developer Collaboration modules is the ability to share NetBeans projects and source code with other developers. It is often very useful to have other developers look at each other's source code, especially when there is a bug that is proving difficult to isolate or a development workflow type of activity such as a code review, code inspection or paired programming.

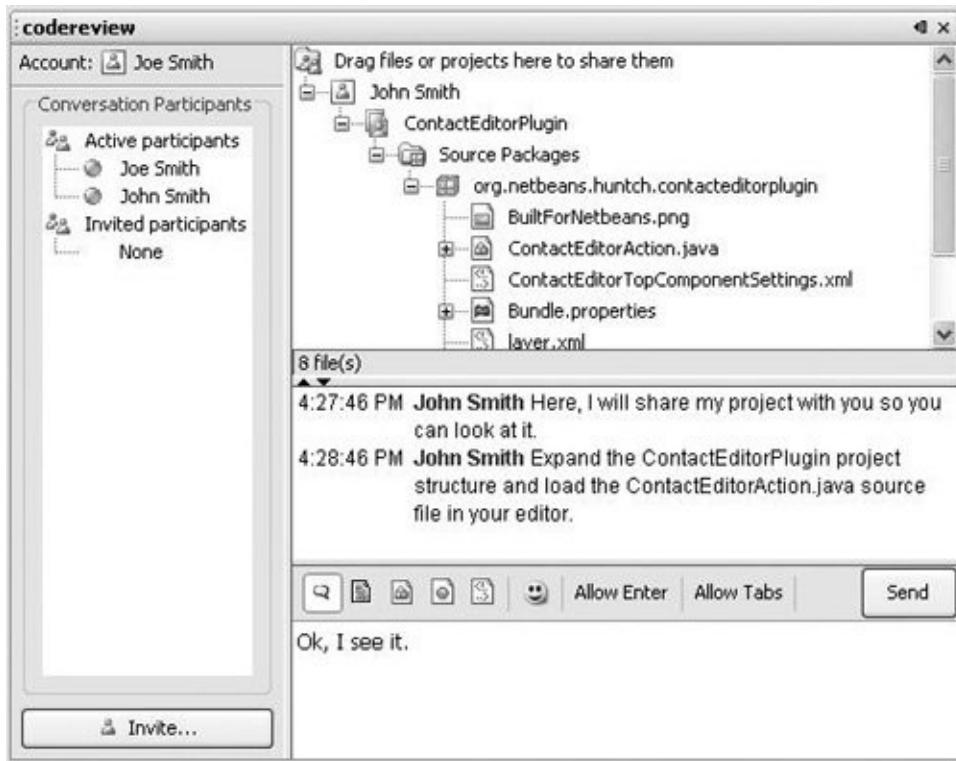
To share source code or a project:

- Drag and drop the project or files from the Projects window into the sharing area of the conversation window (top window).

When you share a project or file, the conversation window will show the shared project or file as shown in [Figure 18-12](#).

Figure 18-12. Sharing NetBeans IDE projects

[\[View full size image\]](#)



When source code or an IDE project is shared with another developer via collaboration, both developers view the same source files in their respective source code editors. In addition, as changes are made to the source code or IDE project, those changes are viewed by both developers.

Projects that are shared can also have the following IDE actions performed on them: Build Project, Clean and Build Project, and Run Project.

To invoke any of these actions on a shared project, right-click the project and select the desired action as shown in [Figure 18-13](#).

Figure 18-13. Shared project actions



The IDE's Source Editor window and Output window on both systems are updated with the same information as changes or actions take place on shared projects and files. When projects and files are shared, no project source code or metadata is copied to other developers' systems as part of sharing. Only one copy of the project exists. Any edits to shared project and files along with any actions performed on shared projects and files are updated only on the system of the developer who shared the projects and files.

Appendix A. Importing an Eclipse Project into NetBeans IDE

- [Getting the Eclipse Project Importer](#)
- [Choosing Between Importing with and Importing without Project Dependencies](#)
- [Importing an Eclipse Project and Preserving Project Dependencies](#)
- [Importing an Eclipse Project and Ignoring Project Dependencies](#)
- [Handling Eclipse Project Discrepancies](#)
- [Handling Eclipse Project Reference Problems](#)

IF YOU HAVE PROJECTS THAT YOU HAVE STARTED DEVELOPING IN ECLIPSE, you can easily migrate those projects to NetBeans IDE. The Eclipse Project Importer automates the importing of Eclipse projects by processing the Eclipse project metadata and mapping it directly to new NetBeans projects. For NetBeans IDE 5.0, this module works for Eclipse 3.0 and 3.1 projects and is available through the NetBeans Update Center.

The NetBeans Eclipse Project Importer not only identifies and automatically fixes Eclipse project discrepancies, but also identifies erroneous Eclipse project references and suggests corrective actions to resolve them once the project has been imported into NetBeans IDE. The project importer is also very flexible, because you can select projects from an Eclipse workspace where project dependencies are automatically

identified and marked for importing if such project dependencies exist. In addition, the project importer lets you select individual Eclipse projects for importing into NetBeans IDE without specifying the location of the Eclipse workspace.

This appendix describes how to import projects both from Eclipse workspaces and from specific Eclipse projects.



If you find the project importer is not importing your Eclipse projects as well as you had hoped, you can use a NetBeans IDE project template to create a project using your sources from your Eclipse project. Choose File | New Project in NetBeans IDE and select the "with Existing Sources" template from the appropriate project category (General, Web, or Enterprise).

If your Eclipse project uses a sophisticated Ant script (`build.xml` file), you might want to use one of the "with Existing Ant Script" project templates. See [Chapter 16](#) for more information on working with projects based on an existing Ant script.

Getting the Eclipse Project Importer

The NetBeans Eclipse Project Importer is an optional module for NetBeans IDE 5.0, so it is not in the standard download.

To download and install the Eclipse Project Importer module, launch NetBeans IDE and choose Tools | Update Center from the main menu. In the Update Center, navigate to and select the Eclipse Project Importer module, click the arrow button to add the module to the list of modules to download, and then complete the wizard.

Once you have downloaded and installed the Eclipse Project Importer module, you are ready to begin migrating an Eclipse project to NetBeans IDE.

Choosing Between Importing with and Importing without Project Dependencies

As mentioned earlier, NetBeans IDE's Eclipse Project Importer provides two options for importing Eclipse projects into NetBeans IDE:

- Selecting Eclipse projects from an Eclipse workspace, where project dependencies can be automatically identified, marked, and selected by NetBeans IDE's Eclipse Project Importer.

This option preserves project dependencies and consequently makes it very easy to import Eclipse projects that have dependencies on other Eclipse projects. The project importer automatically determines the project dependencies for you. You can also use this option to import an Eclipse project that has no additional project dependencies.

- Importing only the specified project and ignoring any project dependencies that may exist with other Eclipse projects.

This option is best suited for situations in which you do not have an Eclipse workspace but do have access to an Eclipse project.



In most cases, you should use the option to import Eclipse projects using an Eclipse workspace so that project dependencies are preserved. The option to import Eclipse projects directly while ignoring project dependencies is most applicable when an Eclipse workspace is no longer available as a result of an action such as an Eclipse IDE uninstall that resulted in an Eclipse workspace removal.

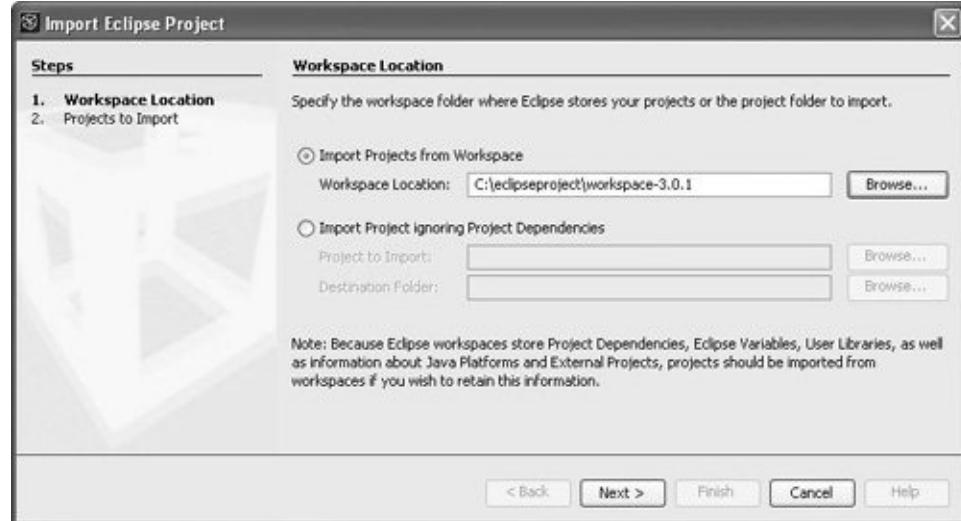
Importing an Eclipse Project and Preserving Project Dependencies

This section describes the procedure necessary to import an Eclipse project from an Eclipse workspace. When using this option to import Eclipse projects, the NetBeans Eclipse Project Importer automatically determines any project dependencies and imports those projects as well.

