THE EXPERT'S VOICE® IN OPEN SOURCE

Covers NetBeans™ Platform 6.5

The Definitive Guide to NetBeans™ Platform

Learn to use the latest NetBeans™ Platform for rapid modular development of small and large rich client applications.

Heiko Böck

Translated by the NetBeans™ Platform Community

Foreword by Jaroslav Tulach, original NetBeans™ API architect

Apress®

The Definitive Guide to NetBeans™ Platform

Heiko Böck

Apress®

The Definitive Guide to NetBeans™ Platform

ISBN-13 (pbk): 978-1-4302-2417-4

ISBN-13 (electronic): 978-1-4302-2418-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jim Freeman
Technical Reviewers: Jaroslav Tulach, Geertjan Wielenga
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Douglas Sulenta
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Ellie Fountain
Proofreader: Nancy Sixsmith
Indexer: BIM Indexing & Proofreading Services
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Dedicated to the NetBeans Platform community

Contents at a Glance

Contents

Foreword

The best way to improve what you do is to find someone who will do it for you better than you could have done yourself. I've seen this principle in action over and over again. For example, it was fine designing the NetBeans IDE and NetBeans Platform on my own. Now, however, it is much better, since there are far more talented developers designing the various individual parts together. Similarly, I was OK writing documentation for the NetBeans Platform, but it is much better to have a group of enthusiastic people who produce tons of interesting tutorials and blogs on topics that I would never have thought up myself. Along the same lines, it was entertaining to contribute to a book about the NetBeans Platform. However, it is an order of magnitude better to see this one by Heiko! It is richer and more interesting. Plus, it covers topics that I never dreamed of covering myself.

My first encounter with this book dates back to 2007 when Heiko finished his German version. I was asked to review the book's content. However, as my German reading abilities are close to zero, my goal was more to read the sample Java code and ensure that correct patterns were used and described. It quickly became easy to see that everything was more than OK. Not only that, I could immediately see that the topics were extraordinary and that they brought new ideas into the NetBeans Platform world.

I am glad that Heiko's interesting insights are now available to a broader audience. Thank you Heiko! Thank you too, dear English translators!

Jaroslav Tulach
NetBeans Team Member

One of the wonderful things about the NetBeans Platform is that it is impossible to run out of new things to do with it. For me, it has been an inexhaustible source of inspiration—and that is still true ten years after I first started working with it!

I have coauthored two books on the NetBeans Platform in the past, and when we were finishing up, there were always things we wished we had more time or space to cover. So there can never be enough books about it—every author has a different perspective on the topic.

To Heiko I offer my heartfelt thanks for shining his unique insight on the NetBeans Platform; to you, dear reader, I wish you happy coding and joy in learning to love a framework that I have helped to build, and that has been a labor of love for many years.

Tim Boudreau
NetBeans Team Member

What I like most about this book is the number of interesting side roads you will find yourself traveling as Heiko takes you by the hand and helps you explore the NetBeans Platform. Many of the topics addressed here have not been addressed anywhere else—from persistence, to the Task List API, to the integration of JDK 6 features into NetBeans Platform applications. You will

have a very hard time avoiding learning many new things here, regardless of the amount of experience and skills you bring to the table.

Over the past two or three years, the NetBeans Platform has seen a continual surge in popularity in companies around the world. This is in part because of its modular architecture, in combination with its reliance on the standard Swing UI toolkit. But, certainly, the growing adoption of the NetBeans Platform can also be attributed to the enthusiasm of its users. Without them, the book you're now holding would certainly not have come to be. In fact, a small army from the NetBeans Platform community translated this book from scratch in the space of a single month. As one of them, I can say that I have learned an unlikely amount of details about the NetBeans Platform—details I would never otherwise have learned about. Many thanks to Heiko for this great book and for all the work that went into it, as well as all the work that went into updating it to 6.5!

Jim Freeman, the book's editor in Prague, deserves high praise for his editing skills, as well as for his sage advice and encouragement to me personally, and for his living room and cups of hot coffee every morning for the duration of the translation project. Many thanks also to Michaela Freeman for being so supportive, too. Also, much gratitude to the team at Apress for their close cooperation and precision work throughout: Steve Anglin, Grace Wong, Douglas Sulenta, and Damon Larson. On a personal note, based on the experiences of the past months, Jim Freeman and Damon Larson are really very highly recommended technical editors indeed.

Finally, to the translators, who tirelessly slaved on their assigned chapters, I express my warm gratitude: Zane Cahill, Stefan Alexander Flemming, Matthias Holzinger, Peti Horozoglu, Martin Klähn, Pavel Kotlov, Christian Pervoelz, Christian Enrique Portilla Pauca, Sven Reimers, Johannes Strassmayr, and Fionn Ziegler.

