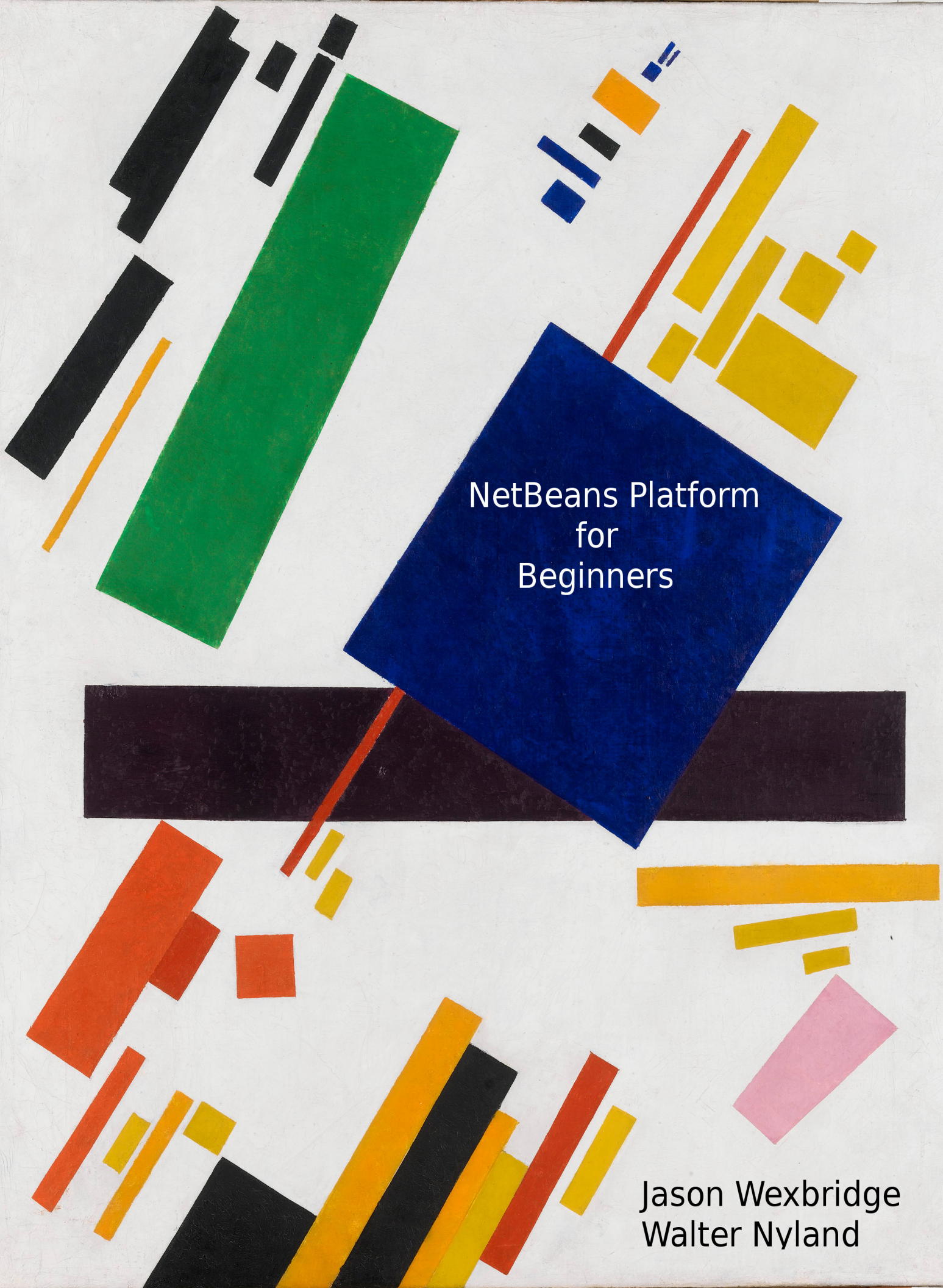


Jason Wexbridge
Walter Nyland

NetBeans Platform for Beginners

Modular Application Development for the Java Desktop

Jason Wexbridge and Walter Nyland

This book is for sale at http://leanpub.com/nbp4beginners

This version was published on 2014-08-31

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Jason Wexbridge and Walter Nyland by spreading the word about this book on Twitter!

The suggested hashtag for this book is #nbp4beginners.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#nbp4beginners

Contents

Introduction **1**

Foreword **2**

Preface **3**

Updates **4**
Update 1: March 26, 2014 4
Update 2: April 30, 2014 4
Update 3: May 30, 2014 5
Update 4: June 26, 2014 5
Update 5: July 29, 2014 6
Update 6: August 30, 2014 6

Guide **8**

Application **10**

Get Started **11**

Part 1: Core **12**

Chapter 1: NetBeans Platform **13**
1.1 Concepts 14
1.1.1 Modularity 14
1.1.2 Abstraction 14
1.1.3 Lifecycle Management 14
1.1.4 Pluggability, Service Infrastructure, and File System 14
1.1.5 Data-Oriented GUI Components 15
1.1.6 Standards 15
1.1.7 Tools, APIs, and Community 15
1.2 Get Started 16
1.2.1 Creation 16
1.2.2 Deployment 19
1.2.3 Commands 23
1.2.4 Properties 23
1.2.5 Features 24

1.2.5.1 Plugin Manager . . . 25
1.2.5.2 Favorites Window . . . 26
1.2.5.3 User Utilities . . . 27
1.2.5.4 Image Editor . . . 28
1.2.5.5 XML Editor . . . 28
1.2.5.6 IDE Defaults . . . 29
1.2.5.7 Apple Application Menu . . . 30
1.2.6 Branding . . . 31
1.2.7 Configuration File . . . 32
1.2.8 Distribution . . . 33

Chapter 2: Module System . . . 34
2.1 Concepts . . . 34
2.1.1 Definition . . . 34
2.1.2 Benefits . . . 35
2.1.2.1 Developers . . . 35
2.1.2.2 Users . . . 35
2.1.2.3 Revenues . . . 35
2.1.3 Characteristics . . . 35
2.1.3.1 Deployment Format . . . 36
2.1.3.2 Uniqueness . . . 36
2.1.3.3 Versioning . . . 36
2.1.3.4 Exposed Interfaces . . . 37
2.1.3.5 Declarative Dependencies . . . 37
2.1.3.6 Activation Type . . . 37
2.1.3.7 Lifecycle . . . 38
2.1.4 Manifest . . . 38
2.1.4.1 General . . . 38
2.1.4.2 Dependencies . . . 39
2.1.4.3 Services . . . 40
2.1.4.4 Visibility . . . 41
2.1.5 Modularity . . . 41
2.2 Get Started . . . 42
2.2.1 Creation . . . 42
2.2.2 Commands . . . 45
2.2.3 Properties . . . 46
2.2.4 Templates . . . 46
2.2.5 Dependencies . . . 49
2.2.6 Versioning . . . 53
2.2.7 Lifecycle . . . 55
2.2.8 Distribution . . . 57
2.2.8.1 Pull . . . 58
2.2.8.2 Push . . . 58

Introduction

NetBeans Platform for Beginners aims to give you a complete and thorough introduction to all the core topics that comprise the NetBeans Platform APIs. Key terms, such as *Module*, *Node*, and *Lookup*, are each dealt with in turn, with a chapter dedicated to each. Each chapter begins with a conceptual overview and rounds off with a set of simple exercises to put the pieces together in the context of real scenarios. Each chapter stands on its own.

It is the hope of the authors that anyone interested in one of the key concepts can jump directly into the related chapter and then read everything that is relevant to that particular topic. Each chapter is divided in two sections, the first a theoretical discussion of the topic at hand, while the second provides hands on exercises in the context of an imaginary car sales application.

Since the focus of the book is specifically on the NetBeans Platform APIs, some interesting and popular technologies and discussions were out of scope and had to be left for another time. In particular, the topics of JavaFX, Maven, OSGi, and JPA, though interesting, are outside the scope of this book, since they are not part of the NetBeans Platform APIs. Similarly, a sequential exploration of the the NetBeans Platform application development cycle (including topics such as testing, internationalization, branding, and the NetBeans installer infrastructure), as well as a step-by-step analysis of the process of porting applications to the NetBeans Platform are not covered in this text. Future books by the authors will endeavor to cover these topics, too.

Throughout this text, existing texts on the NetBeans Platform have been widely consulted and referenced, especially in the conceptual discussions at the start of each chapter. In particular, "NetBeans: The Definitive Guide" (Boudreau, Glick, et al, 2003), "Rich Client Programming" (Boudreau, Tulach, Wielenga, 2005), "NetBeans Platform Developer's Guide" (Petri, 2010), and "The Definitive Guide to NetBeans Platform 7" (Boeck, 2011) have been used in the structuring and content of this book. In gratitude, it is dedicated to the authors of all previous books written on this subject, particularly those mentioned above.

While NetBeans IDE 7.4 was used throughout the writing of this book, later releases of NetBeans IDE should be applicable to these texts, too. The source code for all the examples discussed in this book can be found below.

https://github.com/walternyland/nbp4beginners

Kazimir Malevich's "Suprematist Composition (blue rectangle over the red beam)", 1916, is the illustration shown on the front cover of this book.

And, with that, we wish you happy journeys with the NetBeans Platform!

Foreword

While reading it, I have become fascinated by *NetBeans Platform for Beginners*. When writing my book *Practical API Design: Confessions of a Java Framework Architect* (Apress, 2012), I tried to stress, as much as I could, that a proper API entry point should not be the default Javadoc with its list of classes and methods, but a list of use cases. Newcomers starting to use an API are not interested in wading through the hierarchy of your classes. After all, they have a real problem to solve and they are driven by a need to solve that specific problem. Looking at a flat list of all the classes in an API won't help them with that. However, if your API contains a good list of use cases, chances are high that at least one of them matches the user's needs and that it will be relatively easy to use as a starting point, enabling the user to modify the use cases slightly to solve their specific need.

NetBeans Platform for Beginners has exactly this structure. Each chapter is focused on a particular use case, starting with a description of the problem it helps to solve. Only then does the book show the actual Java code, in the context of a cute car-related sample. The high-level concepts and their usefulness are described first, with their actual usages later. And some of the gory API details, such as the WizardDescriptor class, are left for the last chapter. I love this approach. I believe that *NetBeans Platform for Beginners* should replace our standard NetBeans documentation.

I often say that I have become a good API designer by making all the possible mistakes in the design of NetBeans. One such mistake was not placing enough stress on use case descriptions in our Javadoc. We improved significantly in more recently designed APIs, though the descriptions of the older APIs are really too focused on class hierarchies. Because I like this book so much, I have started putting references to it within each module's Javadoc entry page. In short, if you are developing modular desktop applications in Java, this book is clearly a must read!

Jaroslav Tulach, the founder and initial architect of NetBeans

Preface

By the end of this book, you will have a thorough overview of all the core topics of the NetBeans Platform.

Chapter 1: NetBeans Platform. The NetBeans Platform gives you a well thought-out, widely-used, time-tested, and modular architecture for free. It's an architecture that encourages sustainable development practices.

Chapter 2: Module System. Modules enable development departments to organize their workflow around coarse-grained features, while simultaneously providing flexibility for users, who can install, enable, disable, and uninstall features as needed.

Chapter 3: File System. When applications are installed on a computer, folders and files are created in the computer's file system. Similarly, when modules are installed in a NetBeans Platform application, folders and files are created in its virtual file system. The NetBeans Platform has semantics for converting files to Java objects so that, for example, a file registered in the Menu folder can be converted to a JMenuItem and displayed in the NetBeans Platform menubar.

Chapter 4: Lookup. Lookup is a loose coupling mechanism enabling a component to place a query for a particular interface and get back pointers to all registered instances of it throughout the application, across all modules in the application. Simply put, Lookup is an observable Map, with Class objects as keys and instances of those Class objects as values.

Chapter 5: Action System. The Action System lets you concentrate on your business logic without needing to worry about presentation and enablement, which are taken care of quickly and efficiently via annotations that are turned into entries for inclusion in the System FileSystem during compilation.

Chapter 6: Window System. As applications become more complex, a more complex and flexible user interface is needed. Thanks to annotations that create System FileSystem registrations, the Window System lets you concentrate on business logic instead of spending your time on the plumbing infrastructure needed to display windows to the user.

Chapter 7: Nodes and Explorers. Nodes are a presentation layer between data and any complex data-centric GUI component. Rather than subclassing a different model class for each GUI component you need, you can subclass the Node class and easily display and synchronize the data it visualizes across multiple different GUI components simultaneously.

Chapter 8: Palettes and Widgets. The Visual Library is a generic library for displaying graphic components, which together form graph-oriented visualizations. It is normally used in combination with the Component Palette, which provides a GUI infrastructure for dragging-and-dropping items onto components, which are typically Visual Library components.

Chapter 9: Project System. Letting the user work with individual files is not always optimally efficient, especially when each task involves multiple different yet related files. The Project System lets you work with groups of corresponding folders and files in a coherent manner.

Chapter 10: Miscellaneous. Having covered the core topics, let's now look at some alphabetically sorted supporting features provided by the NetBeans APIs that you will find useful during your journeys with the NetBeans Platform.

Updates

In this section you're shown a list of all the updates made since the initial publication of this book.

The underlying idea is that if you have any comments to share on "NetBeans Platform for Beginners", please let us know at walternyland@yahoo.com or use the feedback page for this book. We promise to respond promptly and we aim to include comments received via monthly updates to this book. Here is the feedback page.

https://leanpub.com/nbp4beginners/feedback

The updates that have been provided so far are integrated into this book, with the listings below showing you where changes have taken place.

Update 1: March 26, 2014

In section 4.1.6, the erroneous override annotations on the *add* and *remove* methods have been removed, with thanks to input from Richard Eigenmann. Moreover, a new sample has been added to GitHub for this section, showing how *CentralLookup* can be used to communicate between modules when context is not relevant. In the example, one module provides a wizard invoked from a new Car menu that, on completion, adds a new Car object to the *CentraLookup* provided by a core module, while another module listens to the *CentralLookup* and updates a node hierarchy whenever a new Car object is found there.

In section 5.1.2.1, some small corrections have been applied to the code, thanks to input from Tushar Joshi. Also, a new sample has been added to GitHub for this section, providing the full code of both complex actions discussed in section 5.1.2.1, in a context that includes a node hierarchy that, depending on whether a Car is selected or a Part, enables/disables the two complex actions discussed in the section.

In section 9.2.2.1, a new explanation and code has been added showing the usage of *LookupProviderSupport.createCompositeLookup*, which enables the *Lookup* of a *Project* to be constructed from a folder in the System FileSystem. In that context, section 9.2.2.1 has been expanded significantly, providing explanations and code for the NetBeans Platform concepts of *privileged templates* and *recommended templates*. The related sample on GitHub has been expanded too, with complete code illustrating the newly introduced topics in the context of the CarProject. Thanks to Dmitry Avtonomov for the idea to include this in this book.

Also, thanks to comments from John, the diagrams at the start of each part have been changed so that the background is not a glaring red but a softer blue.

Update 2: April 30, 2014

Comments received from Guy Daniel, about the phrasing of sentences in the Preface, Guide, and Chapter 1 of the book, have been incorporated. Thanks for your pedantic eye, Guy!

Section 3.1.7 has been expanded to include a discussion of *shadow files*. Thanks to Jean-Marc Borer for this suggestion.

In section 5.1.2.1, a new scenario is discussed, where a *CookieAction* is described that is enabled for multiple different objects. Whether a *Car* or a *Part* is in the *Lookup*, the *CookieAction* automatically enables itself. Thanks to Michal Jucha for the idea to include this topic. A new sample has been added to GitHub for this section.

In section 5.1.2.2.1, a broken link to the list of all Macros has been fixed to http://wiki.netbeans.org/FaqEditorMacros.

Section 5.1.2.3 provides a new advanced scenario for the chapter on Actions. In it, you learn about defining toolbar configurations and about the code you can use to switch between different configurations. Especially in large and complex user interfaces, this can be a useful mechanism for displaying the toolbar buttons relevant to the currently opened set of windows. Michal Jucha suggested the inclusion of this topic. A new sample has been added to GitHub for this section.

In section 7.1.6.1, mentions of *Java Persistence Annotation specification* should have been *Java Persistence API specification*. Thanks to Dustin Marx for identifying this. This is now corrected.

In section 7.1.7, a mention of *org.carsales.api.MainPopupMenu* should have been *org.carsales.api.NodePopupMenu*. Thanks to Dmitry Avtonomov for taking note of this and passing it on.

Section 9.2.2.4 "File Template" has been removed, and section 9.2.2.5 has become 9.2.2.4, because the same topic has been dealt with in more detail in section 9.2.2.1 thanks to last month's update. Thanks to Dmitry Avtonomov for spotting this duplication.

Update 3: May 30, 2014

Throughout the book, wherever a sample is available at https://github.com/walternyland/nbp4beginners that relates to a section, an information icon, as shown here on the left of this paragraph, highlights the availability of the related sample. Use the sample as a guide through the section or as a way to solve problems if you get stuck. The first instance of usage of the information icon is at the start of section 1.2.

Chapter 1 has had a few small additions throughout the chapter, based on feedback received from Peter Vermont, especially about the location of the Branding dialog and the usage of the red Resolve button in the Libraries tab of the Project Properties dialog of NetBeans Platform applications. In section 1.2.7, more details have been added about the *laf* switch, as well as a reference to other values that can be set in the configuration file, for JVM tuning of the application for performance, because of input provided by Miguel.

Section 7.1.5.2 has been expanded with more code and slightly reworded text, thanks to feedback received from Jiri Locker.

Section 9.2.1.2.6 has been added, about branding the labels in the Project System, thanks to a suggestion by kcmoore.

Update 4: June 26, 2014

Section 7.1.6.4.2 has been expanded with code that ensures that a *TopComponent* with *MultiViewElements* is only opened for a *Node* if the *Car* object in the *Lookup* of the *Node* is not found in the *Lookup* of one of the

currently open *TopComponents*. Also, dynamically creating and deleting *MultiViewElements* is introduced, which is possible from NetBeans Platform 8.0 onwards. The related sample on GitHub has been rewritten to illustrate these points.

Section 7.1.6.6.2 has been expanded with code showing how a *Node* can be a drop target. In addition, code has been added to show how you can know on which specific *Node* a drop has taken place, by making use of the *Lookup* of the *Node*. The related sample on GitHub has been rewritten to illustrate these points.