To initiate an import of an Eclipse project along with any additional dependent projects, choose File | Import Project | Eclipse Project in NetBeans IDE. From the Import Eclipse Project wizard, specify the location of the Eclipse workspace you wish to import from. An example is shown in [Figure A-1](#).

Figure A-1. Import Eclipse Project wizard, Workspace Location panel

[\[View full size image\]](#)

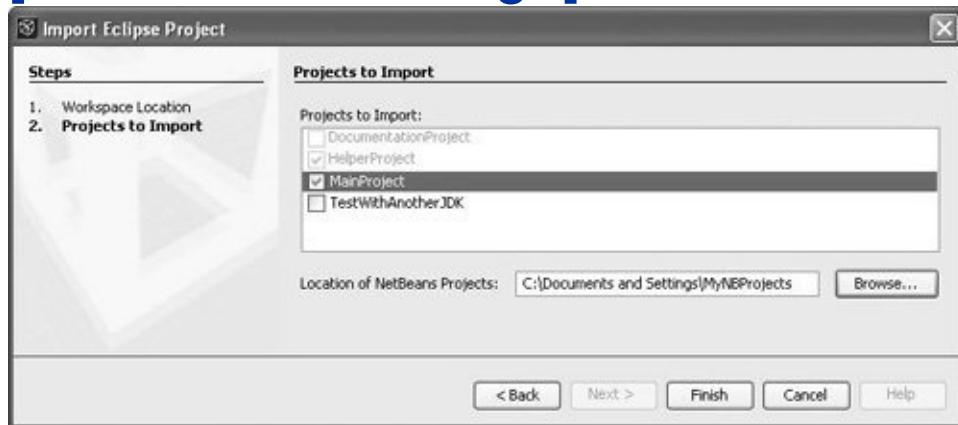


Once you have specified the location of the Eclipse workspace, click Next to display the Projects to Import portion of the Import Eclipse Project wizard. This portion of the wizard asks you which Eclipse projects you would like to import and where to create the imported project(s) for NetBeans IDE.

If the selected Eclipse project has dependencies to other projects, the project importer automatically determines this and marks those dependent projects for import also. In the example in [Figure A-2](#), MainProject was initially selected, but because MainProject depends on HelperProject, HelperProject is also marked for importing.

Figure A-2. Import Eclipse Project wizard, Projects to Import panel

[[View full size image](#)]



To complete the Import Eclipse Project wizard and complete the importing of the Eclipse project(s) into NetBeans IDE, click the Finish button.

If NetBeans IDE discovers discrepancies in the Eclipse project(s) while processing the Eclipse project information and believes

that it can resolve those discrepancies automatically, the project importer displays a warning dialog box, showing any Eclipse project discrepancies it has found and the corrective action NetBeans IDE has taken. If you see this dialog box, no additional work is required. For more information on Eclipse project discrepancies, see [Handling Eclipse Project Discrepancies](#) later in this appendix.

If the project importer discovers problems in an Eclipse project, such as a project resource that could not be found, a warning dialog box is displayed, along with appropriate actions for you to take to resolve these reference problems once the project has been imported. This dialog box presents a guide to correcting problems that have been detected in the Eclipse project that the project importer cannot correct itself. For more information on Eclipse project reference problems, see [Handling Eclipse Project Reference Problems](#) later in this appendix.

If the project importer has found no project reference problems in the Eclipse project you are importing, you can begin using NetBeans IDE on your newly imported Eclipse project(s). For instance, you can immediately begin using the IDE's Projects window to traverse the imported project(s), or you can begin running the newly imported Eclipse projects by right-clicking the newly imported project's node in the Projects window and selecting Run Project from the contextual menu.

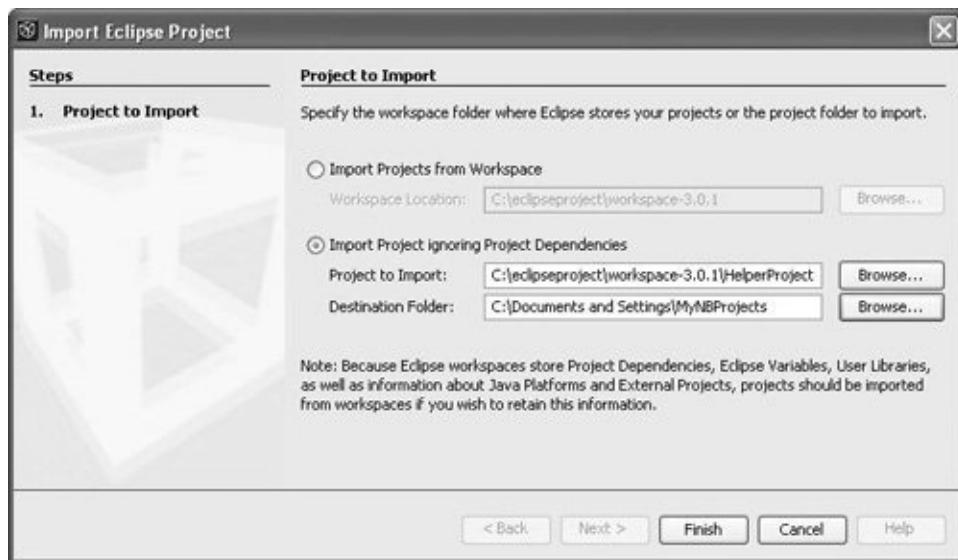
Importing an Eclipse Project and Ignoring Project Dependencies

This section describes the procedure necessary to import an Eclipse project when an Eclipse workspace is not available or when you want to import a specific project and ignore project dependencies. When using this option to import Eclipse projects, the project importer ignores any project dependencies that may exist between projects.

To initiate an import of an Eclipse project and ignore any project dependencies that may exist, choose File | Import Project | Eclipse Project in NetBeans IDE. In the Import Eclipse Project wizard (shown in [Figure A-3](#)), select the Import Project Ignoring Project Dependencies radio button, and specify the Eclipse project to import and the destination folder where the project should be imported.

Figure A-3. Import Eclipse Project wizard, Project to Import panel

[\[View full size image\]](#)



Once you have specified the location of the Eclipse project to import and the destination folder, click Finish to complete the project import.

If NetBeans IDE discovers discrepancies in the Eclipse project while processing the Eclipse project information and believes that it can resolve those discrepancies automatically, the project importer displays a warning dialog box (like the one in [Figure A-4](#)), showing any Eclipse project discrepancies it has found and the corrective action NetBeans IDE has taken. If you see this dialog box, no additional work is required. For more information on Eclipse project discrepancies, see [Handling Eclipse Project Discrepancies](#) later in this appendix.

Figure A-4. Notification of inconsistencies in the imported Eclipse project that NetBeans has corrected

[[View full size image](#)]



If the project importer has discovered problems in your Eclipse project, such as a project resource that could not be found, NetBeans IDE displays a Reference Problems warning dialog box (like the one in [Figure A-5](#)), along with appropriate actions for you to take to resolve these Eclipse project reference problems once the project has been imported into NetBeans. This dialog box is presented so that you may correct problems that have been detected in the Eclipse project while the project importer is processing the Eclipse project information. For more information on Eclipse project reference problems, see [Handling Eclipse Project Reference Problems](#) later in this appendix.

Figure A-5. Notification of reference problems in the imported project that you must resolve after closing the dialog box



If the project importer has found no project reference problems in the Eclipse project you are importing, you can begin using NetBeans IDE on your newly imported Eclipse project, as long as you do not have other dependent projects that must be imported. If there are no additional project dependencies for the project you have just imported, you can immediately begin using the IDE's Projects window to traverse the imported project(s), or you can begin running the newly imported Eclipse projects by right-clicking the newly imported project's node in the Projects window and selecting Run Project from the contextual menu.

Handling Eclipse Project Discrepancies

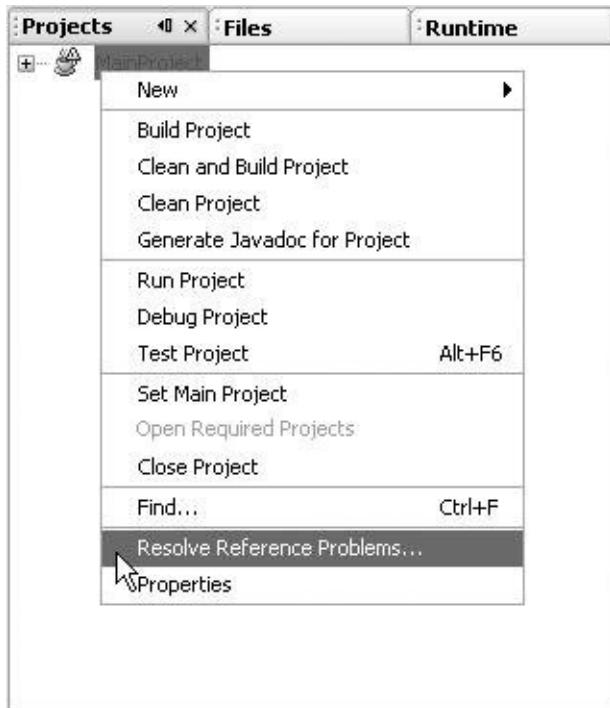
If the NetBeans Eclipse Project Importer identifies Eclipse project discrepancies while importing an Eclipse project, a warning dialog box is displayed, explaining the Eclipse project discrepancies discovered, along with the action NetBeans will take. You do not need to take any additional action when this warning dialog box appears. See [Figure A-4](#) for an example of this warning.

Handling Eclipse Project Reference Problems

If the NetBeans Eclipse Project Importer identifies an Eclipse project reference problem while processing an Eclipse project, a warning dialog box is displayed. This dialog box reports the project reference problem and suggests the corrective action you should take. [Figure A-5](#) shows an example of this warning dialog box.

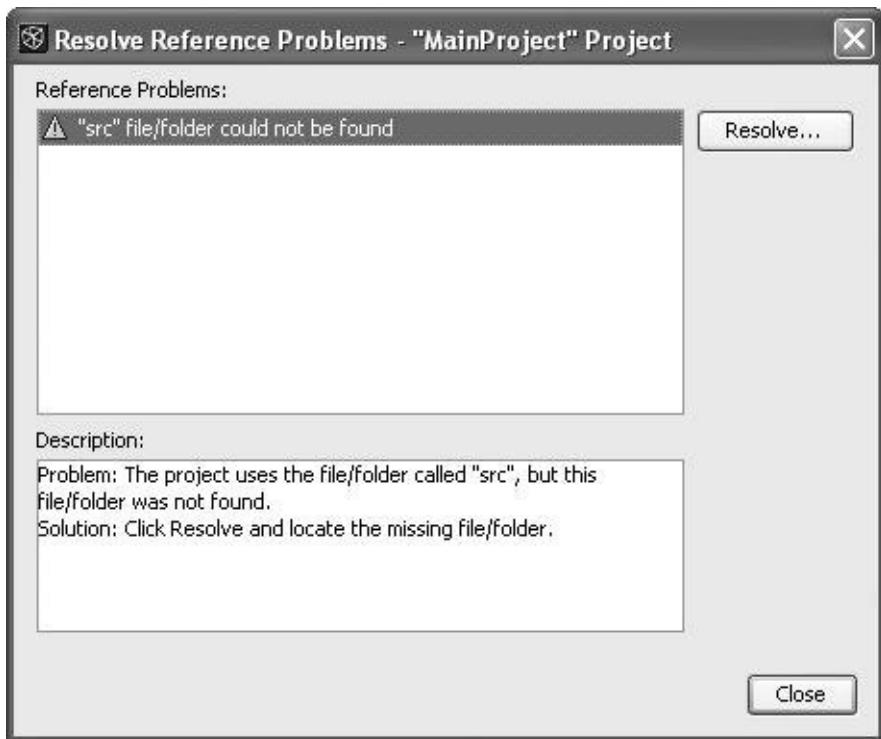
To resolve these detected Eclipse project reference problems, go to the NetBeans Projects window, right-click the project that has just been imported, and select Resolve Reference Problems from the contextual menu (see [Figure A-6](#)).

Figure A-6. Contextual-menu item for resolving reference problems in an imported Eclipse project



After you select Resolve Reference Problems, the Resolve Reference Problems dialog box (shown in [Figure A-7](#)) is displayed, identifying what NetBeans has found for Eclipse project reference problems.

Figure A-7. Resolve Reference Problems dialog box



The NetBeans Resolve Reference Problems dialog box describes the Eclipse project reference problems the project importer has found in the imported Eclipse project, along with a suggested solution. To resolve the discovered Eclipse project reference problem(s), simply click the Resolve button. Although the project importer cannot automatically resolve all problems it

has found with Eclipse projects, it does give you suggested solutions as to how to resolve the Eclipse project reference problems it has found.

Appendix B. Importing a JBuilder Project into NetBeans IDE

- [Getting the JBuilder Project Importer](#)
- [Importing a JBuilder 2005 Project](#)
- [Project Import Warnings](#)
- [Running the Imported Project](#)

IF YOU HAVE PROJECTS that you have been developing with JBuilder 2005, you can easily migrate those projects to NetBeans IDE 5.0. The JBuilder Project Importer automates the importing of JBuilder projects by processing JBuilder project metadata and mapping it directly to new NetBeans IDE projects including projects with dependencies. For NetBeans IDE 5.0, this module works for JBuilder 2005 and is available as a NetBeans plug-in module that can be found on the NetBeans Update Center.

Getting the JBuilder Project Importer

The JBuilder Project Importer is an optional module for NetBeans IDE 5.0, so it is not in the standard download.

To download and install the JBuilder Project Importer module, launch NetBeans IDE and choose Tools | Update Center from the main menu. In the Update Center, navigate to and select the JBuilder Project Importer module, click the arrow button to add the module to the list of modules to download, and then complete the wizard.

Once you have downloaded and installed the JBuilder Project Importer module, you are ready to begin migrating a JBuilder project to NetBeans IDE.



The JBuilder 2005 Project Importer works best on general Java projects. For web application projects, Java EE projects, and Java ME projects, you may find importing JBuilder projects easier using the appropriate NetBeans new project template for creating projects with existing sources.

Importing a JBuilder 2005 Project

To initiate an import of a JBuilder project using the NetBeans JBuilder Project Importer, choose File | Import Project | JBuilder 2005 Java Project from the main menu of NetBeans IDE. In the Import JBuilder 2005 Project wizard, enter the location of the JBuilder project file, a `.jpx` file, as the project to import, and a destination folder for the NetBeans project metadata. Then click Finish.

When a JBuilder project is imported, NetBeans IDE leaves your JBuilder project sources in their original location so you can continue to work with your project in JBuilder. However, if you change any project settings in JBuilder, these changes are not updated in the NetBeans project.



If you change project settings in a JBuilder project that you have previously imported into NetBeans using the JBuilder Project Importer, you can import the JBuilder project again with a new NetBeans project name.

If NetBeans IDE cannot find references to resources in the JBuilder project selected to be imported, the JBuilder Project Importer wizard asks you to specify the JBuilder installation directory so it may be searched for the missing resources. Alternatively, you have the option to skip this search and complete the project import. However, if you choose to skip the searching of missing resources, the imported project may not work properly after importing.