Update 5: July 29, 2014

Foreword added, by NetBeans API architect Jaroslav Tulach.

In chapter 1, two mistyped instances of "cluster" (which had been spelled "cluser") have been corrected. One instance where "OSGI" was used has been corrected to "OSGi". Thanks, Norman Fomferra, for spotting these typos. Thanks to our reader "Turtle", we have fixed a typo in section 2.2.6, in the "Provided Tokens" row of the table, while we improved it further by adding a link to section 4.1.1.5, which provides further information on tokens.

In response to comments from Norman Fomferra, a new preliminary section has been added about the Application that is the focus of the exercises in this book, as well as a new preliminary section on how to Get Started, especially to highlight the *NetBeans API Documentation* plugin, while section 4.1.6, on the Central Lookup, is now associated with a new example, sample 4.2.4, available in the GitHub repository. The sample consists of an application providing three modules, that is, an API, a viewer, and a creator. The creator provides a wizard which, when completed, adds a new Car object to the Central Lookup. The viewer module listens to the Central Lookup for new Cars and then repopulates a Node hierarchy when a new Car is available. While the general usage of Lookup relates to *selection*, that is, *context sensitivity*, this sample shows an implementation of Lookup that provides a global singleton for registering and finding objects throughout the application.

Based on a suggestion by Taco Ditiecher, the sample that illustrates section 9.2 has been expanded to include all the code relevant to section 9.2.1.2.4 on subprojects. To be precise, the sample now includes a new module that defines the Report subproject, together with a new NodeFactory registered in the Lookup of the Car project. Only the SubprojectProvider is part of the public API of the Report subproject module, so that the Car project module can reference it in its Lookup without having access to the implementation of the Report subproject.

Update 6: August 30, 2014

Several small typo fixes and grammatical adjustments throughout the book, thanks to Dustin Marx. Another small correction is in section 1.2.3 – the last command should have been "Properties", not "Branding and Properties", fix has been made thanks to Lorthirk spotting the error and mentioning it on the Feedback page.

Section 1.2.7, about changing the default configuration file, has been rewritten in response to comments received from Norman Fomferra.

Section 4.1.2.1 has been added, about rewriting the global context provider.

A new sample has been added to the GitHub repository to illustrate section 7.1.6, which deals with the *Actions* that relate to *Nodes*. It has been created and contributed by Chris Esposito from Boeing. About the sample, he writes the following.

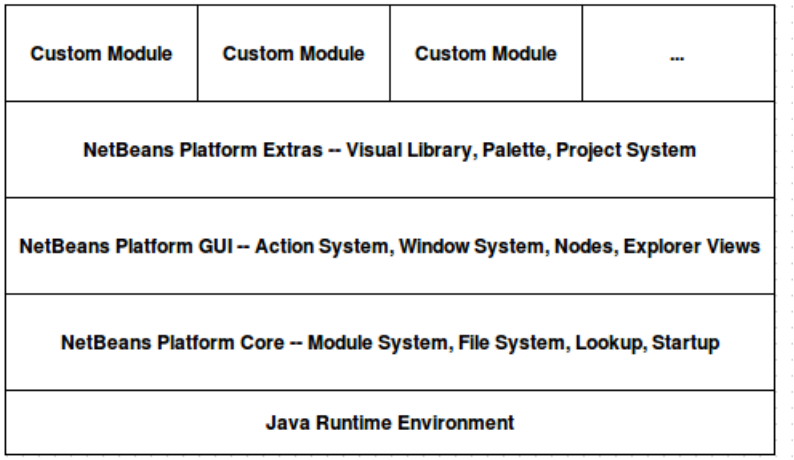
> Arbitrarily nested node hierarchies can be created, and cut / copy / paste / delete / move up & down / drag & drop of user-defined domain objects are available at every level below the root. The acceptability of a drop / paste can also take into account properties of both the dragged node and the dropped node (actually the domain object beans inside both nodes). It's only 6 classes and it's a more complete example than I've been able to find elsewhere online.

The sample is named *CutCopyPasteNB* and is in section 7.1.6 of https://github.com/walternyland/nbp4beginners.

Guide

The key challenge you are going to face with this book is how to map what you learn to the application you're trying to create with or port to the NetBeans Platform. You will frequently find yourself thinking to yourself: "OK, interesting, but how does this relate to my specific context with my particular business requirements?"

To help you, the texts have been structured in a very specific way, following the architecture of the NetBeans Platform from the ground up, though the assumption is you already know Java, so no time at all will be spent on the base level of the NetBeans Platform, i.e., the Java runtime environment.



Throughout, you will be creating NetBeans modules that you will plug into the NetBeans Platform application, providing the highest layer of the architecture shown above.

In "Core", the first part, you learn about the absolute fundamentals of working with the NetBeans Platform. Read through it and do the exercises, while creating your first NetBeans modules, though be aware that the immediate entry points to the kinds of applications you will want to create yourself are introduced in the second part.

In "GUI", the second part, you are given three entry points to porting your own application to the NetBeans Platform. You will probably make use of all three of them. They provide entry points into the NetBeans Platform to move domain-specific elements, such as the GUI components of your own application, into your new NetBeans Platform application, once you have one or two NetBeans modules as an initial architecture. Each entry point is listed below, together with the approach to take in leveraging the entry point in the context of porting your own application to the NetBeans Platform.

Entry Point	Approach
Action	Annotate your existing *Actions* and *ActionListeners* with the annotations you learn about in chapter 5. That will result in your own *Actions* and *ActionListeners* being registered in components such as the menubar and toolbar of the NetBeans Platform application.

Entry Point	Approach
Window	Copy all your own *JPanels* and other GUI containers into the *TopComponents* introduced in chapter 6. Wizards and templates in NetBeans IDE can help you do this very effectively. In this way, you integrate your GUI components with the NetBeans Platform window system.
Node	If you have JavaBeans or files, which you're accessing via database access code and FileIO, begin by working with *Nodes*. In chapter 7, you learn how to render JavaBeans via the NetBeans Platform *BeanNode* and files via the NetBeans Platform *DataNode*. The entirety of chapter 7 is relevant as an entry point when working with Java business objects of any kind, as well as files on disk.

Another way to approach this book is to consider *where the data in your application is coming from.*

Data Source	Approach
Database	Begin by creating a *Node* hierarchy, discussed in chapter 7, illustrated in the example in section 7.2.1. Base your *Node* hierarchy on the *BeanNode* class, display the hierarchy in a *BeanTreeView*, and expose the hierarchy in the constructor of the *TopComponent* that contains the hierarchy and the *BeanTreeView*. Then the Properties window will automatically display the hierarchy and you can begin adding *Actions* to the contextual menu of the *Node*, as well as to the menubar and toolbar of the application.
File	Begin by creating a *ChildFactory* that loads and parses your file, as discussed in chapter 7, illustrated in the example in section 7.2.2. For each item parsed from the file, whatever the item is, create a *DataNode*. Display the *DataNode* in an explorer view in a *TopComponent*. Once this has been done, work on the *Actions* in the contextual menu of the *Node*, as well as the menubar and toolbar of the application.
Multiple Files	Start by looking at the Project system, discussed in chapter 9. In doing so, begin by defining what a project means for your scenario. For example, is there a specific type of file that uniquely identifies a folder as a project? Create a *ProjectFactory* and then a *Project*, then continue from there with the other features of your project implementation. Once you have created the project infrastructure, move on to the *Actions* on the project, as well as the *Actions* in the menubar and toolbar of the application.

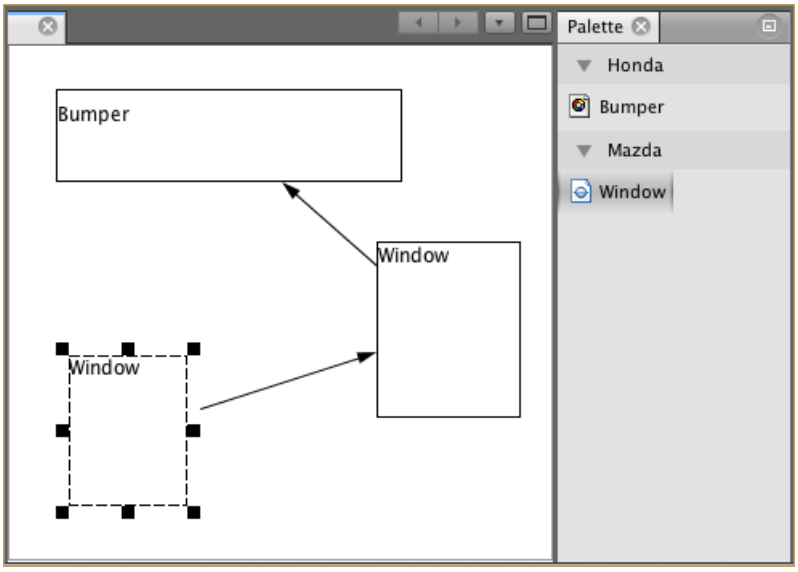
In "Extras", the third part of the book, you are introduced to topics that are not essential parts of the NetBeans Platform, nor are parts that each and every NetBeans Platform application makes use of, though they are, of course, very useful and directly applicable once the key topics have been covered.

Therefore, our advice is to first work through part 1, without immediately applying the concepts discussed to your own scenarios. Then tackle part 2, and after that part 3, using the tables above as your guide to the areas that are most relevant to your needs.

While reading through this book, refer back to this page frequently as a way to keep yourself focused on the specific business needs with which you started using the NetBeans Platform.

Application

Throughout this book, you will be working on an application for the automobile industry. We imagine we need to create an application for visualizing the sales of a car and its parts.



Car Sales Analysis

In a sense, the application could be anything at all, focused on some business use case. Simply to have something to focus on, we chose cars. We could also have chosen airplanes, or hospitals, or patients, or farms, or maps, or any other object that needs to be managed or monitored in some way.

Each chapter focuses on a specific aspect of the NetBeans Platform and then gives you some tasks and exercises to do that are loosely focused on the car paradigm. For example, when we discuss the benefits of registering objects in the System FileSystem, you work on creating an About Box for the application, since this lets you see the usefulness of registering Java objects and retrieving them from the System FileSystem.

Similarly, when we examine the NetBeans Lookup, you create a Car Validator API and then you create various implementations. The validators could be for anything at all, for example, you might want to validate the size of a car part before it is entered into a database, you might want to validate the license plate of a car, you might want to validate a pricing structure, and so on.

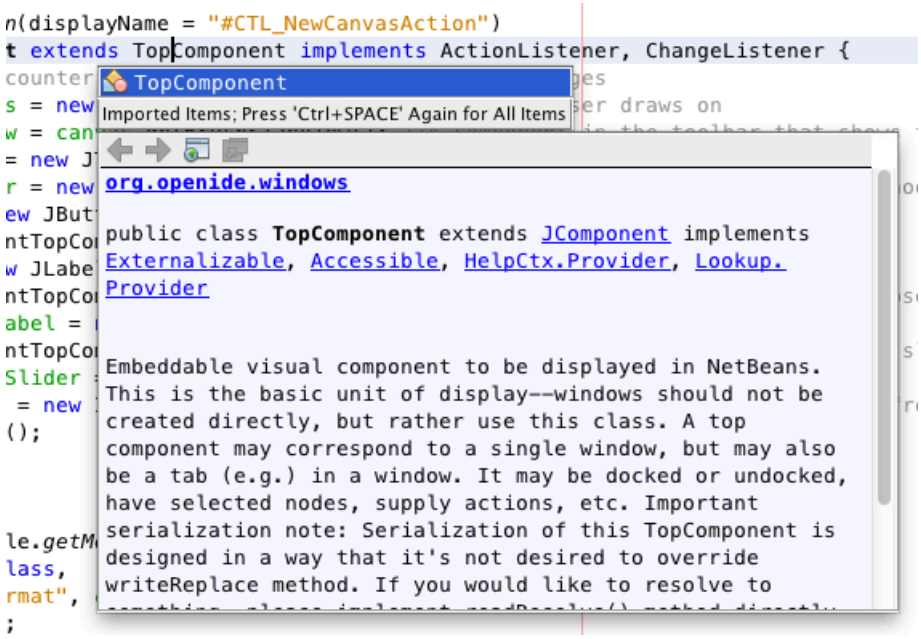
Therefore, the order in which the topics are addressed does not reflect the order in which you are likely to build the features of an application on the NetBeans Platform. For example, the About Box is probably not a very big concern of yours, while the representation of data in a node hierarchy might also not be your major focus. Instead, modularity and the window system are generally the key features that beginners want to leverage in their own applications on the NetBeans Platform.

In short, the application that you'll be working on only exists to the extent that its features are relevant to the main arguments of the book and should be seen more as a playground rather than an actual application that you're striving to create. After all, it's unlikely that you actually want to create a car sales analysis application.

Get Started

Throughout *NetBeans Platform for Beginners*, we assume that you're using NetBeans IDE 7.4 or above as your development environment. It is generally a good idea to always use the latest release of NetBeans IDE and, when new releases become available, we will write our newest samples in the latest version of NetBeans IDE.

Together with installed and running NetBeans IDE, you are highly recommended to install the *NetBeans API Documentation* plugin, after going to Tools | Plugins in the IDE. Once the plugin has been installed, the Java Editor has access to code completion and javadoc for the NetBeans APIs.



NetBeans API Documentation plugin

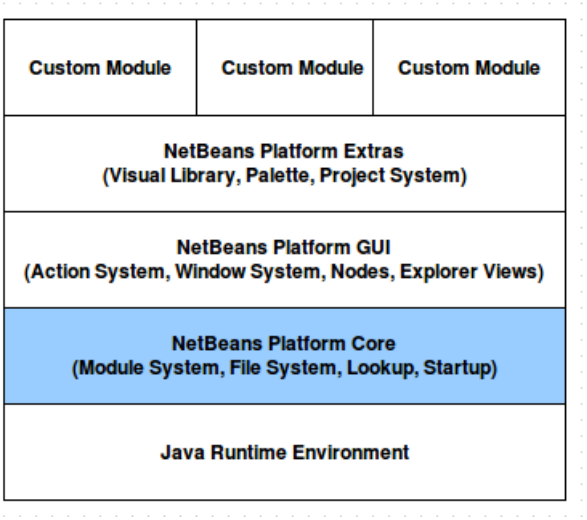
Also note that you can associate NetBeans API source code with the Java Editor. Do this by going to Tools | NetBeans Platforms and then use the Sources tab to specify a ZIP or folder containing the sources, which can be found below.

https://netbeans.org/downloads/zip.html

You will then be able to hold down the Ctrl key and then click from the Java Editor into the source code of the NetBeans API classes you are using.

Part 1: Core

In part 1, you focus on the mandatory core features of the NetBeans Platform. After a high level introduction of what the NetBeans Platform is, you delve into the module system and the file system, together with the loosely-coupled communication solutions provided by the *Lookup* mechanism.



Together, these pieces form the core of the NetBeans Platform. The startup seqence is also part of the core and we don't need to spend any time talking about it because it simply works. Whenever an application on the NetBeans Platform starts up, the internal startup sequence bootstraps the application and presents the GUI to the user. In the chapter on the module system, you learn how you can influence the startup sequence, which should be kept to a minimum so that the startup sequence is as simple, speedy, and optimized as possible.

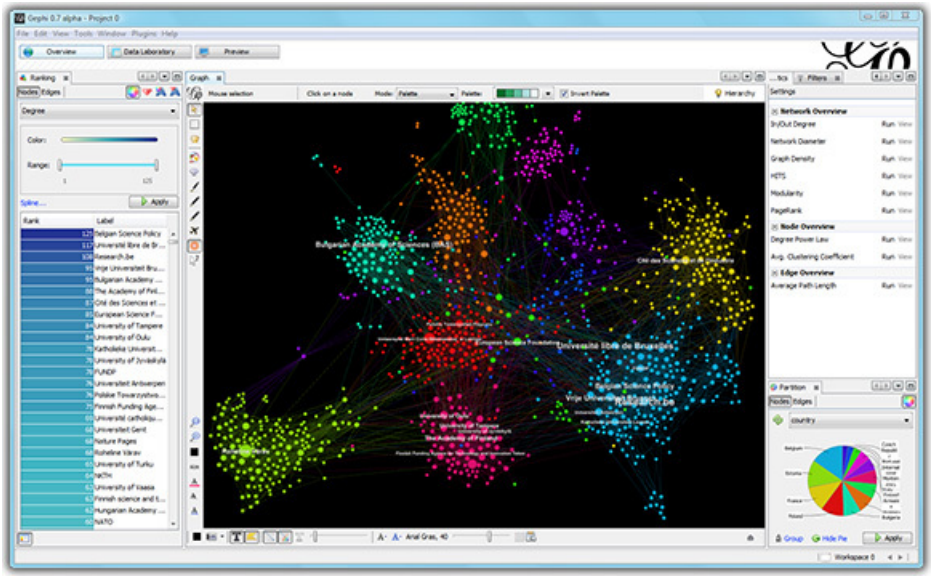
Chapter 1: NetBeans Platform

The NetBeans Platform gives you a well thought-out, widely-used, time-tested, and modular architecture for free. It's an architecture that encourages sustainable development practices.

The NetBeans Platform is a generic application framework for Java desktop applications. The NetBeans Platform provides the infrastuctural plumbing that, without it, every developer has to write themselves, such as solutions for persisting application state; connecting actions to menu items, toolbar items and keyboard shortcuts; window management, and much more. The NetBeans Platform provides all these out of the box so that you don't need to manually code these or other basic features yourself. Instead, you can focus on what your customers care about: domain-specific business logic. For example, developers of software for oil flow analysis can focus on their algorithms, while everything around it, from the architecture of the application to the display of windows to the user, is managed by the NetBeans Platform.

The NetBeans Platform provides a reliable and flexible application architecture that can save you years of development time. There are many strategies and patterns available to help create applications that are robust and extensible, as its developers have many years of experience in creating flexible solutions.

Many applications have been created on the NetBeans Platform. One of them is the award winning visualization tool Gelphi, shown below.



Gephi.org

A thorough list of NetBeans Platform applications is here: https://platform.netbeans.org/screenshots.html

1.1 Concepts

The key concepts around which the NetBeans Platform is designed are as follows.

1.1.1 Modularity

The modular nature of a NetBeans Platform application gives you the power to meet complex requirements with relative ease. You're able to assemble small, simple, and easily tested modules that encapsulate coarsely-grained application features.

Versioning, available per module, helps give you confidence that your modules will work together, while strict control over the public APIs they expose help you create a flexible application that's easier to maintain.

Since your application can use either standard NetBeans Platform modules or OSGi bundles, you're able to integrate third-party modules or develop your own.

1.1.2 Abstraction

Many applications, especially as they increase in size, tend to have multiple features performing similar tasks. The NetBeans Platform solves this problems through very heavy use of abstractions. For example, when you interact with a file, you will be using a FileObject, not an instance of java.io.File. When you deal with menu items and toolbar buttons, under the hood you will be using NetBeans Platform Action classes, instead of directly interacting with menus and toolbars, though you can create custom components to use if you need them. Another abstraction, on top of files and other business objects, is called Nodes.

In general, the NetBeans Platform provides high-level abstractions to handle the common cases found in software development, while allowing the flexibility needed to do something more low-level if the need arises.

1.1.3 Lifecycle Management

Just as application servers, such as GlassFish or WildFly, provide lifecycle services to web applications, the NetBeans Platform aims to provide lifecycle services to Java desktop applications. Application servers understand how to compose web modules, EJB modules, and related artifacts, into a single web application. In a comparable manner, the NetBeans Platform understands how to compose NetBeans modules into a single Java desktop application.

Furthermore, there is no need to write a main method for your application because the NetBeans Platform already contains one. Support is provided for persisting user settings across restarts of the application, such as, by default, the size and positions of the windows in the application.

1.1.4 Pluggability, Service Infrastructure, and File System

End users of the application benefit from pluggable applications because these enable them to install new features, in the middle of release cycles, into their running applications. At runtime, NetBeans modules can be installed, uninstalled, activated, and deactivated.

The NetBeans Platform provides an infrastructure for registering and retrieving service implementations, enabling you to minimize direct dependencies between individual modules and enabling a loosely coupled architecture, via built-in strategies supporting "high cohesion and low coupling".

The NetBeans Platform provides a virtual file system, which is a hierarchical registry for storing user settings, comparable to the Windows Registry on Microsoft Windows systems. It also includes a unified API providing stream-oriented access to flat and hierarchical structures, such as disk-based files on local or remote servers, memory-based files, and even XML documents.

1.1.5 Data-Oriented GUI Components

Most serious applications need more than one window. Coding good interaction between multiple windows is not a trivial task. The NetBeans window system lets you maximize/minimize, dock/undock, and drag-and-drop windows, without you providing any code at all.

JavaFX and Swing are the standard UI toolkits on the Java desktop and can be used throughout the NetBeans Platform. Related benefits include the ability to change the look and feel easily via "Look and Feel" support in Swing and CSS integration in JavaFX, as well as the portability of GUI components across all operating systems and the easy incorporation of many free and commercial third-party Swing and JavaFX components.

With the NetBeans Platform, you're not constrained by one of the typical pain points in Swing: the JTree model is different to the JList model, even though they present the same data. In fact, all data-oriented GUI components in Swing have their own model structure. Switching between GUI components means rewriting the model. The NetBeans Platform Nodes provide a generic model for presenting your data. The NetBeans Platform Explorer Views provide advanced Swing components specifically created for displaying Nodes.

In addition to a window system, the NetBeans Platform provides many other UI-related components, such as a property sheet, a palette, complex Swing components for presenting data, a Plugin Manager, and an Output window.

1.1.6 Standards

One of the most striking aspects of the design and codebase of the NetBeans Platform is its use of *standards*. Wherever a standard for doing something exists, the developers of the NetBeans Platform opt to use it, rather than reinvent the wheel. For example, module manifest files are based on the Java Versioning Specification, Nodes are conceptually based on the JavaBeans BeanContext specification, and so on. Wherever there was an existing standard or a near match, it was used.

What this adherence to standards achieves is extensibility. As other pieces of code that work with the same standards are created, it is much less difficult to get them to interoperate with the NetBeans Platform. It requires greater discipline to adhere to standards than to reinvent the wheel, but doing so gets you maintainability and interoperability, as standards are, by definition, documented, and if something is a standard, others are probably using it as well.

1.1.7 Tools, APIs, and Community

The NetBeans IDE, which is the software development kit (SDK) of the NetBeans Platform, provides many templates and tools, such as the award winning Matisse GUI Builder that enables you to very easily design your application's layout.

The NetBeans Platform exposes a rich set of APIs, which are tried, tested, and continually being improved.

The community is helpful and diverse, while a vast library of blogs, books, tutorials, and training materials are continually being developed and updated in multiple languages by many different people around the world.

1.2 Get Started

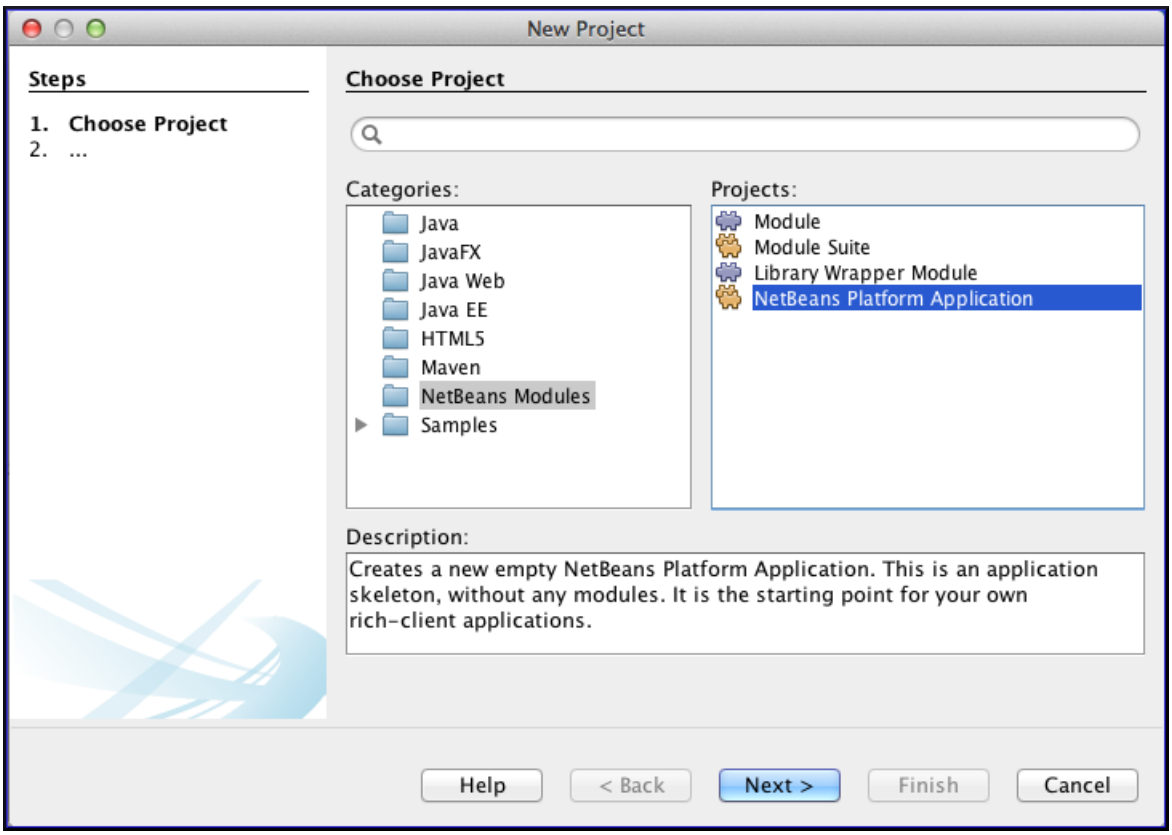
With the theoretical background behind you, let's get your feet wet. What you should notice in the sections that follow is that the NetBeans Platform provides a comprehensive application framework within which any domain-specific application targeting the Java desktop can be developed.

For an example application related to this section, see sample 1.2 at https://github.com/walternyland/nbp4beginners.

1.2.1 Creation

Let's get started creating NetBeans Platform applications!

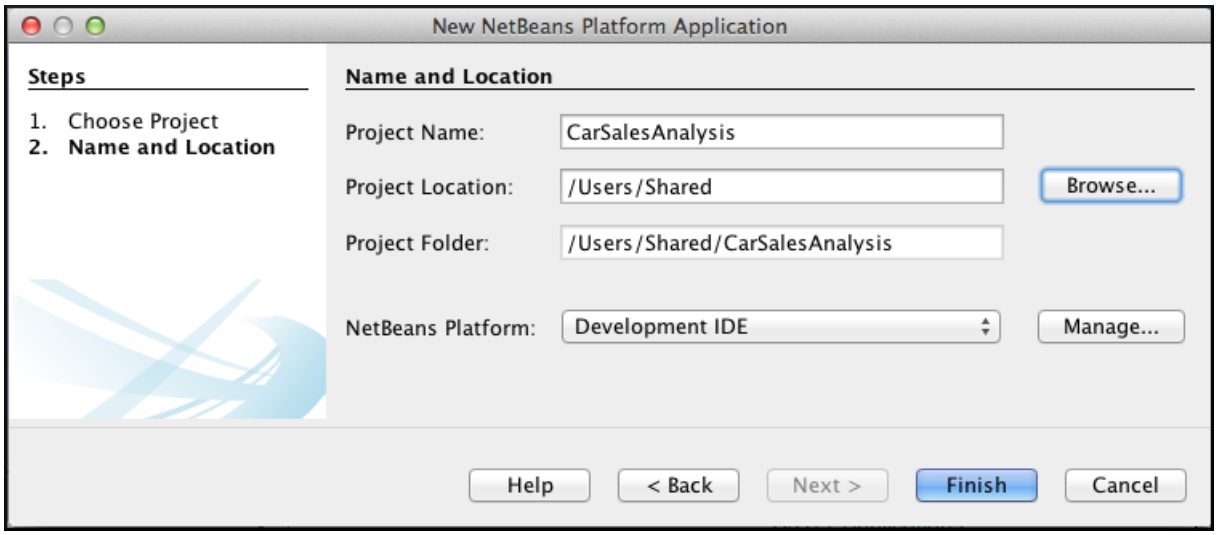
In the **New Project wizard (Ctrl-Shift-N)**, choose **NetBeans Modules | NetBeans Platform Application**.



Step 1 of the New Project wizard

Click Next.

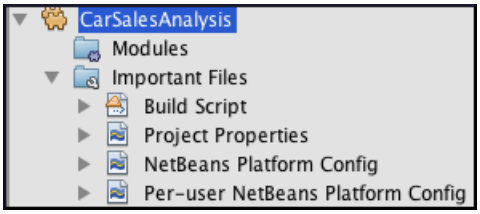
Name the application **CarSalesAnalysis** and specify a location where the application will be stored.



Step 2 of the New Project wizard

Click Finish.

In the Projects window (Ctrl-Shift-1), the starting point for your new application looks as follows.



Logical view of newly created NetBeans Platform application

A simple starting point for a NetBeans Platform application has been created for you. It includes some basic features, such as the Window System and Action System. More features from the NetBeans Platform can easily be included, as explained later in this chapter.

Let's examine the files that are displayed in the Projects window.

Name	Description
Build Script	Provides Ant targets for tasks such as *run*, *build*, and *clean*. When you right-click a NetBeans Platform application in NetBeans IDE and invoke a project command, an underlying Ant target is invoked. The Ant targets can be customized as needed.
Project Properties	Defines NetBeans Platform properties, such as the custom modules you have created that belong to the application, the application title displayed at the top of the application, and the application icon.

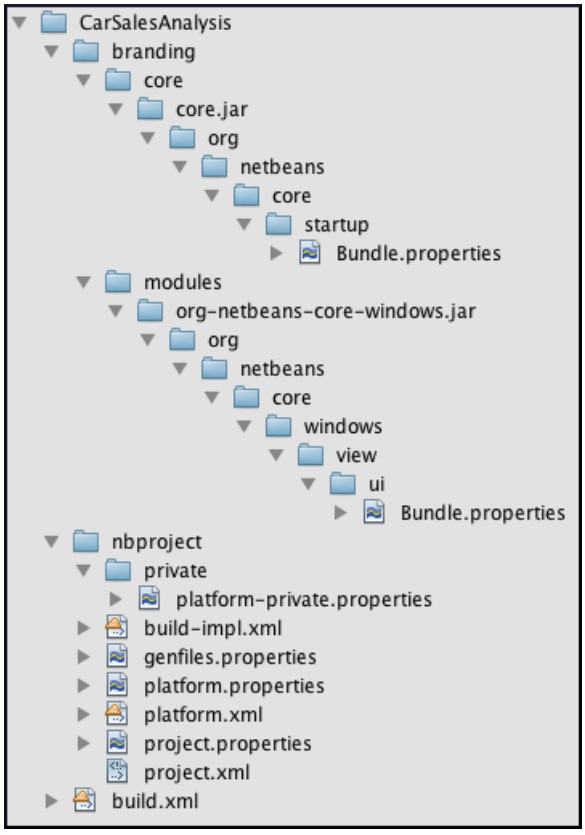
Name	Description
NetBeans Platform Config	Defines the branding token, used for example for the name of the launcher such as the *.exe* file created by the *Package as / ZIP Distribution* menu item, and the groups of NetBeans Platform modules that have been included in the application, as well as the excluded modules from the included groups.
Per-User NetBeans Platform Config	Points to a properties file containing properties used for building the application.

While the Projects window presents the *logical view*, that is, it shows you the list of folders and files with which you are mostly going to be working, the Files window shows you the complete project structure, that is, *everything* that is in the project, whether you will be touching it and working with it or not.

The Files window shows the following folders and files that are not shown in the Projects window.

Name	Description
branding	The *branding* folder contains subfolders representing the JARs that make up the NetBeans Platform. When you need to override an existing text or image in the NetBeans Platform, you place the overridden text in a *Bundle.properties* file, or the overridden image in the folder itself, in the same folder structure as where it is found in the relevant JAR in the NetBeans Platform. When the NetBeans Platform is built, the folders in the *branding* folder are JARred up and organized in such a way that they override their equivalents in the NetBeans Platform. Mostly this content is generated from the Branding dialog, which is displayed when you right-click a project and choose Branding.
build-impl.xml	The Ant build script file that contains all the targets. Do not change this file. Instead, override its targets in your *build.xml* file, which imports *build-impl.xml*.

The Files window with its content, described above, looks as follows.

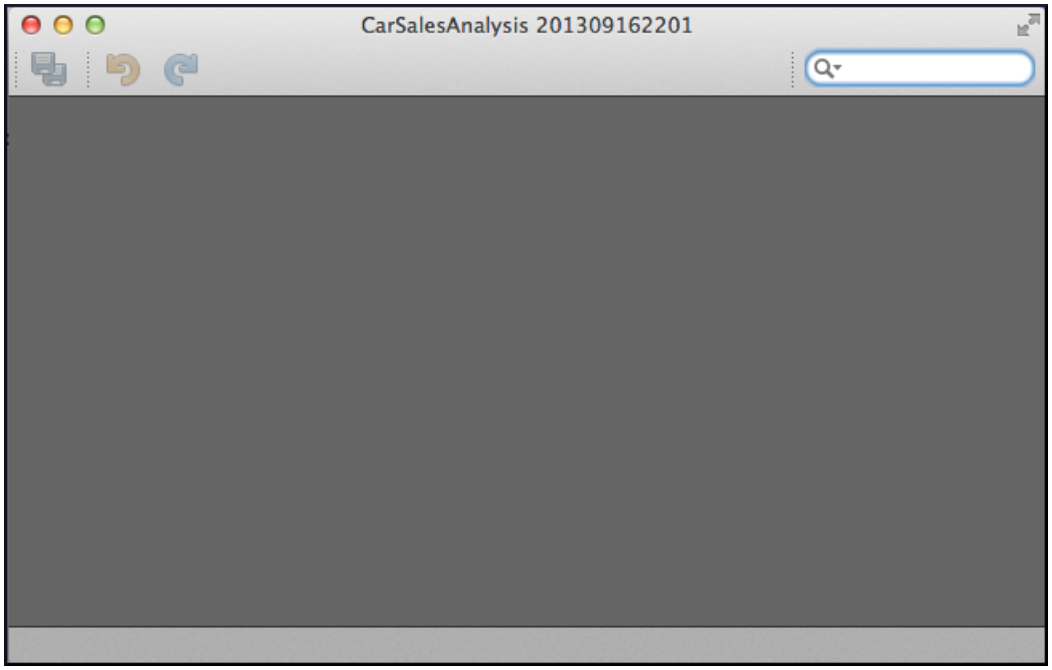


File view of newly created NetBeans Platform application

Now that you understand the structure of the folders and files in your skeleton NetBeans Platform application, let's deploy it.

1.2.2 Deployment

Right-click the application and choose **Run**. The application is deployed and displays as follows.

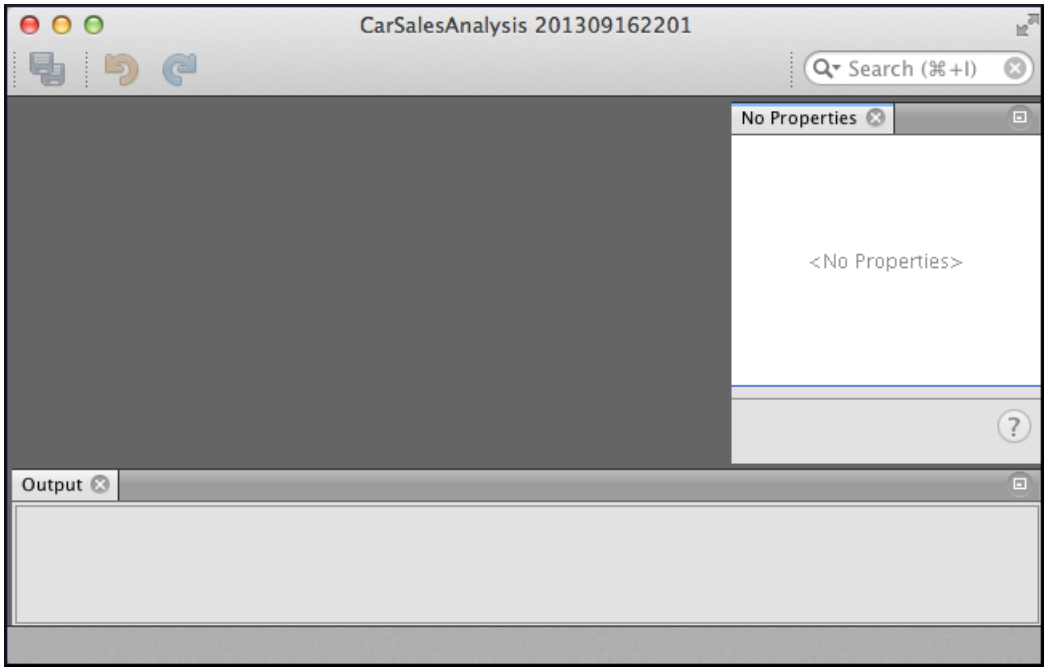


Newly deployed NetBeans Platform application

The menubar is as follows.