After clicking Finish, a NetBeans project will be created for the selected JBuilder project. In addition, NetBeans projects will be

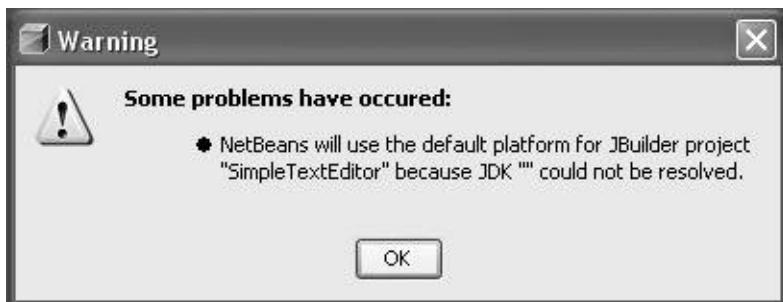
created for dependent projects if they exist.

Project Import Warnings

If, during the importing of a JBuilder project, NetBeans IDE identifies potential issues, the IDE will attempt to resolve many of these issues automatically and inform you of the action it has taken.

For example, if an imported project is using a JDK that cannot be found by NetBeans IDE, the IDE will automatically choose to use the JDK used by NetBeans IDE and inform you with a warning dialog similar to the one shown in [Figure B-1](#).

Figure B-1. Notification of an issue NetBeans JBuilder Project has automatically corrected

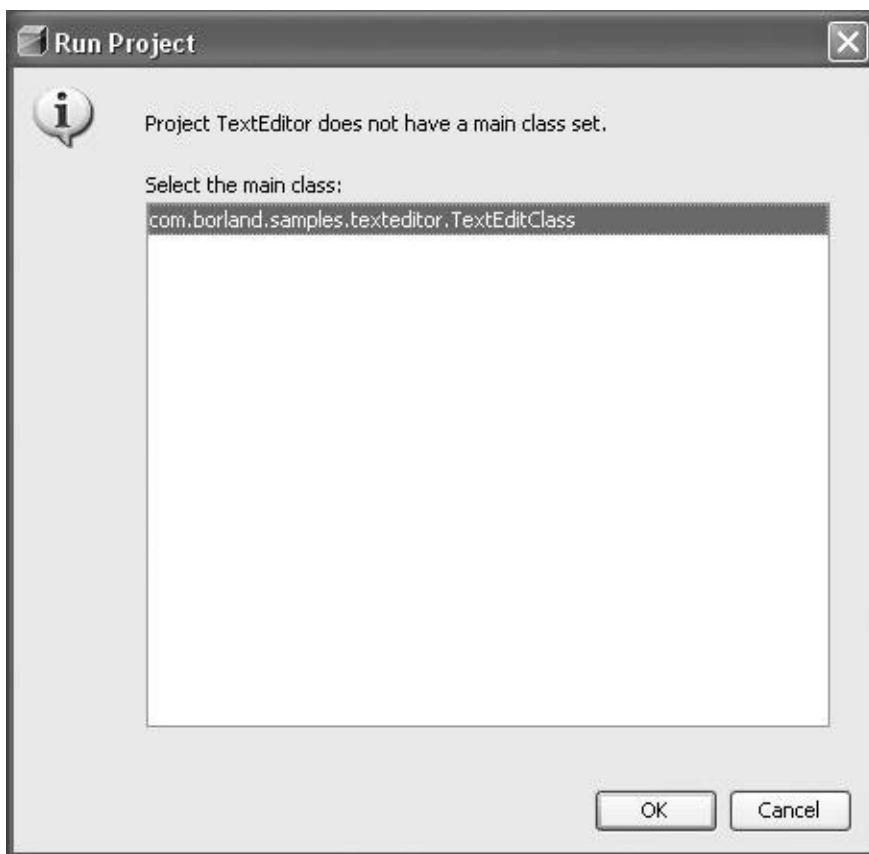


You can change the default platform for an imported JBuilder project in the Project Properties dialog box. Right-click the project in the Projects window and select Properties. Then, in the Project Properties dialog, select the Libraries category and select a Java Platform from the drop-down list or click Manage Platforms to add an additional Java platform.

Running the Imported Project

To run an imported JBuilder project, right-click on the imported project in the Projects window and choose Run Project. If you have not set a main class for the project, you are prompted to do so in a dialog box shown in [Figure B-2](#).

Figure B-2. Selecting project main class



The imported project will now build and run in NetBeans IDE. In addition, you can edit, compile, and debug the imported project.

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

Index

[**SYMBOL**] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

! (not) operator, preprocessor

&& operator

+ (plus sign), splitting strings between lines

. (period), to chain code completion

^ operator

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

abbreviations [See [code templates.](#)]

abilities, configuration

AbsoluteLayout layout manager

abstract elements, WSDL documents

accelerators [See [keyboard shortcuts.](#)]

accounts for collaboration

[logging into server](#)

[managing](#)

[<action> element of project.xml file](#)

Action Source property

[action support \(Struts framework\)](#) 2nd

ActionForm beans

[Actions API](#) 2nd

[active configuration for mobility projects](#)

Add Action wizard

[Add ActionForm Bean Property wizard](#)

[Add ActionForm Bean wizard](#)

[Add Exception wizard](#)

[Add Forward wizard](#)

[Add Forward/Include Action wizard](#)

[Add Managed Bean wizard](#)

[Add Navigation Case wizard](#)

Add Navigation Rule wizard

Add Project option

address attribute, debug target

address parameter (debugger)

addressproperty attribute, debug target

administration, Sun Java System Application Server

aligning form components

allocated objects, tracking 2nd

alphabetically sort button (HTTP Monitor)

analyzing HTTP data

analyzing memory usage

analyzing performance

of code fragments

anonymous inner classes, converting to named 2nd

Ant build scripts

contents of

customizing

debugging

debugging Ant scripts

as file templates

from outside IDE [See [free-form projects](#).]

integrating existing scripts [See [free-form projects](#).]

in mobility projects

profiling and

Sun Java System Application Server and

targets and subtargets

Ant expressions

Ant properties for build outputs

Ant targets 2nd

mapping to IDE commands

Apply Code Changes feature

Compile File command

Debug File command

Debug Project command

Profile Main Project command

Run File command

menu items and shortcuts for 2nd

profiling and

running specific

Apache Jakarta structure

APIs, NetBeans

appearance of form components

Applet Viewer

applet.policy file

applets

applications [See [projects](#).]

Apply Code Changes feature 2nd

applying patches

arity parameter (<action> element)

ascending sort button (HTTP Monitor)

assembling enterprise applications

Assigned Commands (Screen Designer)

associating abilities with configurations

associating message handlers with web services

Attach Profiler feature

Attach Wizard

attaching debugger to applications

authentication

authorizing users for application server instances

auto-incrementation of MIDlet-Version attribute

autocomplete [See [code snippets](#).]

automatic deployment of mobility applications

automatic insertion of closing characters

autoupdate descriptors

average age of live objects

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[Backus-Naur Form \(BNF\)](#)

[badges, CVS](#)

[base template](#)

[basic telemetry profiling](#)

[Bean Patterns node](#)

[bean-managed persistence](#) [See [BMP entity beans.](#)]

[BeanInfo Editor](#)

[behavior of form components](#)

[binding element, WSDL documents](#)

[blank files, building from](#)

[blocks of code](#)

[folding](#)

[indenting](#)

[BMP entity beans 2nd](#)

[BNF grammar](#)

[bookmarking lines of code](#)

[Boolean operators, preprocessor](#)

[Booleans, preprocessor syntax for](#)

[BorderLayout layout manager](#)

[bottom-up approach to creating entity beans](#)

[BoxLayout layout manager](#)

[branches \(versioning\)](#)

checking out
breakpoints 2nd 3rd [See also [debugging applications](#).]
console text for
continuing to next
managing
web applications
Breakpoints window
browsers [See [web browser, default, setting](#).]
build folder 2nd 3rd
 CVS commands and
 mobility projects 2nd
build output [See [Output window, saving content of](#).]
build scripts
 contents of
 customizing
 debugging
 as file templates
from outside IDE [See [free-form projects](#).]
 profiling and
 targets and subtargets
build-impl.xml script 2nd
 merge conflicts
 mobility projects
build.classes.dir property
build.dir property
build.generated.dir property
build.properties file 2nd
build.text.classes.dir property
build.text.results.dir property

build.web.dir property

build.xml script 2nd 3rd 4th

mobility projects

building GUIs [See [GUIs \(graphical user interfaces\)](#).]

building projects 2nd

free-form projects

mobility projects

options for

outside IDE

plug-in modules

business logic for enterprise beans

business methods (enterprise beans) 2nd

refactoring

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

caching service locator strategy

Call Enterprise Bean command

Call Stack window 2nd 3rd

camel case

CardLayout layout manager

Change Method Parameters command 2nd

change tracking [See [tracking changes \(versioning\)](#).]

characters, searching and replacing

checking out sources (version control)

circular dependencies

class libraries

adding to projects

EJB modules

web applications

creating

Mobility Ant Library

stepping into code for

third-party, with plug-in modules 2nd

classes

breakpoint on calls to

compiled, changing location of
compiling into source directories
displaying `toString()` value
as file templates
finding occurrences of
main class

setting or changing
starting debugger outside
moving members to/from
moving to different packages
navigating members of
profiling
renaming occurrences of
restrictions for stepping into
unnesting
web applications 2nd

Classes window

classname attribute, debug target
classpath 2nd

adding to
mobility projects
setting or changing

classpath attribute, debug target 2nd

CLDC (Connected Limited Device Configuration)

cleaning and building projects

closing characters, automatic insertion of
CMP entity beans 2nd

adding fields and relationships

enterprise beans
code blocks, configuration-specific
code completion feature 2nd 3rd
 disabling
 generating JavaBeans component code
 generating methods automatically
code deletion
code formatting
code fragment performance, analyzing
code sections [See [blocks of code, folding.](#).]
code snippets
 code completion feature 2nd 3rd
 disabling
 generating JavaBeans component code
 generating methods automatically
hints to generate missing code
inserting from code templates 2nd
 adding, changing, and removing
 for JSP files
 syntax of
 word-match feature
code templates 2nd
 adding, changing, and removing
 for JSP files
 syntax of
code, stepping through [See also [debugging applications.](#).]
collaboration tools

creating accounts for
interacting with developers
logging into collaboration server
managing accounts for
collapsing blocks of code
colors, code
committing version changes
 searching for commits
comparison operators, preprocessor
compilation classpath [See [classpath](#).]
compilation errors 2nd
Compile File command, mapping Ant target to
compiled classes, changing location of
compiled folder
compiling (building) projects 2nd
 free-form projects
 mobility projects
 options for
 outside IDE
 plug-in modules
compiling JSP files
compiling web services
completion feature 2nd 3rd
 disabling
 generating JavaBeans component code
 generating methods automatically
compliance with Java EE platform
component palette, mobility projects 2nd
components, form

associating icons with behavior and appearance
custom placing and aligning concrete elements, WSDL documents
Concurrent Versioning System (CVS) [See [versioning projects](#).]
#condition directive
conditional breakpoints
configurations for mobility projects
abilities
dependencies with
handling resources for
writing code for
connection factories
Connection wizard 2nd
console messages for breakpoints
Constraints property (TextBox objects)
constructor calls, breakpoints on
consuming Java enterprise resources
consuming web services
contacts and contact groups (collaboration)
container-managed persistence [See [CMP entity beans](#).]
containers
layout manager for, setting
resizing
context parameter (<action> element)
context path (web applications)
Continue command (debugging) 2nd

control panel, NetBeans Profiler
conversation groups (collaboration)
Convert Anonymous Class to Inner command 2nd
Copy Project command
core classes, excluding from profiles
CPU performance profiling 2nd
create methods (enterprise beans)
Create Tests dialog box
Ctrl-spacebar shortcut [See [code snippets](#).]
current thread
cursor position, executing to
Custom Creation Code property
customizing breakpoint console text
CVS (Concurrent Versioning System) [See [versioning projects](#).]
CVS Checkout wizard
CVS menu
.cvspass file

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

data source, for entity beans

databases

creating entity beans on

Derby database integration

using in enterprise applications

DataLoaders

DataObjects

DATE macro

deaths of threads, breakpoints on

#debug directive

Debug File command, mapping Ant target to

Debug Project command, mapping Ant target to

debug-fix target

debug-selected-file target

debug.properties file

Debugger Console

debugger launch parameters

debugging Ant scripts

debugging applications 2nd

applets

backing up to method calls

breakpoints 2nd 3rd

console text for
continuing to next
managing
web applications

fixing code during sessions 2nd
free-form projects 2nd
mobility projects
monitoring variables and expressions
multiple debugger windows
running applications, attaching to
starting outside main class
starting session
stepping through code
thread execution control
web applications

declarative security
default collaboration account
default configuration for mobility projects
default Web browser
#define directive
deleting code
dependencies, project 2nd

importing Eclipse projects
mobility projects
plug-in modules and
depends attribute, debug target 2nd
deploying projects

developed with Matisse

mobility projects
plug-in modules 2nd
web applications
deployment descriptors
enterprise applications
enterprise beans
web applications
Derby integration
descending sort button (HTTP Monitor)
deselecting text, shortcut for
designing forms [See [GUIs \(graphical user interfaces\)](#).]
detecting memory leaks [See [memory usage monitoring](#).]
developer collaboration tools
creating accounts for
interacting with developers
logging into collaboration server
managing accounts for
developing Web applications [See [web applications](#).]
device fragmentation 2nd
code management
JAR file content management
resource management
using different configurations
Device Screen (Screen Designer)
diff files 2nd

patches
directives, preprocessor
disabling breakpoints
displaying projects
dist folder 2nd 3rd
CVS commands and
mobility projects 2nd
dist.dir property
dist.jar property
dist.javadoc.dir property
dist.war property
distributing projects [See [deploying projects](#).]
-Djava.compiler=NONE parameter
do-clean target
document/literal Web services
documentation [See [Javadoc](#).]
downloading NetBeans IDE
duplicating for mobility projects
code blocks
project settings

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[Eclipse projects, importing](#)
[editing code](#) [See also [Source Editor](#).]
[editing collaboration accounts](#)
[editing files](#)
 [web applications](#)
[editing GUIs](#) [See [GUIs \(graphical user interfaces\)](#).]
[editing HTTP requests](#)
[editor hints](#)
[editors for new file types](#)
[EJB modules](#)
 [implementing web services in](#)
 [importing](#)
[#elif directive](#)
[#elifdef, #ifndef directives](#)
[#else directive](#)
[emulator platform, mobility projects](#)
[enabling breakpoints](#)
[Encapsulate Fields command](#) 2nd
[#enddebug directive](#)
[#endif directive](#)
[enterprise applications](#)
 [assembling](#)

importing
profiling [See also [profiling applications](#).]
enterprise beans (EJB)
 adding
 implementing web services in
 importing
 refactoring
 resource consumption
Enterprise template category
entity beans 2nd
 creating, bottom-up approach
 creating, top-down approach
errors
 compilation errors [See [compilation errors](#).]
 Eclipse project reference problems
 JBuilder projects import
 in JSP files
 event listening and handling
 events for form components
 exception breakpoints
 exclamation point (!) operator
 excluding files in CVS operations
 execution control [See [breakpoints](#); ; [stepping through code](#).]
 expanding and folding code
 expressions
 changing values of, when debugging
 monitoring

preprocessor syntax
watches on
:ext connection method
extends clause
external resources for web applications 2nd
external sources, making available
external SSH clients
Extract Interface command 2nd
Extract Method command 2nd
Extract Superclass command 2nd
ExtractTask task

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[faces-config.xml file](#)

[Faces-Servlet class](#)

[Fast Import command](#)

[Favorites window](#) 2nd

[fields](#)

[breakpoints on](#)

[encapsulating](#)

[moving between classes](#)

[names for](#)

[watches on](#)

[file differences \(versioning\)](#) [See [versioning projects](#), [tracking changes](#).]

[file history](#) [See [versioning projects](#).]

[file structure](#) [See [structure of projects](#).]