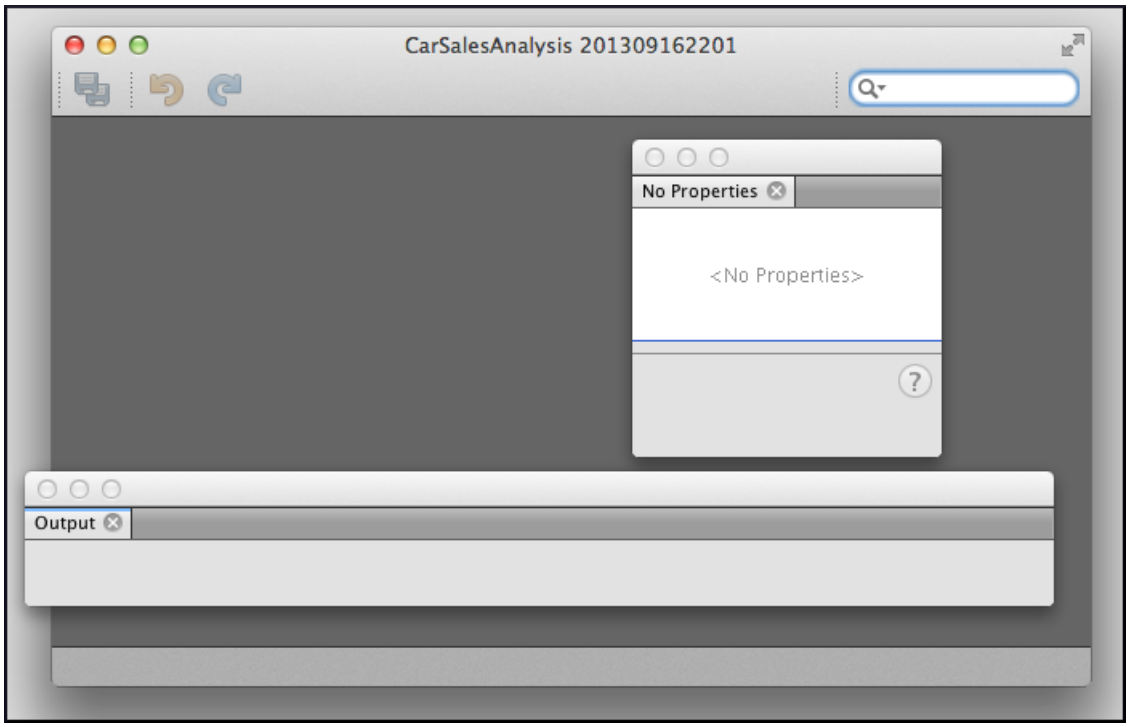
Default menubar of skeleton NetBeans Platform application

Explore the menu items and open some of the windows. The application has a menubar, a toolbar, and the ability to display multiple windows simultaneously. For example, go to the Window menu and find the Output window and Properties window. Then open them. You should see the below.



Output window and Properties window

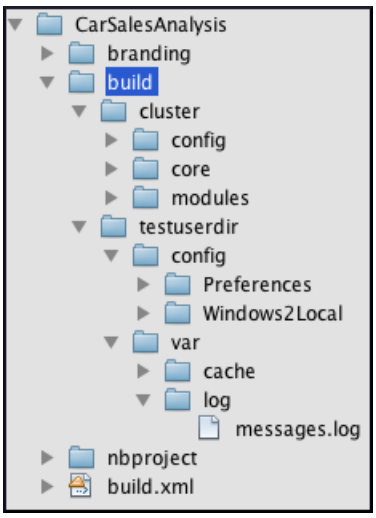
The windows can be undocked from the main window, and even moved onto different monitors, if you right-click a tab and choose Float.



Floating windows

Close the application.

Look again in the Files window. Notice that a *build* folder has been created:



Build folder after closing the application

The *build* folder contains the following folders and files.

Name	Description
cluster	The NBM archive files of all the NetBeans modules created in the application, as well as their startup instructions, such as definitions of their activation types.
testuserdir/config	The *user directory* of the application during development. Here all the customizations you make to the application are stored, such as the preferences that you change and the window layouting that you customize at runtime. These are restored when you redeploy the application during development. To reset the application to its application-defined definitions, remove the *build* folder, or choose *Clean* from the application project commands, which achieves the same thing.
testuserdir/var	The *log file* of the application is *testuserdir/var/log/messages.log*. This is one place to look when things don't work as you expect.

The *build* folder is *generated* when the application is built and *added to* during the time it is running. For example, when you move a window to a different location while the application is running, that location is stored in *testuserdir/config/Windows2Local* when the application shuts down, and restored when the application restarts.

In the same way as you have a *testuserdir* during development, the user of the application will also have a *user directory* where their runtime customizations will be stored. The location of the *user directory* is always separated from the location of the *installation directory*. When the user uninstalls the application, their customizations remain intact in the *user directory* and can be reused when they start up a new installation of the application.

1.2.3 Commands

When you right-click the NetBeans Platform project in the Projects window, you see a popup menu containing NetBeans Platform project commands, the most significant of which for NetBeans Platform projects are as follows.

Project Command	Description
Build, Clean and Build, and Clean	*Build* compiles the application, with all its modules, into the *build* folder, visible in the Files window. *Clean* removes the *build* folder. The instruction to *clean the application* simply means to invoke the *Clean* command.
Package as	Makes the application available to be distributed to your users as a ZIP containing the binary distribution, including launchers, NetBeans modules as binary NBM archive files, a Mac OSX application, or an installer for each operating system set in the Project Properties dialog.
Run, Debug, Profile, and Test	Deploys the application, start the Java Debugger, start the Profiler, or run all the tests in the application.
JNLP and OSGi	Creates Java web-start enabled applications or run the application in an OSGi container. Though both of these are supported, neither are recommended, unless there are strong and specific business requirements for them.
Branding	Opens the Branding dialog, which is discussed below.
Properties	Opens the Project Properties dialog, which is discussed below.

1.2.4 Properties

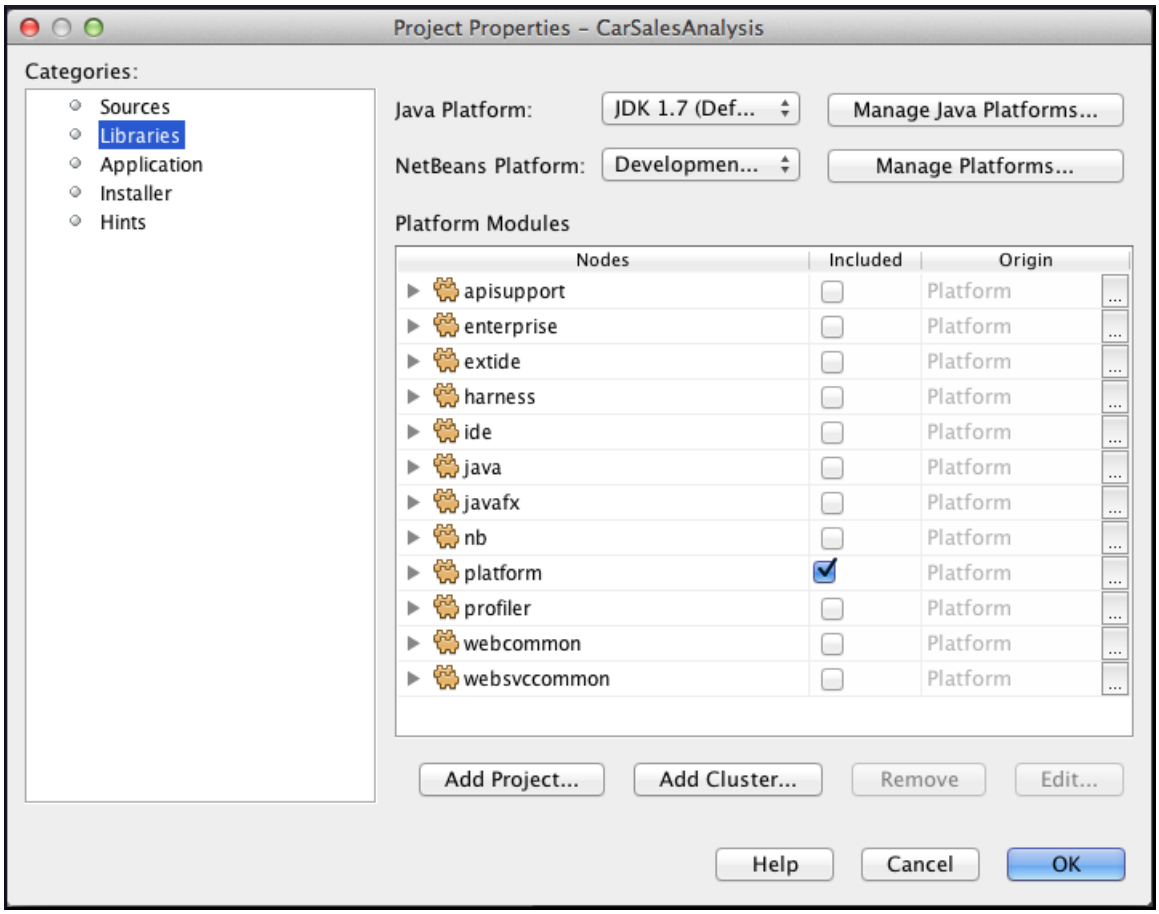
When you right-click the NetBeans Platform project in the Projects window and choose Properties, the *Project Properties* dialog opens. The following tabs are displayed.

Name	Description
Sources	Specify the NetBeans modules that form part of the application and view the project location.
Libraries	Specify the JDK location, NetBeans Platform location, NetBeans Platform modules included in the application, as well as the modules from other groups of NetBeans modules, such as the *ide* cluster, which provides NetBeans modules that are part of NetBeans IDE.
Application	Specify the branding token for the application, which is used as the name of the launchers, e.g., the *.exe* file, that is created when you choose *Package as / ZIP Distribution.*
Installer	Specify the operating systems for which you'd like to have Installers generated. Then click OK and right-click the application and choose *Package as / Installers* to let the installers be generated for you.

Name	Description
Hints	Specify the Java hints to be used throughout all the Java source files in the NetBeans modules constituting the application.

1.2.5 Features

Right-click the project and choose Properties. The Project Properties dialog opens. The list of NetBeans Platform modules currently available to the application is shown in the Libraries tab. Each module, or groups of interdependent modules, provides a feature to the application, such as the Properties window or the Output window. In the dialog below, a check mark includes the NetBeans Platform module in your application.



Adding new features

The list of items in the *Nodes* column above represent the folders that will be part of the application once the user has installed it. In addition to the *bin* folder, where the launchers, such as the *.exe* file are found, there will be a *platform* folder, containing the NetBeans modules you have chosen from the cluster of NetBeans modules that make up the NetBeans Platform. Similarly, there might also be an *ide* folder, if you have selected NetBeans modules from the *ide* cluster, such as the *Image* module, which is discussed later in this chapter.

To add new NetBeans modules, from the *platform* cluster or any other cluster, place a checkmark in the

checkbox next to the module. Some modules depend on other modules and you will automatically be shown which other modules to include together with the modules you selected. To help you, a red Resolve button will become enabled to prompt you to let the IDE automatically add required dependencies for you. It does this by analyzing the selected modules and then identifying their dependencies, so that you do not manually need to do this. Then click OK to exit the Project Properties dialog, clean the application, and run it again to see the new feature available during development.

Note. In NetBeans IDE 8, you may see a message that tells you that *Module JavaFX wrapper in platform requests the token org.openide.modules.jre.JavaFX but there are no known providers*. You can solve this problem by expanding the *platform* node and then unchecking the checkmark next to *JavaFX Wrapper*, assuming that you are not planning to work with JavaFX in your application. As stated in the introduction to this book, JavaFX is outside the scope of this book, a separate book is being written on this subject in the context of the NetBeans Platform.

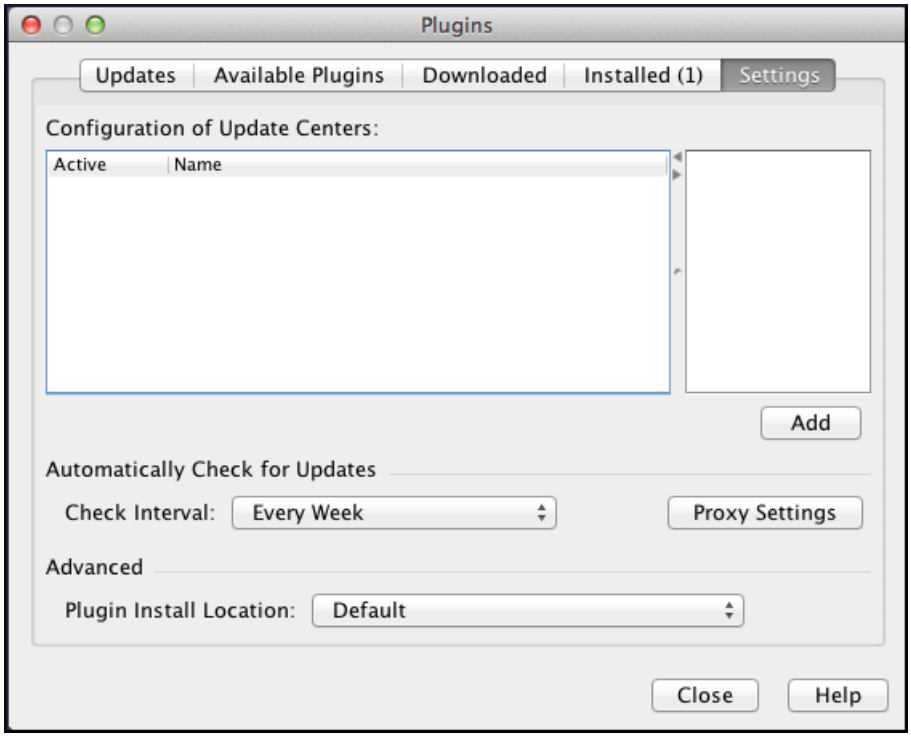
Let's now use the mechanism described above to add a few new features to your application. These features are all very typical features for any NetBeans Platform application to have, but are not included by the template that created the application structure for you.

1.2.5.1 Plugin Manager

The Plugin Manager lets users install plugins from *Update Centers* registered in the application, while they can also install modules downloaded onto their computer.

This tool is provided by two modules in the *platform* cluster: *Auto Update Services* and *Auto Update UI*. Include these two modules, click OK to exit the Project Properties dialog box, clean the application, and run it again.

From the Tools menu, you will be able to open the Plugin Manager, which looks as follows.



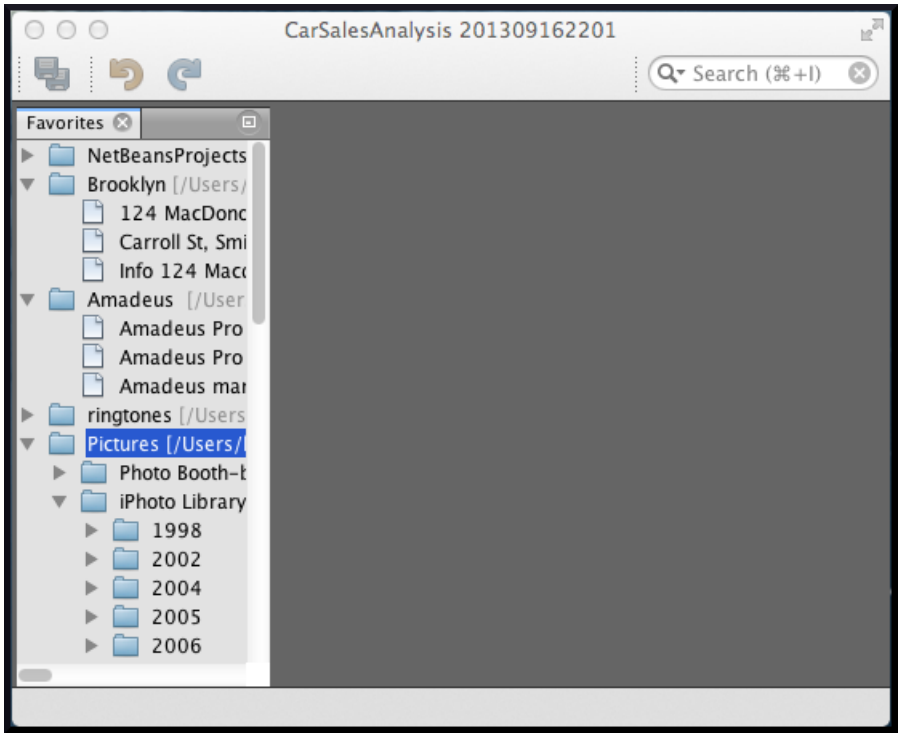
Plugin Manager

1.2.5.2 Favorites Window

The Favorites window is a file browser. Include it in your application if your users will find it helpful to be able to browse their disk and add new folders and files to this window for further exploration.

This tool is provided by one module in the *platform* cluster: *Favorites*. Include this module, click OK to exit the Project Properties dialog box, clean the application, and run it again.

From the Window menu, you will be able to open the Favorites window, which looks as follows.

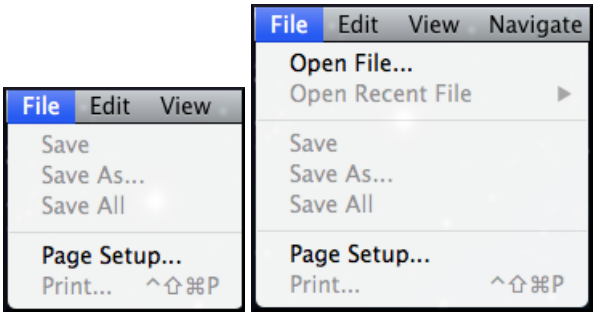


Favorites Window

The labels that you see in the Favorites window, such as "Favorites" in the tab, can be branded. Right-click the application, choose *Branding*, and then look for the labels you want to override in the *Resource Bundles* tab. In each case, you can provide your own, which will override the labels defined in the NetBeans Platform.