[file templates](#) 2nd

[file-specific commands](#)

[files](#)

[adding to projects](#)

[EJB modules](#)

[web applications](#)

comparing [See [diff files](#).]

compiling individually

creating 2nd

editing

filtering from project distributables

navigating

not in project

running

searching among

version control [See [versioning projects](#).]

viewing parts simultaneously

Files window 2nd 3rd

EJB modules

web applications

Filesystems API

Filter template

filtering code for profiling

filtering files from project distributables

filtering resources for mobility projects

filtering sources from step-into debugging

filtering task-list entries

Find Usages command

finder methods (enterprise beans) 2nd

finding code [See [searching and replacing code](#).]

finding NetBeans APIs

Fix Imports command

fixed watches

fixing code while debugging 2nd

Flow Designer

FlowLayout layout manager

folder parameter (<action> element)

folders

adding to classpath

adding to Favorites window

adding to web applications

folding blocks of code

Font property

fonts, code

forc code template

fori code template

:fork connection method

fork attribute, debug target

.form files 2nd

form components

associating icons with

behavior and appearance

custom

placing and aligning

Form Editor

Form Tester

format parameter (<action> element)

formatting code

forms, previewing

Forward Displayable property

fragmentation of devices 2nd

code management

JAR file content management

resource management

using different configurations

free-form projects 2nd

Attach Profiler feature
changing target JDK
compiling
creating new
custom menu items
debugging 2nd
debugging Ant scripts
file-specific commands
mapping targets to IDE commands
profiling
running
fresh builds
FTP, deploying mobility projects by
full-scale Java EE development
assembling enterprise applications
compliance, ensuring
consuming resources
database support and Derby integration
entity beans 2nd
creating, bottom-up approach
creating, top-down approach
importing enterprise applications
refactoring enterprise beans
runtime environment
security

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[garbage collection process](#)

[General template category](#) 2nd

[general VCS profiles](#)

[generated code, customizing](#)

[generating missing code](#)

[Generations value](#)

[genfiles.xml file](#)

[Glassfish Java EE 5 server](#)

[graphical user interfaces](#) [See [GUIs \(graphical user interfaces\)](#).]

[GridLayout layout manager](#)

[GridLayout layout manager](#)

[grouping breakpoints](#)

[GroupLayout layout manager](#)

[GUIs \(graphical user interfaces\)](#)

[component behavior and appearance](#)

[component size and alignment](#)

[custom components](#)

[customizing generated code](#)

[deployment, Matisse and](#)

[event listening and handling](#)

forms as file templates

layout managers

previewing forms

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[handleRequest\(\) method](#)

[handleResponse\(\) method](#)

[headless builds](#) 2nd

[mobility projects](#)

[heap contents, displaying](#)

[Hello World project \(example\)](#)

[hiding code sections](#)

[hiding projects](#)

[hierarchy of projects](#) [See [subprojects](#).]

[histories, versioning](#)

[home methods \(enterprise beans\)](#)

[host attribute, debug target](#)

[HTML files, creating](#)

[HTML launcher for applets](#)

[HTML template](#)

[HTTP Monitor](#)

[analyzing data](#)

[replaying requests](#)

[setting up](#)

[HTTP transactions, monitoring](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[icons](#), associating with form components

[IDE debugger](#) [See [debugging applications](#).]

[ide-file-targets.xml file](#) 2nd

[IDE-generated static stubs](#)

[#if directive](#)

[#ifdef, #ifndef directives](#)

[ignoring files in CVS operations](#)

[Image Dialog](#)

[implementing methods](#)

[implements clause](#)

[importing](#)

[Eclipse projects](#)

[enterprise applications](#)

[file templates](#)

[handling automatically](#)

[JBuilder projects](#)

[projects](#) 2nd

[incorporating resources into enterprise applications](#)

[indenting code](#)

[index.jsp file](#)

[JSF support and](#)

Struts support and
insertion of closing characters
insertion point, moving
Inspector component (mobility projects)
Inspector window
installing

Mobility Pack

NetBeans Developer Collaboration Tools

NetBeans IDE

NetBeans Profiler

nonstandard emulator platforms

instant messages to developers

integers, preprocessor syntax for
interacting with developers

interfaces, creating from existing statements

internal SSH clients

inviting developers to conversation groups

isolating debugging to single thread

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[J2EE Container-generated static stubs](#)

[J2EE version 2nd](#)

[J2ME mobility applications](#) [See [mobility projects.](#).]

[J2ME Wireless Toolkit project importing](#)

[JadTask task](#)

[JAR files](#) [See also [projects, deploying.](#).]

[adding to classpath](#)

[adding to web applications](#)

[auto-incrementation of MIDlet-Version attribute](#)

[changing location of](#)

[Matisse, deployment and](#)

[mobility projects](#)

[NBM files vs.](#)

[packaging applets into](#)

[writing manifests for](#)

[java files](#)

[filtered from project distributables](#)

[generating skeleton targets for](#)

[Java Application template](#)

[Java applications, importing](#)

Java BluePrints Solutions Catalog 2nd
enterprise application structure
Java BluePrints structure
importing enterprise applications
Java classes [See [classes, breakpoint on calls to.](#).]
Java code completion box [See [code completion feature.](#).]
Java code templates [See [code templates.](#).]
Java EE Development
configuring IDE for
ensuring compliance with platform
extending applications with web services [See [web services.](#).]
full-scale applications
assembling enterprise applications
consuming resources
database support and Derby integration
entity beans 2nd
importing enterprise applications
refactoring enterprise beans
runtime environment
security
Java BluePrints Solutions Catalog 2nd 3rd
server support
Java enterprise applications [See [EJB modules](#); ;
[web applications.](#).]
Java GUIs [See [GUIs \(graphical user interfaces\).](#).]
Java Library template 2nd

Java packages 2nd

compiling individually

moving classes to

plug-in modules as

Java sources

checking out from repositories

compiling classes into directories with

displaying

external

folder for (web applications)

JDK [See [JDK \(Java Development Kit\)](#).]

restrictions for stepping into

structure of

Java virtual machine arguments

java-home parameter (JVM)

JavaBeans components, generating code for

JavaBeans objects, as file templates

javac task, changing target JDK for

Javadoc

disabling popups

displaying while editing

external

jumping to

JavaHelp Integration API

JavaServer Faces technology 2nd

JBuilder projects, importing

JDBC drivers

JDK (Java Development Kit)

documentation [See [Javadoc](#).]

stepping into code for target, changing 2nd version of
JMS messages, sending JNLP distributions
joining conversations with developers
JSF framework, web applications on JSF support
JSP code templates 2nd
JSP files
 creating
 error checking
 passing request parameters to servlet from, viewing
JSR-101 JAX-RPC
JSR-109 Enterprise web services 2nd
JSR-172 compliance
jump list shortcuts
jumping [See [navigating, among code sections.](#).]
JUnit templates
JVM (Java Virtual Machine)
 arguments
 attaching NetBeans Profiler to parameters of server instances
jvm-options parameter (JVM)
jvmarg parameter, debug target
JXTA peer-to-peer module

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[keyboard shortcuts](#)

[for Ant targets](#)

[changing 2nd](#)

[code completion feature 2nd 3rd](#)

[disabling](#)

[generating JavaBeans component code](#)

[generating methods automatically](#)

[hints to generate missing code](#)

[inserting from code templates 2nd](#)

[adding, changing, and removing](#)

[for JSP files](#)

[syntax of](#)

[for macros \[See \[macros\]\(#\).\]](#)

[navigating between Projects and Files windows](#)

[for text selection](#)

[word-match feature](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[layer.xml file](#)

[layout managers](#)

[Layout property \(Form items\)](#)

[Lazy Initialized property](#)

[leaks, memory](#)

[lib subfolder \(in dist folder\)](#)

[libraries](#)

[adding to projects](#)

[EJB modules](#)

[web applications](#)

[creating](#)

[Mobility Ant Library](#)

[stepping into code for](#)

[third-party, with plug-in modules](#) 2nd

[libraries folder \(web applications\)](#) 2nd

[Library Manager](#) 2nd

[Library Wrapper Module projects](#) 2nd

[licenses for plug-in modules](#)

[life-cycle methods \(enterprise beans\)](#) 2nd

[lightbulb icon](#)

[line breakpoints](#)

line-by-line execution
lines of code
 bookmarking
 completing [See [code completion feature](#).]
 numbers for
 live profiling results, viewing 2nd
Loaders API 2nd
:local connection method 2nd
local CVS repositories
Local Variables window 2nd 3rd 4th
local variables, breakpoints on 2nd
localizing mobility applications
locations for mobility project resources
logging breakpoints
logging into collaboration server
logical structure of projects [See [structure of projects](#).]
<login-config> element
low overhead profiling 2nd

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[macros](#)

[main class](#)

[setting or changing](#)

[starting debugger outside](#)

[main project](#) 2nd

[debugging](#) [See also [debugging applications](#).]

[managed beans](#)

[manifests for JAR files, writing](#)

[mapping bean schema to persistence model](#)

[mapping keyboard shortcuts](#) 2nd

[mapping targets to IDE commands](#)

[Apply Code Changes feature](#)

[Compile File command](#)

[Debug File command](#)

[Debug Project command](#)

[Profile Main Project command](#)

[Run File command](#)

[matching words in files](#)

[Matisse project](#) 2nd

[maximizing windows](#)

[#mdebug directive](#)

members of classes, navigating
memory profiling
memory usage monitoring 2nd
 memory leaks 2nd
menu items for Ant targets 2nd
merge conflicts 2nd
merging files [See [updating versioned files](#).]
message destinations
message element, WSDL documents
Message Handler template
message handlers in web services
message-driven beans
message.properties files
metadata, project 2nd
method breakpoints
methods
 backing up to calls
 breakpoint on calls to
 consolidating from superclasses
 creating from existing statements
 executing without stepping into
 finding occurrences of
 generating to implement and override
 moving between classes
 renaming occurrences of
 signature for, changing
MIDlet class 2nd
MIDP development [See [mobility projects](#).]
MIDP Visual Designer
 Flow Designer

Screen Designer

MIPD Canvas

missing code, generating

Mobile Application template

Mobile Class Library template

Mobile template category

Mobility Ant Library

Mobility Pack, downloading and installing

mobility projects

active configuration

classpath configuration

configuration abilities

configuring for different devices

debugging

dependencies of

deploying automatically

device fragmentation 2nd

code management

JAR file content management

resource management

using different configurations

headless builds

importing

localizing applications

MIDlet-version incrementation

physical structure of 2nd

preprocessor use

reusing settings and configurations

templates for 2nd
visual design for
Flow Designer
Screen Designer
wireless connection tools
modifying collaboration accounts
Module Suite projects 2nd
Module template
monitoring applications [See also [profiling applications](#).]
monitoring HTTP transactions
monitoring variables and expressions
Move Class command 2nd 3rd
Move Inner to Outer Level command 2nd
moving insertion point
moving windows

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

__NAME__ macro

navigating

among code sections

to file's node

within files

between open files

projects

Navigator window 2nd

.nbattrs files

nbbrowse element, debug target

nbjpdaconnect task, debug property

nbjpdastart task, debug target

NBM files

nbproject folder 2nd 3rd

NetBeans APIs

NetBeans Developer Collaboration Tools

creating accounts for

interacting with developers

logging into collaboration server

managing accounts for

NetBeans Eclipse Project Importer

NetBeans Mobility Pack [See [mobility projects](#).]

NetBeans plug-in modules 2nd

adding licenses to

additional information on

building and testing

packaging and distributing

registering

setting up

using NetBeans APIs

NetBeans Profiler [See [profiling applications](#).]

netbeans.home property, debug target 2nd

New Action wizard 2nd

New File Type wizard 2nd

New File wizard 2nd

New Module Project wizard

New Project wizard 2nd 3rd 4th

EJB modules

New Window Component wizard

newo code template

Nodes API

non-caching service locator strategy

nonstandard emulator platforms

not operator, preprocessor

notes, writing to oneself

NullLayout layout manager

numbers for code lines

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[obfuscated folder](#)

[ObfuscateTask task](#)

[obfuscation](#)

[object allocation, analyzing 2nd](#)

[objects assigned to variables, monitoring](#)

[onMessage methods \(enterprise beans\)](#)

[operations, adding to web services](#)

[|| operator](#)

[operators, preprocessor](#)

[optimizing applications \[See \[debugging applications.\]\(#\)\]](#)

[Output window, saving content of \[See also \[compilation errors.\]\(#\)\]](#)

[outputs, mapping free-form project sources to overriding methods](#)

[overriding targets in build scripts](#)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

PACKAGE macros

packages 2nd

compiling individually

moving classes to

plug-in modules as

packaging applets

Palette Manager

Palette window

passing request parameters to web applications

patches (versioning)

pattern parameter (<action> element)

Pause command (debugging)

performance [See [profiling applications.](#).]

period (.), to chain code completion

permissions for applets

persistent entity beans

physical structure of IDE projects 2nd

browsing

dependencies 2nd

Eclipse projects

mobility projects

plug-in modules and
EJB modules
mobility projects 2nd
web applications
placing form components
platform versioning, preprocessor coding and
platforms, adding
plug-in modules
adding licenses to
additional information on
building and testing
packaging and distributing
registering
setting up
using NetBeans APIs
plus sign (+), splitting strings between lines
policies for applets
portType element, WSDL documents
Post-Creation Code property
Post-Init Code property
Pre-Creation Code property
Pre-Init Code property
preprocessed folder
 preprocessors 2nd
preverification
preverified folder
Preverify Task task
previewing forms
primary key class, entity bean
priorities for tasks

private files (web applications)

private folder

private.properties file

Profile Main Project command, mapping Ant target to profiling applications

attaching Profiler to JVM

downloading and installing NetBeans Profiler

free-form projects

Monitor Application task

Profiler control panel

starting session

program testing [See [testing programs](#).]

project classpath [See [classpath](#).]

Project Folder field

Project Properties dialog box

project.properties file 2nd

merge conflicts

project.xml file

merge conflicts

properties in (free-form projects)

projects

adding project results to

compiling (building) 2nd

free-form projects

mobility projects

options for

outside IDE

plug-in modules

create methods (enterprise beans)

creating new 2nd 3rd
EJB modules
enterprise applications
for Java BluePrints solution
free-form projects
mobility projects [See also [mobility projects](#).]
plug-in modules
with JSF support
with Struts support
debugging [See [debugging applications](#).]
dependencies [See [physical structure of IDE projects, dependencies](#).]
deploying
developed with Matisse
mobility projects
plug-in modules 2nd
web applications
files not in, working with
hiding and displaying
importing 2nd
location and name of
metadata 2nd
mobility projects [See [mobility projects](#).]
navigating
profiling applications
attaching Profiler to JVM
downloading and installing NetBeans Profiler

free-form projects
Monitor Application task
Profiler control panel
starting session
putting into CVS
rich-client applications 2nd
running 2nd
applets
attaching debugger when
free-form projects
Java BluePrints projects
JBuilder projects, imported
memory usage
monitoring
on computers without NetBeans IDE
performance analysis
searching files of
sharing [See also [developer collaboration tools](#).]
structure of 2nd
browsing
EJB modules
mobility projects 2nd
web applications
templates for [See also [templates](#), [EJB modules](#).]
testing [See [testing programs](#).]
versioning [See [versioning projects](#).]
web-based [See [web applications](#).]

Projects window 2nd 3rd 4th 5th

CVS status

EJB modules 2nd

web applications

properties of form components 2nd

Properties window 2nd

property parameter (<action> element)

property sheets (Flow Designer)

proxy settings 2nd 3rd

:pserver connection method

public files (web applications)

Pull Up command 2nd

Push Down command 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N]
[O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Z]

QUOTES macro

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[Radio Buttons property](#)

[real-time profiling results, viewing](#) 2nd

[recording macros](#) [See [macros](#).]

[redeploying web applications](#)

[refactoring](#) 2nd

[enterprise beans](#)

[generating JavaBeans component code](#)

[web applications](#) 2nd

[reference problems, Eclipse projects](#)

[referencing Javadoc documentation](#) [See [Javadoc](#).]

[registering](#)

[plug-in modules](#)

[users for application server instances](#)

[web services](#)

[relational databases](#)

[creating entity beans on](#)

[Derby database integration](#)

[using in enterprise applications](#)

[reload button \(HTTP Monitor\)](#)

[Rename command](#)

renaming code elements [See also [refactoring](#).]

enterprise beans

in web applications

replacing code [See [searching and replacing code](#).]

replaying HTTP requests

repositories [See [versioning projects](#).]

request parameters, passing to web applications

requests, SOAP

resizing

form components

windows

Resolve Missing Server Problem dialog box

resource consumption

databases

enterprise beans

JMS messages, sending

resource management, mobility projects

responses, SOAP

results, profiling

resuming execution through method end

reusing mobility project settings

reversing changes in files

rich-client applications 2nd

roles, mapping resources to

RPC/encoded web services

Run File command, mapping Ant target to

Run to Cursor command 2nd

run-selected-file target

running applications 2nd

applets

attaching debugger when
on computers without NetBeans IDE
free-form projects
Java BluePrints projects
JBuilder projects, imported
memory usage
monitoring
performance analysis
RunTask task
runtime arguments
runtime classpath [See [classpath](#).]
runtime environment, Java EE

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[Safely Delete command](#) [2nd](#)

[sample projects, creating](#)

[Samples template category](#)

[saving keystrokes](#) [See [code snippets.](#).]