1.2.5.3 User Utilities

By default, the File menu does not include an *Open File* menu item, as can be seen in the screenshot on the left. By including the *User Utilities* module from the *ide* cluster, the *Open File* menu item is included, as well as an *Open Recent File* menu item, as shown below.



When *Open File* is selected, browse to a file on disk and, if the application understands the selected file type, it will try to open it into the application. If it does not understand the file type, it will try to open the file into the NetBeans Platform text editor.

1.2.5.4 Image Editor

Image files, with extensions such as *jpg* and *png*, can be opened from the Favorites window, or the Open File menu, if the *Image* module is included. The *Image* module is found in the *ide* cluster and depends on the *Navigator API* module, which you will be prompted to include when you select the *Image* module.

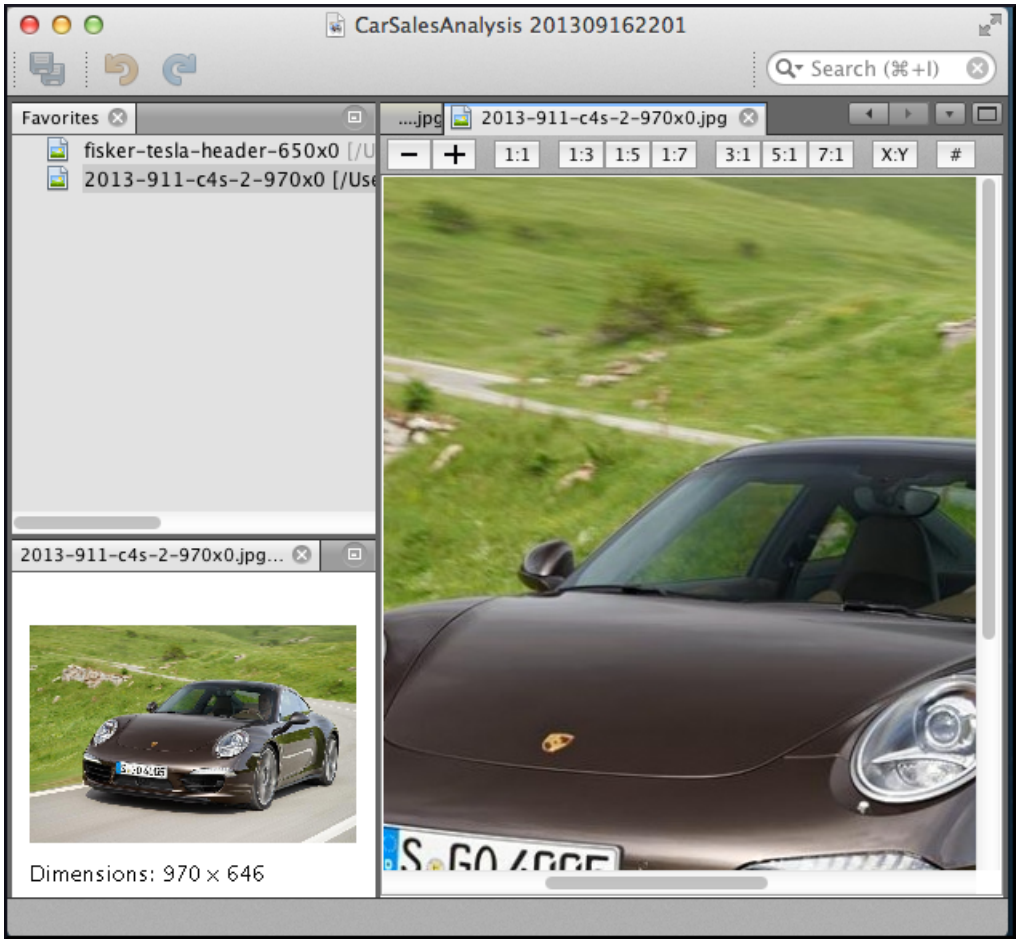


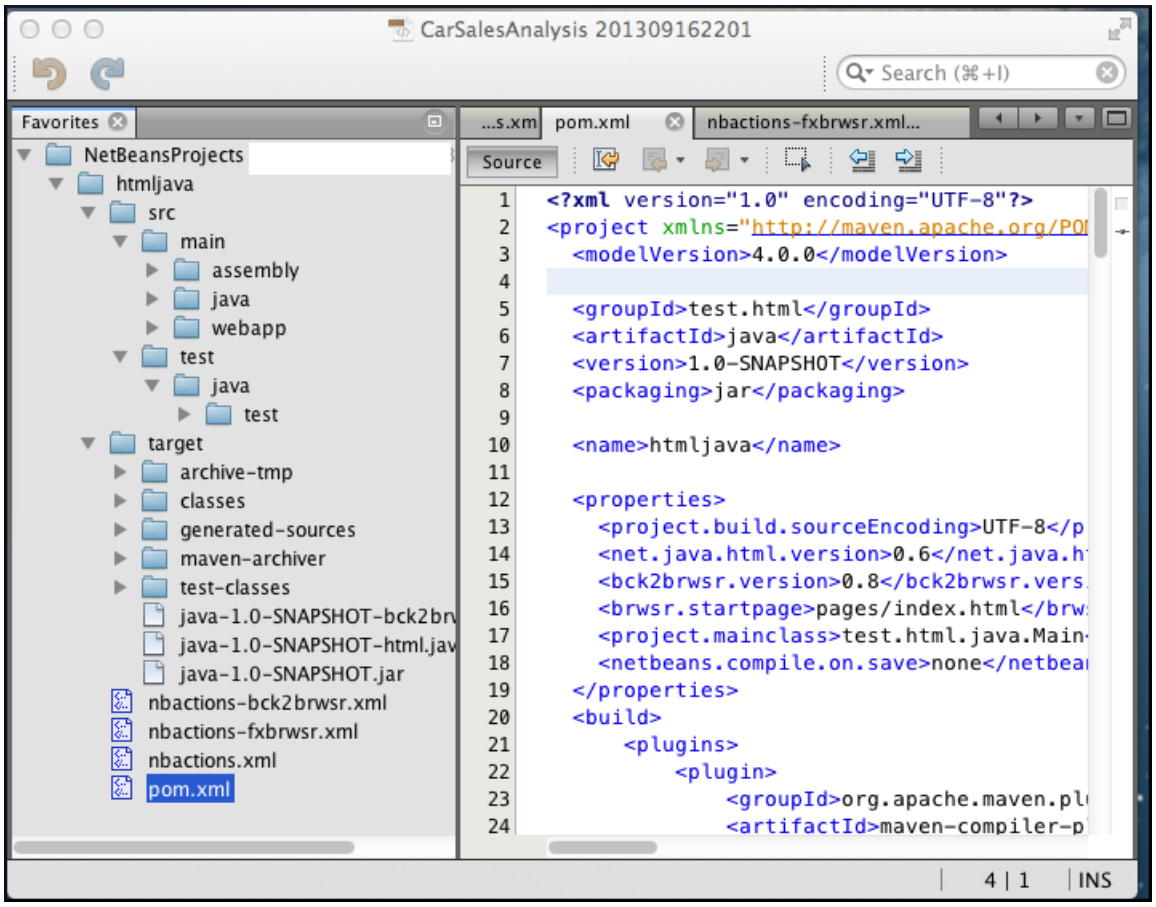
Image Editor

As you can see above, images are opened (either from the Favorites window or via the Open File menu item) into a window specifically for images, while the Navigator window shows you a birds eye view of the image.

1.2.5.5 XML Editor

Users may need to edit XML files of one kind or another in your application. In the *ide* cluster, select *XML Text Editor* and you will be prompted via the red Resolve button to include other XML-related NetBeans modules. In other words, to help you, the red Resolve button becomes enabled to prompt you to let the IDE automatically add required dependencies for you. It does this by analyzing the selected modules and then identifying their dependencies, so that you do not manually need to do this. When you have done so, you will have an XML editor in your application for working with XML files.

Open an XML file (either via the Favorites window or via the Open File menu item) and you will see the XML Editor, as shown below.

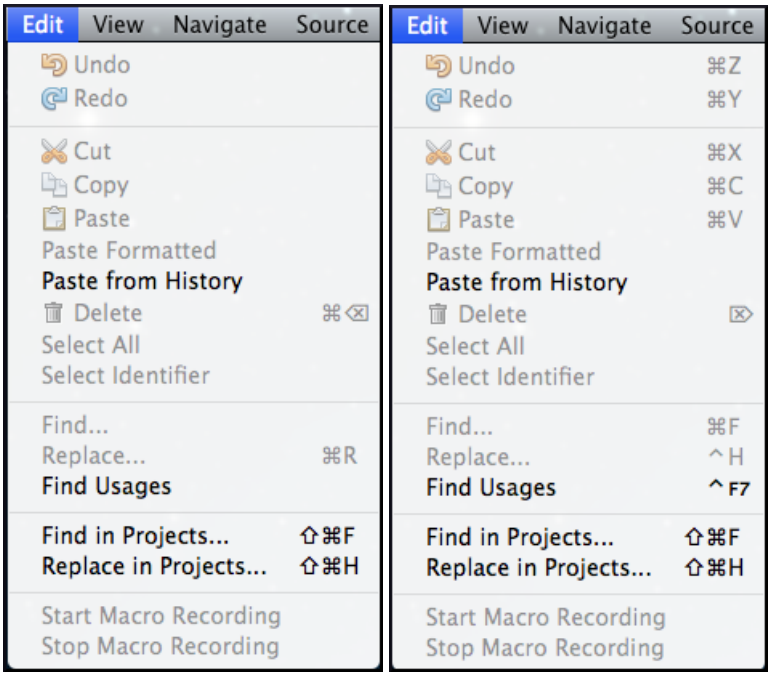


XML Editor

The XML editor can be extended in various ways, such as via new popup menu actions and toolbar buttons. The topic of extending source editors, and of creating them from scratch, is not in the scope of this book because, to do it justice, it deserves a book of its own. A future book by the same authors will address this topic.

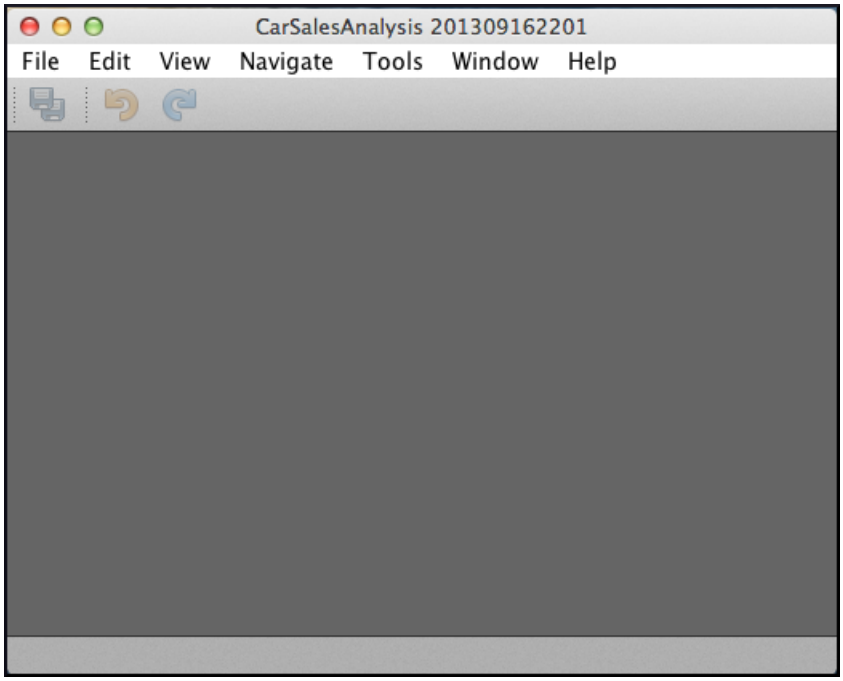
1.2.5.6 IDE Defaults

By default, keyboard shortcuts are not shown for the Undo/Redo/Cut/Copy/Paste menu items in your application, as well as some others, such as Save, as can be seen in the screenshot on the left below. To include them, select *IDE Defaults* from the *ide* cluster, which has the effect that keyboard shortcuts are shown, as shown in the screenshot on the right below.



1.2.5.7 Apple Application Menu

On Mac OSX, the Apple menubar is automatically used, instead of the application's menubar. To exclude this behavior, remove the *Apple Application Menu* module from the *platform* cluster, with the result shown below.



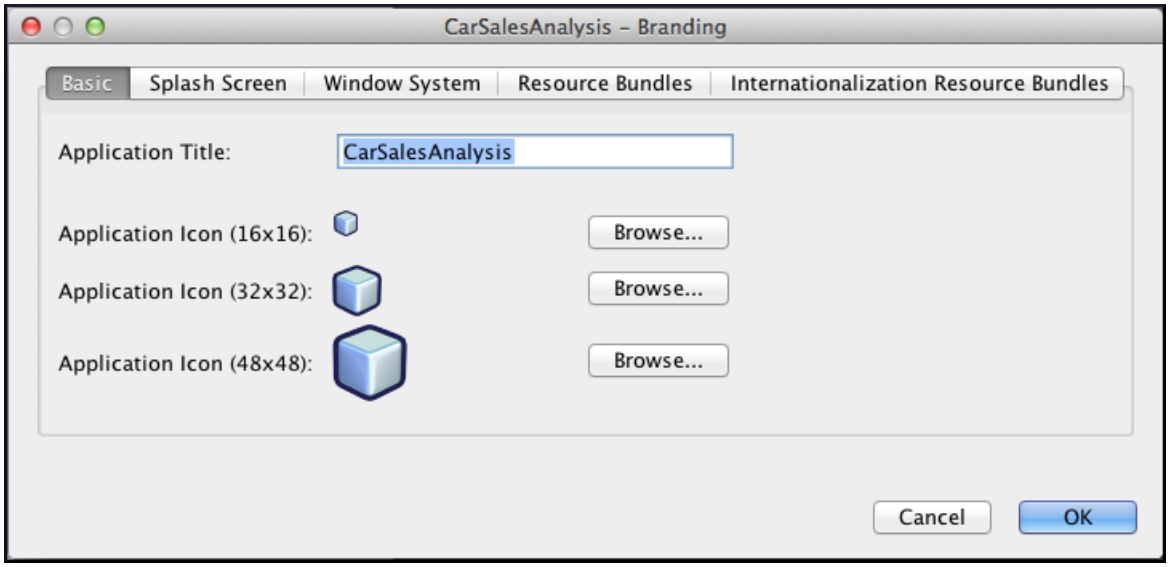
Apple Application Menu

The integration of the application menubar with that of the operating system for Ubuntu Linux is provided by the Java Ayatana plugin.

http://plugins.netbeans.org/plugin/41822/java-ayatana

1.2.6 Branding

Right-click the project and choose Branding. The Branding dialog appears, as shown below.

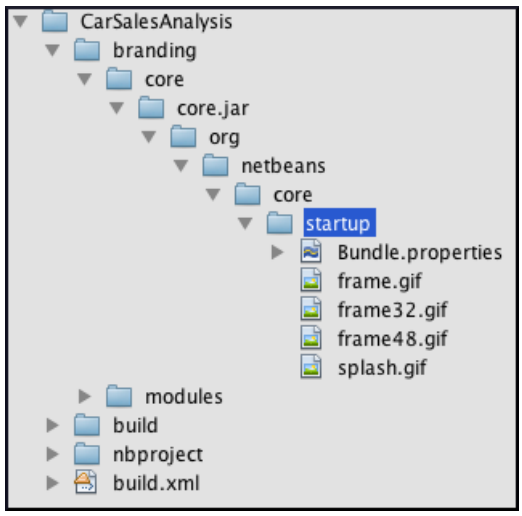


Branding

The tabs you see above are described below.

Tab	Description
Basic	Defines the text displayed in the title bar, as well as the icons shown in the top left of the main window, together with other areas of the NetBeans Platform.
Splash Screen	Defines the splash screen, progress bar, and text message displayed during the startup of the application.
Window System	Not all applications need to be as flexible as the NetBeans Window System is by default. In this tab, you can remove some of the flexibility of the Window System, such as by disabling *Window Drag and Drop* or *Floating Windows* or *Window Resizing* for the whole application.
Resource Bundles	Lets you look for labels used in the NetBeans Platform and replace them with your own.
Internationalization Resource Bundles	Lets you search through localized labels and replace them with your own.

After you make a change in the Branding dialog, such as the icons or splash screen, and click OK, you will see in the Files window that the images you specified as icons and splash screen have been copied into the branding folder, as shown below.



Branding

The folders above are turned into JARS, overriding the equivalent JARs in the NetBeans Platform, when the application is built.

1.2.7 Configuration File

Each application on the NetBeans Platform has a configuration file, in its installation directory's *etc* folder. For example, for NetBeans IDE, this is the "netbeans.conf" file, which most NetBeans IDE users know about and use frequently, since that's where the IDE's user directory and JVM arguments are defined. The same is true for any application created on top of the NetBeans Platform, that is, each application has its own configuration file.

When you choose *Package as | ZIP Distribution* in the IDE, or if you run the related Ant target on the command line, the default configuration file from *harness/etc/app.conf* is added to the ZIP distribution of your application. However, you can provide your own configuration file instead of the one automatically provided. For example, if you want to set the Nimbus look and feel for the application, you can add *laf Nimbus*, preceded by two hyphens, to the *default_options* key of your own configuration file, as shown below.

```
default_options="--laf Nimbus"
```

Normally, several other values, in addition to *laf*, are added to the *default_options* key in the configuration file, such as JVM switches for tuning the application for performance, as documented below:

https://performance.netbeans.org/howto/jvmswitches/index.html

To include your own configuration file, instead of the one provided by default, set the *app.conf* property in the *project.properties* file of the application, as shown below. Then, when you choose *Package as | ZIP Distribution*, your own configuration file is included in the ZIP file, instead of the default configuration file. For example,

set the following, which specifies that the file you want to be the configuration file of the application is in the *nbproject* folder of the application.

```
app.conf=nbproject/carsalesanalysis.conf
```

The above key/value pair assumes that there is a file named *carsalesanalysis.conf* in the *nbproject* folder of the application.

1.2.8 Distribution

When the application is complete, package it up by right-clicking it and choosing *Package As*. You can create a ZIP, including launchers, such as an *.exe*, installers, Java web start, and Mac OSX distributions.

Chapter 2: Module System

Modules enable development departments to organize their workflow around coarse-grained features, while simultaneously providing flexibility for users, who can install, enable, disable, and uninstall features as needed.

Far beyond the world of software development, modularity is well established. Poems can be broken into verses, music into measures, fields into plots, and houses into rooms.

2.1 Concepts

Refactoring a monolithic application into its distinct parts makes so much sense that it almost feels superfluous to explain what a module is and what its benefits are. Nevertheless, a brief explanation of *what* and *why* is needed, for those who are new to the concept, and especially for those who are familiar with it, since their understanding of modularity may be different to the implementation of the concept in the NetBeans Platform.

2.1.1 Definition

A NetBeans module is a JAR file, in other words, a Java archive containing a set of classes and other files that make up the module. What makes a JAR a module is comparable to what makes a JAR an OSGi bundle, that is, a set of special keys in the JAR manifest file that mark it as a module and tell the NetBeans Platform what to do to install it. The NetBeans module system provides an alternative to OSGi, one that has all the most important elements of OSGi, without its complexity.

Conceptually, a NetBeans module is a wrapper around a JAR file, providing it with its own classloader and version number. By default, any classes within a module cannot be *seen* by any classes in any other module. The *import* statement cannot be used to import such classes. In fact, compilation fails for any class that imports classes from packages that have not been exposed by the modules in which they are found. The encapsulation provided by a NetBeans module protects its content from being misused, since the module needs to explicitly expose a package for other modules to be able to use the classes found within the package.

Each distinct feature in a NetBeans Platform application is organized into a distinct module. What a *feature* is, and what a module's *boundaries* are, is something about which the NetBeans Platform is not opinionated. Development teams are free to use NetBeans modules in whatever way makes sense to their business needs. How many modules an application should have, and how *large* a module should be, are also decisions about which the NetBeans Platform has no opinion. Multiple small modules may provide a flexible structure, but may be difficult to maintain, while organizing the complete application into a single module, while possible, runs counter to the aims of modularity. Also see section 2.1.5 on Modularity below.

One point of potential terminology confusion is the use of the terms *plugin* and *module*. For most practical intents and purposes, there is no difference. A module is simply a unit of code that you can *plug in* to the

NetBeans Platform. The term *plugin* has been popularized by various other environments and can easily be applied to modules created for the NetBeans Platform as well. Traditionally, the term *module* is used in the NetBeans Platform environment, since the NetBeans Platform itself is composed of modules. For example, you cannot say that the core NetBeans Platform is composed of *plugins*. On the other hand, if you are creating new features for the NetBeans Platform but your feature set is composed of *multiple modules*, you might prefer to refer to those modules *collectively* as a single plugin.

2.1.2 Benefits

Without a doubt, rearchitecting an application into a modular structure is painful. On the other hand, maintaining spaghetti code is also painful. Development teams who continue to refuse to restructure their large and unwieldy applications into more manageable parts run the risk of having severely buggy projects that increasingly become unmaintainable.

In the context of software development, the notion that functionality should be discrete and capable of being added or removed painlessly has advantages to the following:

2.1.2.1 Developers

Having features organized within distinct modules helps development teams since, typically, developers within a team work on distinct features of the application.

In monolithic applications, the classes constituting the features are spread throughout the application, whereas in a modular application all classes defining a feature are found coherently within the same module or group of modules. Therefore, a set of modules can be assigned to a particular group of developers, making the boundaries of the modules the boundaries of responsibility for groups of related developers.

2.1.2.2 Users

Large applications have so many features that end users typically do not need all of them. To enhance the performance of the application and avoid an unnecessarily cluttered user interface, end users can enable or disable the features within the application by installing or uninstalling the related modules. A monolithic application does not allow for this kind of flexibility at all.

2.1.2.3 Revenues

To entice potential end users to adopt a piece of software, release a basic set of features for free. Once the basic set of features have proved useful to users who have freely begun using the software, advanced features can be made available, in the form of new modules, at a price.

Each module has its own lifecycle, including hooks for inserting licensing checks and security authentication. In this way, an interesting and lucrative financial model can be constructed on top of the modularity that NetBeans Platform applications inherently provide.

2.1.3 Characteristics

Modularity places the following requirements on the modules constituting a modular application.

2.1.3.1 Deployment Format

To support the ideals of modularity, it should be possible to make a module available to users via a single deployment package. Comparable to a JAR achive, and building on top of the same specification and concept, it should be possible to bundle an arbitrary set of source files and other resources into an archive for distribution to users of the application.

The NetBeans Platform equivalent of the JAR archive format is the NBM archive format. The letters NBM stand for "NetBeans Module". Together with module-specific metadata, such as instructions about the conditions under which the module should be activated within the application, the classes making up the module are packaged into an NBM file.

In Ant-based NetBeans modules, a target is included for generating the NBM file, while a goal is available in Maven-based NetBeans modules for the same purpose. NetBeans IDE includes a "Create NBM" menu item for NetBeans module projects, so that NBM files can be created in the IDE. NBM files can be inspected in the same way as any other archive, such as via a ZIP utillity, for example.

2.1.3.2 Uniqueness

It should be possible to uniquely identify a module. For that purpose, a unique String as an identifier for the module is a mandatory part of manifest files in NetBeans modules, using the *OpenIDE-Module* key. Via tools in NetBeans IDE you can define the unique identifier and modify it as needed.

Be aware that the *OpenIDE-Module* key, together with versioning details, is also required for providing automatic updates in the Plugin Manager.

2.1.3.3 Versioning

Versioning is a feature available to modules. Those responsible for the final application can then select modules and versions of modules that are most suitable for a specific business need. It is not necessarily advisable to select the latest version of a module. The latest version might be too buggy or not cooperate well with the rest of the application. It may be better to select an older version of a module or even to not include the given module in the final application at all.

NetBeans modules differentiate between *specification* versions and *implementation* versions.

Versioning Type	Description
Major Release	The version indicating the *generation* of the module, that is, only increment this version when, for example, the architecture of the module is rewritten. The major release version is appended to the end of the *OpenIDE-Module* key in the manifest, such as *org.carsales.carviewer/1*, where */1* indicates that this is the first major release of the module.
Specification	The version of the officially exposed interfaces, defined by the *OpenIDE-Module-Specification-Version* key in the manifest. The assumption is that the interfaces are a public API and that the new version is backward compatible with previous versions. Following convention, the first bug fix release of a module appends *.1* after the previously defined specification version, such as from *4.0* to *4.0.1*.

Versioning Type	Description
Implementation	The implementation state of a module, define by the *OpenIDE-Module-Implementation-Version* key in the manifest. The implementation version is not assumed to be backward compatible and can only be used within a specific version of the application. This is useful if a module explicitly depends on one specific implementation of another module, which should be avoided as far as possible.

Related documentation is below.

- http://bits.netbeans.org/dev/javadoc/org-openide-modules/org/openide/modules/doc-files/api.html
- http://wiki.netbeans.org/VersioningPolicy

2.1.3.4 Exposed Interfaces

Each module should be able to define public interfaces enabling other modules to access related features. In standard Java application, well-intended agreements and conventions about accessing dependencies are fragile and difficult to enforce.

The NetBeans Platform enforces access conventions between modules. Only if a class is in a package that has been explicitly exposed to the rest of the application, and a module has explicitly expressed an interest in depending on it, can code from one module be used in another module.

Packages to be exposed to other modules in the application are declared via the *OpenIDE-Module-Public-Packages* key in the manifest. By default, no packages are made public to other modules in the application. Only classes in packages that have been explicitly declared in the manifest of, either manually or via the tools in NetBeans IDE, are made available via NetBeans Platform classloaders to modules that have a declared dependency on it.

2.1.3.5 Declarative Dependencies

Each module should be able to declare which of the other modules in the application are required for it to function.

In NetBeans modules, the *OpenIDE-Module-Dependencies* key in the manifest specifies the environment required by the module, that is, the unique name and version of the modules that need to be available for the module to be loaded into the application. The NetBeans Platform checks the availability of the required dependencies and, if the dependencies are met, loads the module into the application.

2.1.3.6 Activation Type

Sometimes a module cannot operate alone, but rather needs to act in sync with other modules. When one module is activated, a set of modules is enabled. Similarly, when one is disabled, more than one can be disabled as well. In a sense, modules behave in orchestration, in the sense that one guides the state of the others. This is a very important aspect of modules running inside a NetBeans Platform application. The way a module is enabled and disabled is determined by the activation type.

There are three basic types of modules in the NetBeans Platform, that is, regular, autoload, and eager. Each of these is useful for the *role* a module can have.

Activation Type	Role
regular	The module represents important, user-visible functionality. It will act as a *director* of the enablement state of other modules. Regular modules are listed in the Plugin Manager.
autoload	Certain modules should not be controlled by end users at all. For example, modules providing useful libraries should be enabled when some other module needs such a library. It should not be possible to disable a library module via an explicit user action.
eager	In contrast to autoloads, eager modules are enabled as soon as possible like an eager trumpet in an orchestra that just jumps into a song at the earliest opportunity. Like autoload modules, eager modules cannot be disabled directly by the user.

Also see section 2.1.4.4 on Visibility below for details, especially the discussion on *kit modules*.

2.1.3.7 Lifecycle

For a module to truly support modularity, it must have its own independent lifecycle.

In the case of NetBeans modules, the NetBeans Platform manages the NetBeans module lifecycle. The NetBeans Platform handles the loading and configuration of NetBeans modules, while also processing the installation code, if any, and the unloading of modules when the application shuts down.

2.1.4 Manifest

Each module in a NetBeans Platform application has a manifest, providing a description of the module and its environment. When the NetBeans Platform loads a module, the manifest is the first file it reads.

If the *OpenIDE-Module* key is identified in the manifest, the NetBeans Platform knows it is dealing with a NetBeans module. As a result, the *OpenIDE-Module* key is the only mandatory key in the manifest. Its value can be any identifier, though normally the code name base, that is, the unique identifier of the module, is used.

2.1.4.1 General

Below are the general keys that can be defined in the NetBeans module manifest.

Key	Description
OpenIDE-Module	Defines a mandatory unique identifier for the module.
OpenIDE-Module-Name	Defines the display name for the Projects window and Plugin Manager.
OpenIDE-Module-Short-Description	Defines a short description for the Plugin Manager.
OpenIDE-Module-Long-Description	Defines a long description, listing the complete feature set, for the Plugin Manager.
OpenIDE-Module-Display-Category	Defines the virtual group to which the module belongs.

Key	Description
OpenIDE-Module-Install	Defines path to ModuleInstall class.
OpenIDE-Module-Layer	Defines path to module layer.xml file.
OpenIDE-Module-Public-Packages	Defines the packages public to other modules.
OpenIDE-Module-Friends	Provides access to the module's public packages only to the modules listed here.
OpenIDE-Module-Localizing-Bundle	Defines path to module's localization bundle.

2.1.4.2 Dependencies

The following manifest keys are used to define dependencies and versions.

Key	Description
OpenIDE-Module-Module-Dependencies	Defines dependencies between modules, optionally including the lowest required version number.
OpenIDE-Module-Package-Dependencies	Defines the rare case where a module depends on a specific package.
OpenIDE-Module-Java-Dependencies	Defines the rare case where a module depends on a specific Java version.
OpenIDE-Module-Specification-Version	Defines the specification version, usually written in the Dewey decimal format.
OpenIDE-Module-Implementation-Version	Defines the implementation version, usually by means of a timestamp.
OpenIDE-Module-Build-Version	Defines the build version, usually by means of a date stamp.
OpenIDE-Module-Module-Dependency-Message	Defines text displayed if a module dependency cannot be resolved. It may be normal to have an unresolved dependency, in which case it is a good idea to show the user a helpful message, informing them where the required modules can be found.
OpenIDE-Module-Package-Dependency-Message	Defines text displayed if a module dependency cannot be resolved. It may be quite normal to have an unresolved dependency. In this case, it is a good idea to show the user a helpful message, informing them where the required modules can be found.
OpenIDE-Module-Deprecated	Defines the module as deprecated. A warning is logged if the user tries to load the module into the application.

Key	Description
OpenIDE-Module-Deprecation-Message	Defines text to add information to the deprecated warning in the application log. It is used to notify the user about alternate module availability. Note that this message will only be displayed if the attribute OpenIDE-Module-Deprecated is set to true.

2.1.4.3 Services

The following keys are used to define service information:

Key	Description
OpenIDE-Module-Provides	Use this attribute to declare a service interface to which this module furnishes a service provider.
OpenIDE-Module-Requires	Alternatively, declare a service interface for modules needing a service provider. It doesn't matter which module provides an implementation to this interface. This enables the definition of modules that depend on a specific operating system, since this key is used to check the presence of a particular token. The following tokens are available: *org.openide.modules.os.Windows*, *org.openide.modules.os.Linux*, *org.openide.modules.os.Unix*, *org.openide.modules.os.PlainUnix*, *org.openide.modules.os.MacOSX*, *org.openide.modules.os.OS2*, and *org.openide.modules.os.Solaris*. For example, you can provide a module that automatically loads on Windows systems but automatically deactivates on all others.
OpenIDE-Module-Needs	This key is an understated version of the *Require* attribute and does not need any specific order of modules. This may be useful to API modules that require a specific implementation.
OpenIDE-Module-Recommends	Set optional dependencies. If a module provides, for example, a *java.sql.Driver* implementation, it will be activated, and access to this module will be enabled. Nevertheless, if no provider of this token is available, the module defined by the optional dependency can be executed.
OpenIDE-Module-Requires-Message	Like two previous similar messaging keys, this defines a message displayed if a required token is not found.

2.1.4.4 Visibility

The following keys are used to define the visibility of NetBeans modules for enablement, installation, disablement, and uninstallation:

Key	Description
AutoUpdate-Show-In-Client	Defines whether the module is displayed in the Plugin Manager or not.
AutoUpdate-Essential-Module	Defines whether the module is essential to the application so that it cannot be deactivated or uninstalled.

Together with the visibility keys, described above, there is the concept of *kit modules*. Each module visible in the Plugin Manager, that is, when *AutoUpdate-Show-In-Client* is set to *true*, is considered to be a *kit module*. All modules on which the kit module defines a dependency are handled in the same way as the kit module, with the exception of non-visible modules that depend on other kit modules. For example, if a kit module is deactivated, all dependent modules will be deactivated as well.

This lets you build wrapper modules to group several related modules for display to the user as a single unit. For example, create an empty module in which the attribute *AutoUpdate-Show-In-Client* is set to *true*, while defining a dependency on the modules to be grouped. Then, in the dependent modules, set the *AutoUpdate-Show-In-Client* key to *false*. The dependent modules will not be shown in the Plugin Manager, only the kit module, thereby simplifying the process of installing new features into your application.

2.1.5 Modularity

When you're new to modular architectures, a question that is likely to arise is: *"How should I determine the best way to split my application into modules?"*

In general, there's no single answer for knowing how to split an application into modules. Here are some factors to consider.

Factor	Description
Porting/From Scratch	If you're designing a new application from scratch, it's easier to achieve smallness in your modules and you'll have a better design, that is, looser coupling, because of it. On the other hand, if you're porting an existing application to the NetBeans Platform, you'll probably just have one or two big modules at first and later, as you work with your code more and while working on new features, you'll achieve smallness as you make things more modular over time.
Complexity	The more tiny modules you have, the more complexity you add to the deployment and management of your application. Having hundreds of modules will make it difficult to maintain an overview, especially for newcomers to the development team, so think carefully before making every package, or maybe even worse, every class, its own module. On the other hand, package boundaries are a good place to start, if your packages are well-structured.

Factor	Description
Rate of Change	Usually, not all the modules in your application will evolve at the same rate. You may want to think twice before putting things together in one module that clearly have different rates of change. For example, user interfaces tend to change more often than business logic. Also, in a user interface, business logic such as file importers or exporters might be things you want to change or extend over time and you don't want to redeploy the whole UI every time you do that.
Ownership	Especially in larger projects, there are often multiple teams, with each team focused on a specific technical expertise. Alternatively, your teams might contain members with different specific disciplines. Consider aligning the way you design your modules with the way you design your teams, so that you don't end up with modules that need to be developed and maintained by multiple teams simultaneously.
Responsibility	You may come to the conclusion that the *size* of a module is less important than its *responsibility* within the application. Following this approach, start by writing down all the *features* that your application provides, using gerunds, such as "viewing", "editing", "reporting", and then try and organize your application architecture such that each module encapsulates all the code relevant for a specific feature. Related to this approach, think of the *tiers* of your application and modulerize accordingly.

For more on this topic, see http://java.dzone.com/news/how-to-split-into-modules.

2.2 Get Started

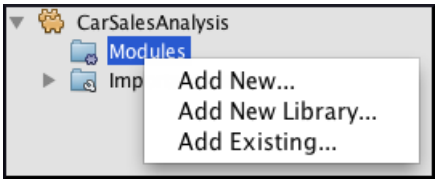
In this section, you learn how to create new modules and have them interact with each other.

For an example application related to this section, see sample 2.2 at https://github.com/walternyland/nbp4beginners.

2.2.1 Creation

Let's start by creating a new module in the *CarSalesAnalysis* application.

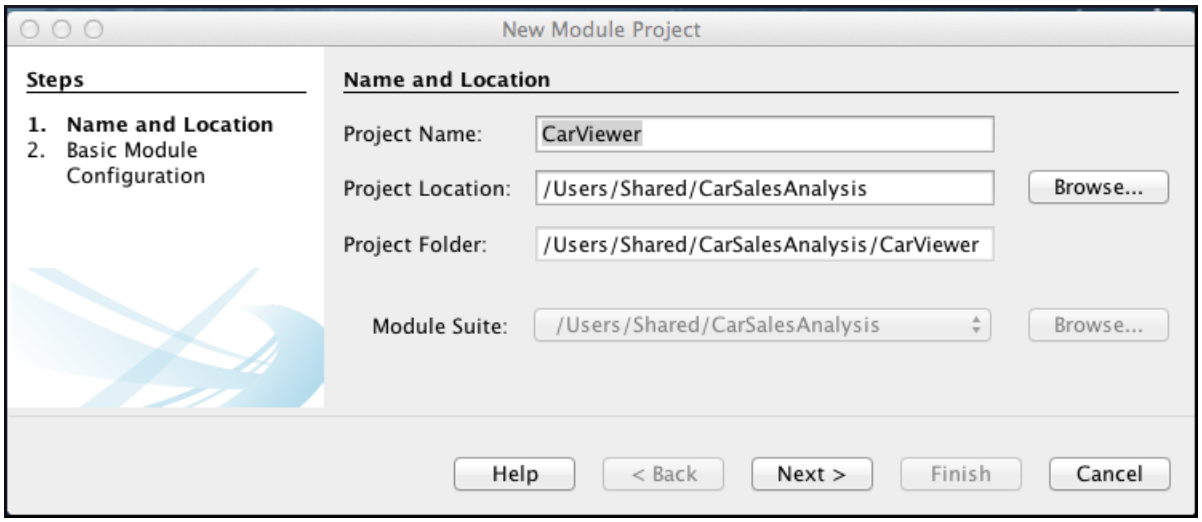
To do so, in the Projects window (Ctrl-Shift-1), right click the application's *Modules* node, and you should see a popup menu with the three menu items, *Add New*, *Add New Library*, and *Add Existing*, as shown below.