[schema \(bean\), mapping to persistence model](#)

[SCP, deploying mobility projects by](#)

[Screen Designer](#)

[scrolling window](#)

[Search History window](#)

[searching and replacing code](#)

[searching for commits](#)

[searching for NetBeans APIs](#)

[sections of code](#) [See [blocks of code, folding.](#).]

[<security-constraint> element](#)

[<security-role> element](#)

[security](#)

[applet permissions](#)

[Java EE platform](#)

[select methods \(enterprise beans\)](#)

[semicolons to finish statements](#)

[Send JMS Message command](#)

[server log](#)

server parameter (debugger)
server proxy settings
server selection (web applications)
server-classpath parameter (JVM)
service element, WSDL documents
Service Locator template
Servlet template
 servlets 2nd
session beans
Sessions window
setting main project 2nd 3rd
setting proxy in IDE
sharing IDE projects
shortcuts [See [keyboard shortcuts](#).]
simple declarative security
sizing
 form components
 windows
skeleton targets for compile files
sliding windows
snapshots, profile 2nd 3rd
snippets of code
 code completion feature 2nd 3rd
 disabling
 generating JavaBeans component code
 generating methods automatically
 hints to generate missing code
 inserting from code templates 2nd
 adding, changing, and removing
 for JSP files

syntax of
word-match feature
SOAP messages 2nd
Sony-Ericsson devices, deploying to
Source Editor 2nd 3rd [See also [editing code](#).]
maximizing space for
navigating from
splitting
unrecognized file types
source level, configuring
sourcepath attribute, debug target 2nd
sources
checking out from repositories
compiling classes into directories with
displaying
external
folder for (web applications)
JDK [See [JDK \(Java Development Kit\)](#).]
restrictions for stepping into
structure of
Sources window
splash screen (mobility projects)
splitting Source Editor
splitting windows
src folder 2nd
SSH, connecting to repositories with
starting conversations with developers
starting debugging session 2nd [See also [debugging applications](#).]

startup screen (mobility projects)
statements, finishing
states, thread
Step commands 2nd 3rd
stepping through code
sticky tags for repositories [See [branches \(versioning\)](#).]
stopping builds
String Editor
strings
 preprocessor syntax for
 splitting between lines
structure of projects 2nd
 browsing
 dependencies 2nd
 Eclipse projects
 mobility projects
 plug-in modules and
 EJB modules
 mobility projects 2nd
 web applications
Struts framework, web applications on
struts-config.xml file
style attribute, WSDL documents
subclasses, moving code to
subpackages
subprojects 2nd
 importing Eclipse projects
 mobility projects

plug-in modules and
subset operator (preprocessor)

substitution codes for breakpoint console text
subtargets in build scripts

Sun Java Studio Mobility project, importing

Sun Java System Application Server

Sun Java System Messaging Server

superclasses

extracting to consolidate common methods

moving code to

Surviving Generations value 2nd

suspend parameter (debugger)

suspending threads

suspension of execution [See [breakpoints](#).]

switching branches (versioning)

syntax formatting

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[table components \(mobility projects\)](#)

[tag completion feature](#)

[Tag File template](#)

[tag files, debugging](#)

[Tag Handler template](#)

[Tag Library Descriptor template](#)

[tagging repositories](#)

[tags and tag libraries](#)

[Target Displayable property](#)

[target JDK](#)

[targets in build scripts](#)

[task list feature](#)

[telemetry, profiling](#)

[Template Manager](#)

[templates](#)

[EJB modules](#)

[file templates](#)

[creating and customizing](#)

[mobility projects 2nd](#)

[plug-in modules](#)

[project](#)

web applications
test folder 2nd
test package folder (web applications)
Test packages Folder field
testing programs
 for Java EE platform compliance
 plug-in modules
 web applications
testing web services
text selection shortcuts
third-party libraries with plug-in modules 2nd
threads 2nd 3rd
 breakpoints on 2nd
 execution control
 monitoring states of
tiles-defs.xml file
 __TIME__ macro
time stamp button (HTTP Monitor)
TLD files
To Do window
TODO lines
Tomcat server [See [web applications](#).]
Tomcat source structure
toolbar
 Flow Designer
 IDE, buttons on
 Screen Design
top-down approach to creating entity beans
toString() method, displaying value of
tracking changes (versioning)

reversing changes

tracking notes to oneself [See [task list feature](#).]

transition sources, mobility projects

translation issues with mobility projects

transport attribute, debug target 2nd

transport parameter (debugger)

troubleshooting applications [See [debugging applications](#).]

types element, WSDL documents

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[UDDI registry](#)

[UEI \(Unified Emulator Interface\)](#) 2nd

[#undefine directive](#)

[undeploying web applications](#)

[undoing changes in files](#)

[unit testing](#) [See also [testing programs.](#)]

[unnesting classes](#)

[Update Center wizard](#) 2nd

[updating versioned files](#) 2nd 3rd

[URL namespace \(web applications\)](#)

[Usages window](#)

[use attribute, WSDL documents](#)

[Use Supertype Where Possible command](#) 2nd

[Use This Criterion for Search checkbox](#)

[__USER__ macro](#)

[users, registering for application server instances](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[validation.xml file](#)

[validator-rules.xml file](#)

[value types](#)

[variables](#)

[breakpoints on 2nd](#)

[changing values of, when debugging](#)

[displaying from previous calls](#)

[monitoring 2nd](#)

[preprocessor syntax](#)

[watches on](#)

[VCS \(version control system\) \[See \[versioning projects.\]\(#\)\]](#)

[verifying compliance with Java EE platform](#)

[version, JDK](#)

[versioning projects 2nd](#)

[adding and removing from repositories](#)

[branches 2nd](#)

[checking out sources from repository](#)

[committing changes 2nd](#)

[histories](#)

ignoring specific files

mobility project configurations

patches

putting projects into CVS

setting up CVS

tracking changes 2nd

updating files

without using CVS

Versioning window

versions of platforms, preprocessor coding and

visual design for mobility projects

Flow Designer

Screen Designer

VM arguments

VM Telemetry Overview window

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[wait screens \(mobility projects\)](#)

[WAR files](#)

[creating](#)

[customizing contents of](#)

[packaging applets into](#)

[Watches window](#) 2nd

[Web Application Listener template](#)

[web applications](#) 2nd 3rd

[adding files and libraries](#)

[applet creation and deployment](#)

[changing default Web browser](#)

[debugging](#)

[deploying](#)

[editing and refactoring files](#)

[implementing web services in](#)

[importing](#)

[on JSF framework](#)

[monitoring HTTP transactions](#)

[physical structure of](#)

[profiling](#) 2nd [See also [profiling applications](#).]

representation of, in IDE
on Struts framework
testing and debugging
web browser
 default, setting
 running applets in
 viewing files in
web folder
web pages
web pages folder
Web Service Client wizard
web services
 consuming
 creating new 2nd
 defined
 implementing in EJB modules
 implementing in web applications
 message handlers in
 proxy settings 2nd
 templates for 2nd
 testing
Web template category
web.xml file
 JSF support and
 simple declarative security
 Struts support and
WebDAV, deploying mobility projects by
webservices.xml file
WebSphere server

welcomeJSF.jsp file
welcomeStruts.jsp file
window management
Window System API
windowing component, adding to IDE
Windows API
Wireless Connection Tool
With Existing Ant Scripts template
With Existing Sources template
word-match feature
working directory branch, viewing
writing notes to oneself [See [task list feature.](#).]
WS-I compliance
wscompile tool
WSDL File template
WSDL files
 creating 2nd
 creating web services from
 publishing

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[-Xdebug parameter](#)

[XML-related templates](#)

[-Xnoagent parameter](#)

[-Xrunjdwp parameter](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)]
[[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Z](#)]

[ZIP distributions](#)