Popup menu on Modules node

Action	Description
Add New	Starting point for creating a new NetBeans module and including it in the application.
Add New Library	Starting point for wrapping an external JAR into a new NetBeans module and including it in the application.
Add Existing	Starting point for including an existing NetBeans module that is not yet part of the application into the application.

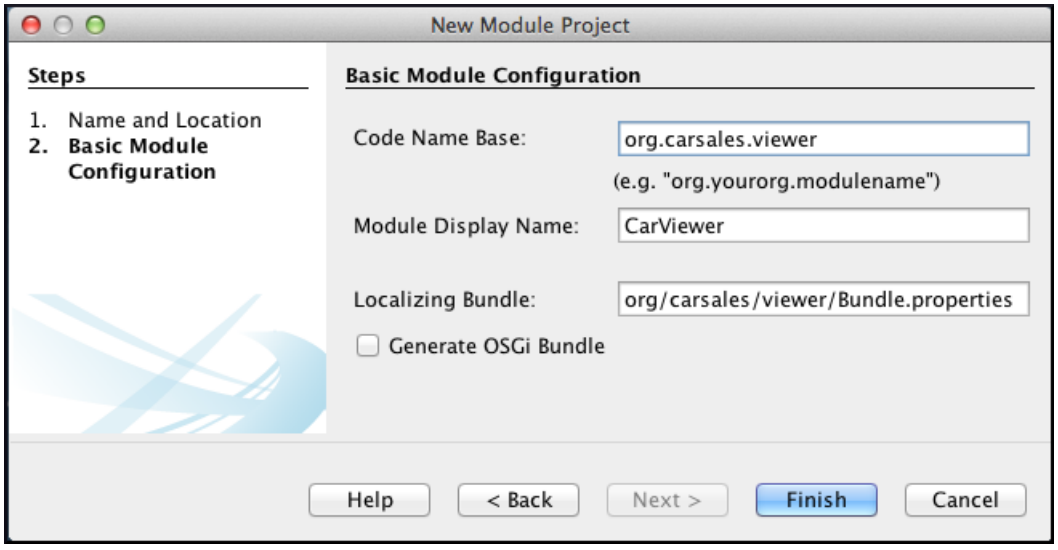
Choose *Add New*. The New Module Project wizard is shown.



First step of New Module wizard

Another way to get started with the wizard above, instead of right-clicking the Modules node in the application, is to go the the New Project dialog (Ctrl-Shift-N) and then choose NetBeans Modules | Module.

Give the module a name, in this case *CarViewer*, and click Next. In the second step of the wizard, specify the code name base, that is, the unique identifier of the new module, which in this case is *org.carsales.viewer*.

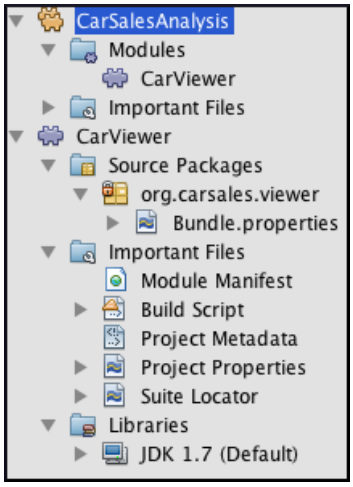


Second step of New Module wizard

Click Finish.

Now you have created a new module.

In the Projects window, the new module looks as follows.

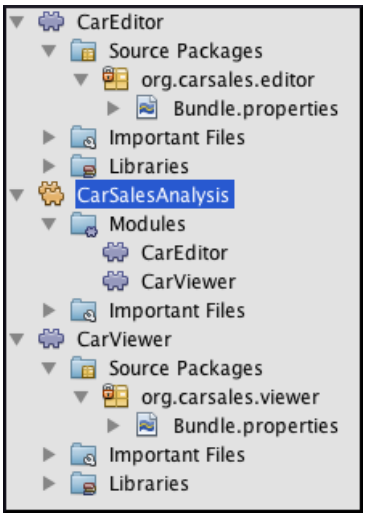


CarViewer module in CarSalesAnalysis application

Let's examine its structure.

File	Description
Bundle.properties	Defines the localization properties file, registered in the Manifest.
Module Manifest	Defines the Manifest template, where the optional *Bundle.properties*, *layer.xml* and *ModuleInstall* class can be registered, among other files.
Build Script	Defines the Ant script with predefined targets for compiling, running, etc.
Project Metadata	Defines the XML declaration file for dependencies and public packages, when the module is compiled the project metadata is combined with the Manifest template, to form the real Manifest file in the module.
Project Properties	Defines module properties such as Java compiler arguments.
Suite Locator	Defines the location of the application of which the module is a part.

Create a second module, named *CarEditor*, with code name base *org.carsales.editor*. You should now see the following.



CarEditor module in CarSalesAnalysis application

2.2.2 Commands

When you right-click the Module project in the Projects window, you see a popup menu containing NetBeans module project commands, the most significant of which for NetBeans modules are as follows.

Project Command	Description
Build, Clean and Build, Clean	When you *Build* a module, all its Java classes are compiled, annotations are processed, and a *build* folder is created (visible in the Files window). XML layer annotations are processed into a *generated-layer.xml* file, in *build/classes/META-INF*. When you *Clean* a module, the *build* folder is deleted.
Run, Debug, Profile, Test	Deploy the module, start the Debugger, start the Profiler, or run all the tests defined in the module.

Project Command	Description
Reload in Target Platform	When the application of which this module is a part is running, you can make a change in a NetBeans module and then redeploy it into the running application via this menu item.
Install/Reload in Development IDE	If you are creating a NetBeans module to extend NetBeans IDE, click this menu item to install the module into the currently running NetBeans IDE, that is, into the same development environment where you're developing the module.
Create NBM	When you're ready to distribute the NetBeans module as a separate entity, i.e., as a patch or as a separate feature for your users, click this menu item to create an NBM deployment archive, in the *build* folder (visible in the Files window).

2.2.3 Properties

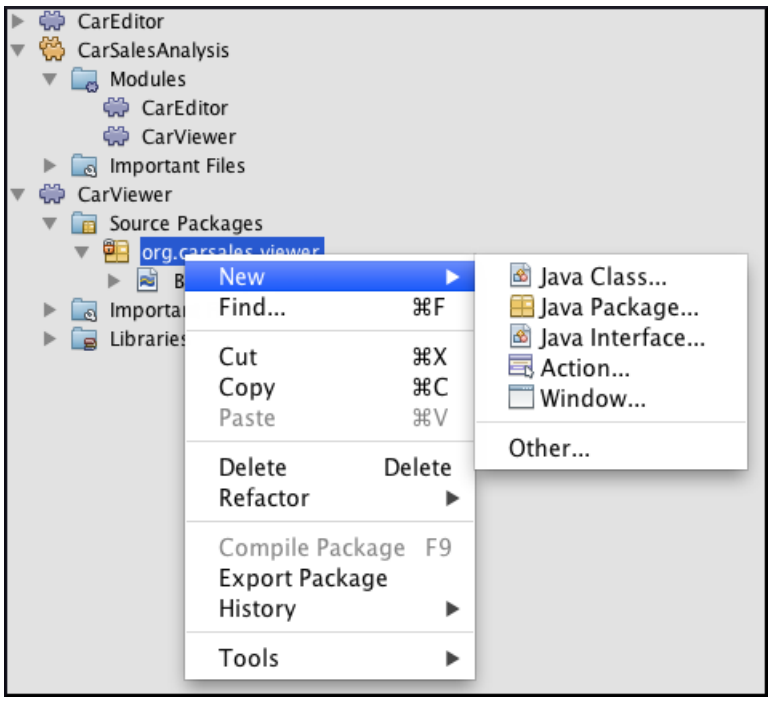
When you right-click the Module project in the Projects window and choose Properties, the *Project Properties* dialog opens. The following tabs are displayed.

Tab	Description
Sources	Shows the folder where module sources are found, the folder where the application is found, and lets you specify the Java source level of the code in the module.
Libraries	Shows the modules that the module depends on and the JARs wrapped within the module.
Display	Lets you set the display name, used in the Projects window and Plugin Manager, as well as other settings for the Plugin Manager, that is, the category, short description, and long description.
API Versioning	Shows the code name base, while letting you set specification and implementation versioning details, activation types (regular, autoload, or eager), public packages, and friends of the module.
Build	Shows where the JAR will be created, while letting you set whether a restart is needed after module installation, as well as the license, home page, and author, which are used in the Plugin Manager.
Formatting	Lets you set formatting rules for the Java editor used by Java source files in the module, such as tab size and line wrap.
Hints	Points to the application Hints tab where you can set Java hints to be shown throughout the application in the Java editor.

2.2.4 Templates

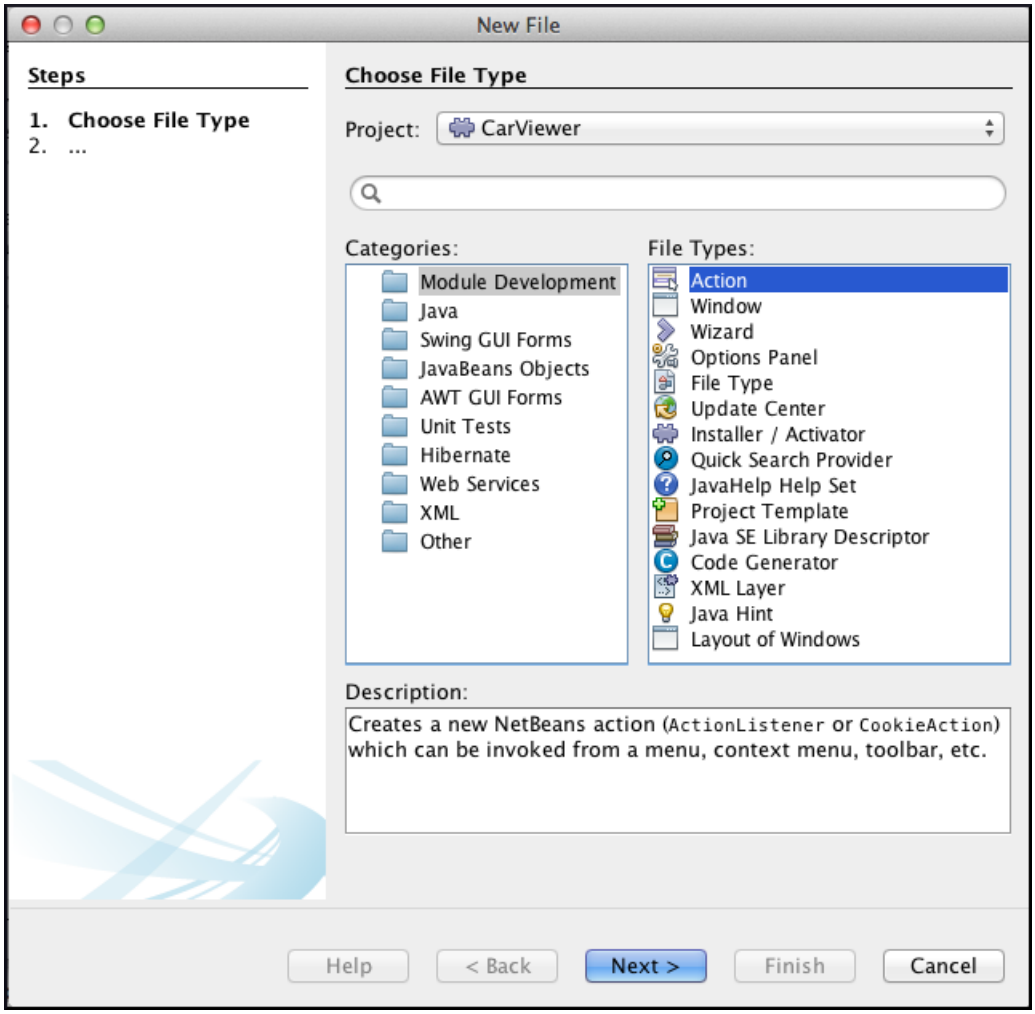
Several templates are provided by NetBeans IDE to help you get started with typical scenarios. Right-click on a package in a NetBeans module and choose *New*. You are then shown frequently used templates defined for

NetBeans modules and those templates that you have recently used.



Creating new files in a NetBeans module

Choose *Other...* in the list above and then choose Module Development. The New File dialog now shows you all the templates specifically useful in the context of NetBeans module development.



Templates for NetBeans Module Development

Template	Description
Action	Extend the Action System, with new menus, toolbars, shortcuts.
Window	Extend the Window System, with new windows.
Wizard	Create new multi-step dialogs.
Options Panel	Extend the Options window.
File Type	Create new MIME types, with new *DataObjects*.
Update Center	Register an XML file into the Plugin Manager, listing on-line modules.
Installer/Activator	Influence the lifecycle of a NetBeans module.
Quick Search Provider	Add new content to the Quick Search drop-down.
JavaHelp Help Set	Create definition and content for JavaHelp.
Project Template	Wrap a project into a wizard and register in New Project wizard.

Template	Description
Java SE Library Decriptor	Register a JAR into NetBeans IDE Library Manager.
Code Generator	Create a new code generator for NetBeans IDE.
XML Layer	Create and register a new *layer.xml* file.
Java Hint	Create a new Java hint for NetBeans IDE.
Layout of Windows	Customize the default window layout.

As you can see from the descriptions above, not all the templates relate to your work in creating applications on top of the NetBeans Platform. Some relate specifically to extending NetBeans IDE. Aside from the templates that are specific to NetBeans IDE, every template will be used in one way or another in various places in this book.

2.2.5 Dependencies

Let's now develop our small application a bit further, just to see how NetBeans modules can be set up to work together and interact with each other. The key concept that you will learn about is how NetBeans modules can *depend* on each other in a *structured and coherent* manner, as opposed to a manner that is unplanned, chaotic, haphazard, unstructured, and ultimately unmaintainable.

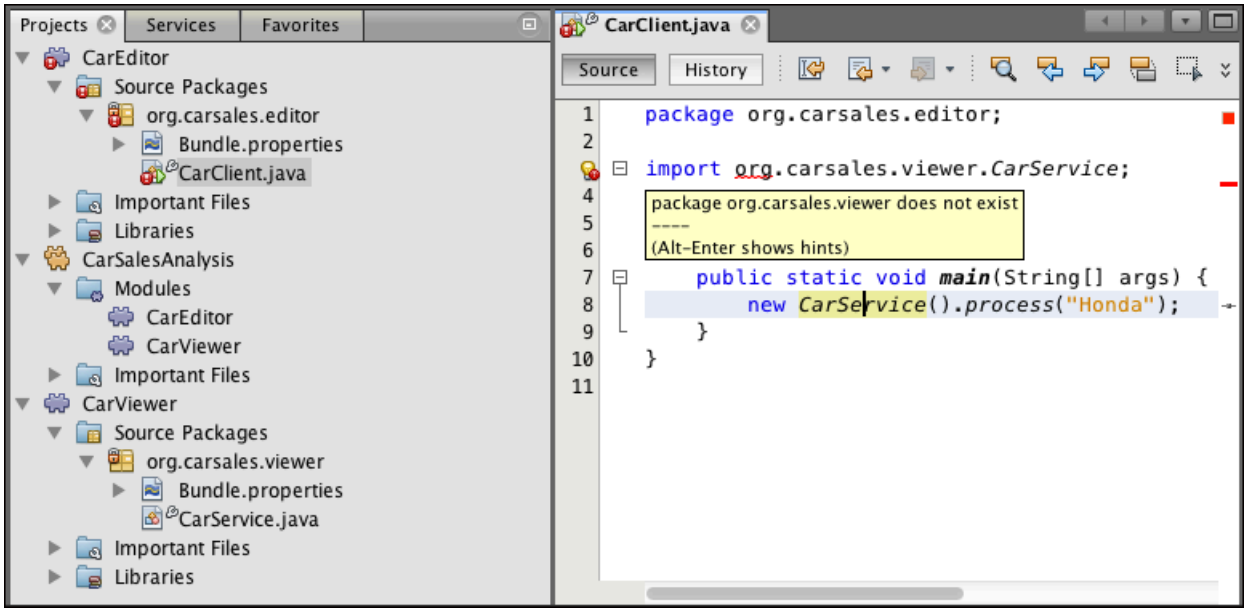
In the *CarViewer* module, create a new Java class named *CarService*, in a package named *org.carsales.viewer*, and add the following content to it.

```
package org.carsales.viewer;
public class CarService {
    public void process(String carBrand){
        System.out.println("Hello " + carBrand + "!");
    }
}
```

In the *CarEditor* module, create a new Java class named *CarClient*, in a package named *org.carsales.editor*, and add the following content to it.

```
package org.carsales.editor;
import org.carsales.viewer.CarService;
public class CarClient {
    public static void main(String[] args) {
        new CarService().process("Honda");
    }
}
```

In the Java editor, notice that an error message is shown in *CarClient.java*, as shown in the screenshot below.

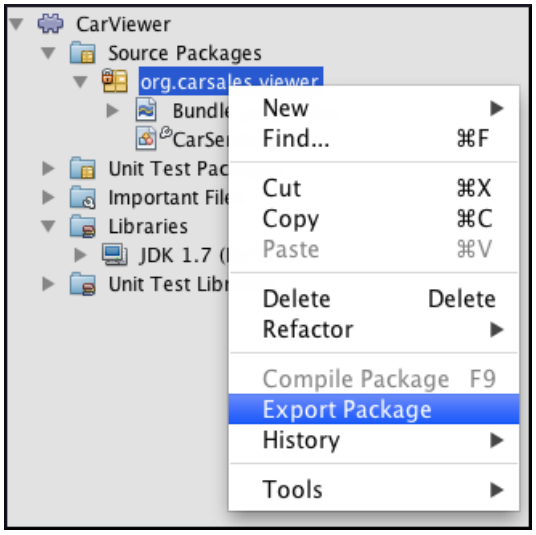


CarClient cannot use CarService

The error indicates that you are referencing a Java class, in this case *CarService*, without having imported it, and that the *import* statement fails because the referenced class is not on the classpath of the module. Also notice that when you try to compile the class or build the module, the compiler fails to compile the class, and error messages remain.

As discussed in the earlier section *Exposed Interfaces*, remember that the NetBeans Platform enforces access conventions between modules. Only if a class is in a package that has been explicitly provided to the rest of the application, and a module that wants to consume it has explicitly expressed an interest in doing so, can code from one module be used in another module. It is this mechanism that ensures that code is not used across modules unless the author of the module explicitly intended for cross-modular usage to be possible. In turn, this helps reduce unintended spaghetti code.

In the *CarViewer* module, right-click the *org.carsales.viewer* package and choose *Export Package*, as shown below.



Export Package menu item on NetBeans module package

After you choose *Export Package*, open the *Project Metadata* file in the *Important Files* node of the module. You should see that the package has now been declared as being public to the application.

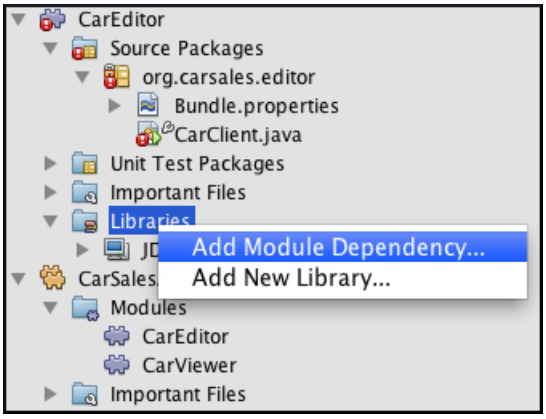
```
<public-packages>
    <package>org.carsales.viewer</package>
</public-packages>
```

The *Export Package* menu item is new from NetBeans IDE 7.4 onwards. The older and still available approach to achieve the same as the above, that is, expose packages as being public to the application, is to, instead of using *Export Package*, right-click the NetBeans module and choose *Properties*. When the *Project Properties dialog* opens, use the *API Versioning* tab to specify the packages that should be declared as being public to the application.

As a result, when the module is compiled, the *Manifest* in the NetBeans module will include the *org.carsales.viewer* package as a value of the *OpenIDE-Module-Public-Packages* key. You will *not* see this key in the *Manifest* file that is in the sources of your NetBeans module. Instead, that file is a *template* providing a starting point for the *real* manifest file that is created during the compilation process and included in the NetBeans module deployment package, that is, the NBM file.

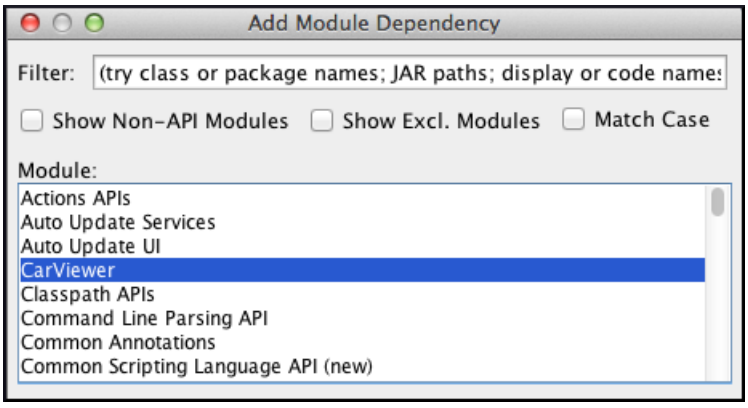
The above steps are those that need to be taken for any package that should be *provided* by a NetBeans module to other NetBeans modules in the application. However, this is only one side of the contract between two modules. The other side is that the *consumer* of a package in another NetBeans module should declare a *dependency* on that NetBeans module.

To declare a dependency, right-click the *Libraries* node in the *CarEditor* module, as shown below.



Add Module Dependency menu item on NetBeans module Libraries node

When you click *Add Module Dependency*, the dialog below opens. From the list, choose *CarViewer*, since that is the NetBeans module that provides the package containing the *CarService* class.



Add Module Dependency dialog with CarViewer module selected

Only modules that have at least one publicly declared package are shown in the *Add Module Dependency* dialog.

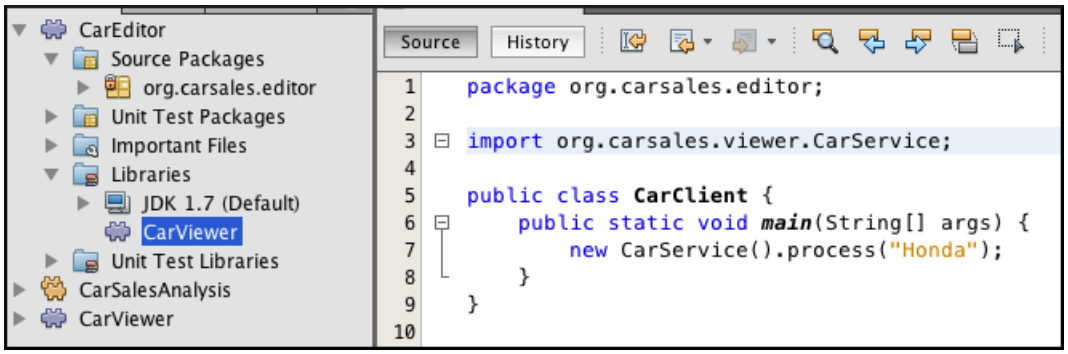
Also notice that there is a filter in the *Add Module Dependency* dialog. Use it when you know the name of the class you would like to use but not the name of the module that provides it. Type the name of the class and then the list narrows to show only those modules that provide a class matching the filter text.

After you click OK in the *Add Module Dependency* dialog, open the *Project Metadata* file in the *Important Files* node of the module. You should see that the module has been declared as a dependency.

```
<dependency>
    <code-name-base>org.carsales.viewer</code-name-base>
    <build-prerequisite/>
    <compile-dependency/>
    <run-dependency>
        <specification-version>1.0</specification-version>
    </run-dependency>
</dependency>
```

As a result, when the module is compiled, the *Manifest* in the NetBeans module will include the *org.carsales.viewer* module as a value of the *OpenIDE-Module-Module-Dependencies* key. Again, you will *not* see this key in the *Manifest* file that is in the sources of your NetBeans module because that file is a *template* providing a starting point for the *real* manifest file that is created during the compilation process and included in the NetBeans module deployment package, that is, the NBM file.

You should now be able to type the import statement, or have NetBeans IDE create it for you from the Java hint in the left sidebar, and no more error messages should be shown in the Java editor, as shown in the screenshot below.



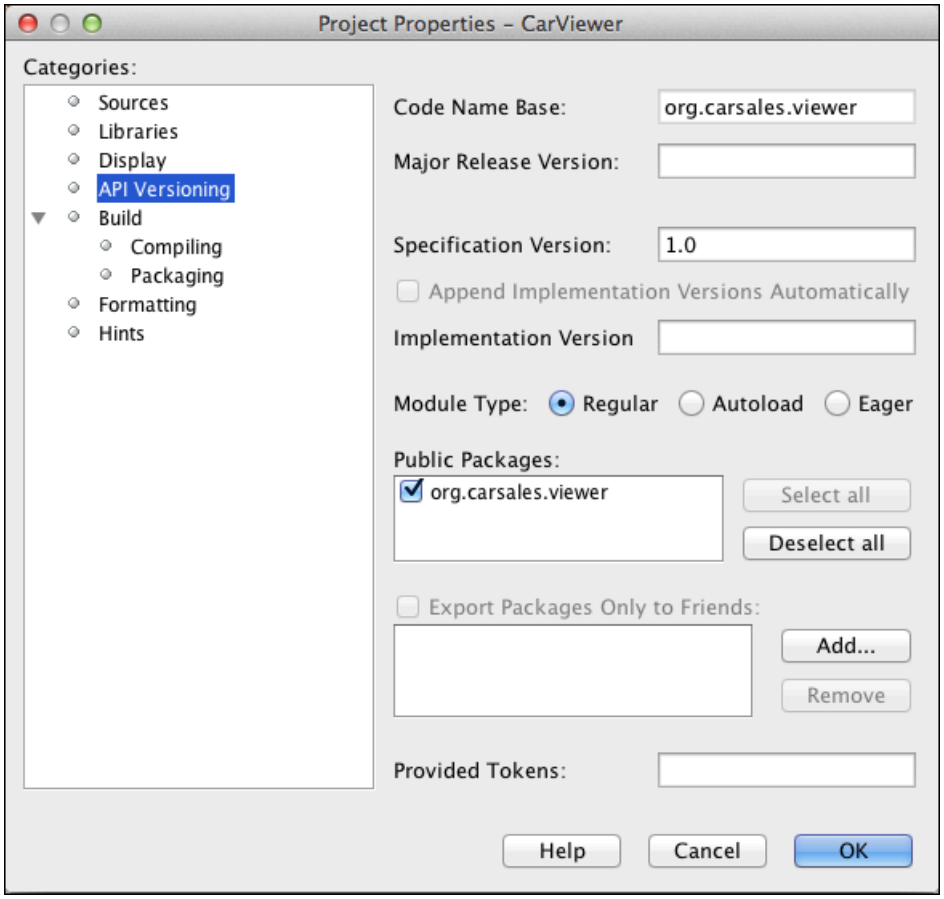
Dependencies correctly set, so no compilation errors

Therefore, remember that to use code across modules, the *provider* needs to make its package(s) public, while the *consumer* needs to depend on the provider. When these two conditions are true, the NetBeans Platform classloading mechanism enables the classes to *see* each other. However, while those conditions are *not* true, the class cannot be imported.

A nice side effect is that if a package is *not* public, no other module can use it, meaning that utility methods that should only be used within a module can stay hidden within the module if they are in a package that has *not* been made public. No longer will you have developers on your team abusing the intentions you had when writing your utility methods, unless you expose those classes for usage across the application.

2.2.6 Versioning

In the Project Properties dialog of each module, the *API Versioning* tab defines important metadata.



API Versioning

Setting	Description
Code Name Base	A string defining the unique ID of the module.
Major Release	The version indicating the *generation* of the module, that is, only increment this version when, for example, the architecture of the module is rewritten. The major release version is appended to the end of the *OpenIDE-Module* key in the manifest, such as *org.carsales.carviewer/1*, where */1* indicates that this is the first major release of the module.
Specification	The version of the officially exposed interfaces, defined by the *OpenIDE-Module-Specification Version* key in the manifest. The assumption is that the interfaces are a public API and that the new version is backward compatible with previous versions. Following convention, the first bug fix release of a module appends *.1* after the previously defined specification version, such as from *4.0* to *4.0.1*.

Setting	Description
Implementation	The implementation state of a module, define by the *OpenIDE-Module-Implementation Version* key in the manifest. The implementation version is not assumed to be backward compatible and can only be used within a specific version of the application. This is useful if a module explicitly depends on one specific implementation of another module, which should be avoided as far as possible.
Module Type	Sometimes a module cannot operate alone, but rather needs to act in sync with other modules. When one module is activated, a set of modules is enabled. See section 2.1.36 on Activation Type above for details.
Public Packages	A list of packages that any module depending on this module can reference.
Export Packages Only to Friends	Implementation classes in a module may be exposed to a defined set of so-called "friend" modules, that is, authors of other modules can ask to be listed as a friend, using their code name base, giving them privileged access to internal classes in the module.
Provided Tokens	Sets OpenIDE-Module-Provides in the MANIFEST.MF file. See section 4.1.1.5 on Requiring and Providing Services for details.

2.2.7 Lifecycle

Run the application right now and, even though the application compiles and deploys, your *System.out* will not be called and nothing will be displayed in the Output window.

Methods with the signature *public static void main* are meaningless in the NetBeans Platform. The NetBeans Platform has its own main method. To influence the startup sequence of the application, you need to influence the startup sequence of individual modules. Each module has its own lifecycle, which can be influenced via a class that extends *org.openide.modules.ModuleInstall*, which is comparable to an *OSGi Activator* in Eclipse RCP. The class must be registered in the *Manifest* file. When the module is loaded into the application, the NetBeans Platform finds the *OpenIDE-Module-Install* key in the *Manifest* and, if it is there, the NetBeans Platform will execute the value, which must be the fully qualified name of a class extending *org.openide.modules.ModuleInstall*.

To simplify the creation and registration of the *ModuleInstall* class, use the *Installer/Activator* template in the New File dialog. Right-click the *org.carsales.editor* package and choose New | Other and then Module Development | Installer/Activator. Click Next and Finish.

When you open the *Module Manifest* in the *Important Files* node of the module, you should see this key/value statement included now.

```
OpenIDE-Module-Install: org/carsales/careditor/Installer.class
```

The above has been added by the *Installer/Activator* template you used above. Put the content you had in your *CarClient* class into your *ModuleInstall* class, as shown below.

```
package org.carsales.editor;
import org.carsales.viewer.CarService;
import org.openide.modules.ModuleInstall;
public class Installer extends ModuleInstall {
    @Override
    public void restored() {
        new CarService().process("Honda");
    }
}
```

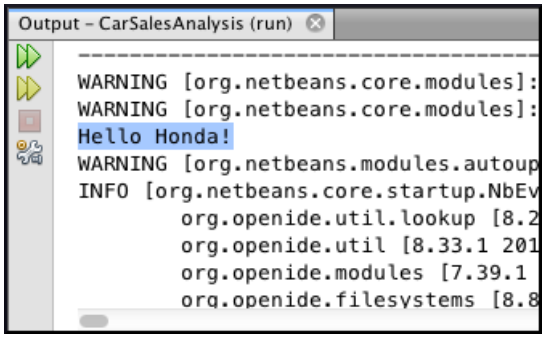
The NetBeans Platform *ModuleInstall* class is the equivalent of the OSGi *BundleActivator* class.

Key methods in the *ModuleInstall* class are as follows.

Method	Description
validate()	Called before a module is installed or loaded. When needed, certain load sequences, such as the verification of a module license, are set here. Should the sequence not succeed and the module not be loaded, throw an *IllegalStateException* method. This exception prevents loading or installing the module.
restored()	Called when an already-installed module is restored (during startup). Should perform whatever initializations are required.
closing()	Called when the NetBeans Platform application is about to exit. The default implementation returns *true*. The module may cancel the exit if it is not prepared to be shut down. Here, you can also test whether the module is ready to be removed. Only once this is true for all the modules in the application can the application itself shut down. You can, for example, show the user a dialog to confirm whether the application should really be closed.
close()	Called when all modules agreed with closing and the NetBeans Platform application will be closed. Here, you call the actions to uninitialize features, if needed. For example, you might close a data connection here.

When using these methods, consider whether the actions you're calling could be set declaratively instead. In particular, in the case of the methods *validate()* and *restored()*, consider that these methods influence the startup time of the whole application. For example, when services need to be registered, you could instead use entries in the System FileSystem, which is discussed in the next chapter. These enable the code to be invoked as needed, without impacting the startup time of the application as a whole.

Now run the application again and the Output window in NetBeans IDE displays the output.



Output window displaying System.out.println

An alternative approach to achieving the same as the above is to annotate a *Runnable* class with the NetBeans Platform *org.openide.modules.OnStart* annotation, as shown below.

```
package org.carsales.editor;
import org.carsales.viewer.CarService;
import org.openide.modules.OnStart;
@OnStart
public class Startable implements Runnable {
    @Override
    public void run() {
        new CarService().process("Honda");
    }
}
```

At compile-time, a NetBeans Platform annotation processor registers the above in the *Modules/Start* folder, within the folder *META-INF/namedservices*, in the *build* folder that you can see in the Files window of NetBeans IDE. At startup of the application, the NetBeans Platform looks in that folder for any *Runnables* it finds there and then invokes those *Runnables*.

The advantage of *@OnStart* over *ModuleInstall* is that no registration is needed in the *Manifest* file. The usage of annotations is preferable over registration via an external file such as the *Manifest*, since you do not need to switch out of your Java editor to another file and because it keeps all your code and registrations in the same place.

Finally, in the same way that the *ModuleInstall* class has a *close()* method, as listed in the table earlier, the same can be achieved via the *@OnStop* annotation on a *Runnable*, comparable to the *@OnStart* annotation vs. *restored()* above.

2.2.8 Distribution

A NetBeans module archive file, known as an *NBM file*, is a deployment package for delivery via the Web. The principal differences between NBM files and module JAR files are as follows.

Unique Feature	Description
Multiple JARs	An NBM file can contain more than one JAR file—modules can package any libraries they use into their NBM file.
Metadata	An NBM file contains metadata that NetBeans will use to display information about it in the Plugin Manager, such as the *Manifest* file a the license.
Signature	An NBM file may be signed for security purposes.

NBM files are ZIP files with a special extension. They use the JDK's mechanism for signing JAR files. Unless you are doing something unusual, you need not worry about the contents of an NBM file—just let the standard Ant build script for NBM creation take care of it for you.

Normally, you distribute your NetBeans module as part of your application. In other words, rather than creating an individual NBM file, you'll create a ZIP file of the whole application or you'll create an installer for installing the whole application.

2.2.8.1 Pull

Sometimes you will need to distribute the NBM file of an individual module, such as when you need to distribute a new feature in the middle of a release cycle or when you want to distribute a patch of a particular NetBeans module.

When this is needed, right-click the NetBeans module in the Projects window and then choose *Create NBM*. In the Files window, expand the *build* folder, and you will see your NBM file, ready to distribute to your users.

Once you have an NBM file, you can make it available as an explicit update by sending it to your users in an e-mail. Another, more elegant, approach is to create an update center, which is an XML file that lists NBM files. In the New File dialog, an Update Center template is provided for creating the required XML file format and for registering it in the application. Upload the XML file to your server, together with the NBM files, and the XML file will automatically be polled by the application for updates. Polling can be done at startup or at specified time periods.

2.2.8.2 Push

Circumstances might lead to the requirement that NBM files should be made available silently, without interaction with the user. The NetBeans Platform provides the Auto Update API for this purpose.