Some of the translators share their experiences translating the book and learning about the NetBeans Platform in the following paragraphs.

Geertjan Wielenga
NetBeans Team Member

Having been a developer of NetBeans Platform applications for the last eight years, I am always on the lookout for more and better documentation. And then Heiko published his book! Without any high expectations, I started reading it to write a review, and was astonished by the wealth of information Heiko managed to cover. So, always looking for ways to contribute to the NetBeans Platform, I volunteered to take part in the translation project to make all this available to a much broader audience.

If you've already read a book on the NetBeans Platform, you may ask yourself why you want to read this one. To make a long story short, it's the unique and detailed content about the basics of the NetBeans Platform that makes this book surpass the others. You will learn about creating loosely coupled modules using the Lookup API, details of the module system, and how to create your own full-scale applications based on the NetBeans Platform.

Thanks to Heiko for the effort he originally put into his book and the opportunity to be part of such an extraordinary project. We all owe Geertjan big, for being the one leading us, offering help, organizing, and pushing the translation effort.

Sven Reimers
Translator

This book is a great source of information and will hopefully be as valuable for other developers as it was for me. I'm thankful for the opportunity to contribute to this project and hope all readers will have as much fun reading as I had translating!

Johannes Strassmayr
Translator

Confronted with the task of porting a big legacy application to the NetBeans Platform, I've read all the literature on the topic and found this particular book a great source of practical examples and well-structured information.

It was a pleasure to read this book, so I was really honored to be part of the translation team. I hope the English version will help even more people to find their way into the NetBeans Platform community.

Pavel Kotlov
Translator

Translating this book was an interesting process—to do a good job, I had to take into account each aspect of the NetBeans Platform, as well as Java programming in general. This book is a good source of experience for programming in Java using the NetBeans IDE and the NetBeans Platform underneath it, which together provide many tools and components for working with Java. Combined with some practice and basic Java knowledge, this book will make you a better programmer.

Translating the chapters assigned to me, I learned a great deal about actions and components for building GUIs for Java applications—I'm confident that this book is a good reference for these topics.

Many thanks to Geertjan for giving me the chance to work on this translation. It has improved my translation abilities and allowed me to learn more about the NetBeans Platform. I look forward to hearing about readers' enjoyment of this book!

Christian Portilla Pauca
Translator

I well remember the first time I read the German version of Heiko's book because it was the moment when I really started understanding the ideas behind the most common NetBeans Platform patterns. So, I'm all the more happy to have been asked to help translate a part of my favorite NetBeans book into English.

It was a great experience to work on a team that managed to translate a whole book in just one month!

Stefan Alexander Flemming
Translator

The book caught my attention during the development of a product based on the NetBeans Platform. It gave me good ideas for solving many challenges and ensuring better product quality. I improved my knowledge of the NetBeans module system while translating the book. It is an honor to have been involved in the translation process.

While translating, I gained a better understanding of the window system, especially regarding the `layer.xml` file and its relation to the System Filesystem. Additionally, I learned a lot about defining and managing dependencies between modules with different versions, as well as integrating modules into the NetBeans Platform.

Fionn Ziegler
Translator

Three years ago, I started an internship in a department developing a NetBeans Platform application. Not having encountered NetBeans at all up to that point, I had to familiarize myself with both the IDE and its platform. Once I realized how easy it is to develop applications with the NetBeans IDE, I bade goodbye to my dear old friend Eclipse and welcomed the NetBeans IDE with open arms! I was later integrated into the development team and brought up to speed amazingly quickly. About a year later, Heiko published his book, and even more concepts became clear to me.

Then came the day the community effort to translate Heiko's book was announced. I volunteered my help to broaden the distribution of knowledge contained in this book to English-speaking readers and developers. After working on the chapter concerning the user interface, I realized that my understanding of the concepts conveyed had increased without my having been aware of it.

Heiko's work putting this book together helped a lot of people to develop NetBeans Platform applications, myself included. Geertjan, as manager of this project, has done tremendous work bringing this project to fruition. We owe many thanks to Heiko and Geertjan for doing what they did. I am hopeful that you as a reader can learn as much as I did when I first began digging my claws into it all. And I hope you enjoy the experience while doing so!

Martin Klähn
Translator

I had seen testimonials to this book frequently pop up on mailing lists and blogs, so when the opportunity presented itself to support a translation, I could not resist the appeal to be associated with this excellent book and a community project involving like-minded NetBeans Platform enthusiasts.

Fortunately, part of my assignment was the Visual Library API, a feature set I never had the opportunity to use before, so the task was also a great learning experience. At the outset, I thought the Visual Library API would be complex with a steep learning curve. However, Heiko shows how quick and easy it is to visualize structures, while highlighting important concepts (such as tools that give rise to contextual actions) and providing useful tips (such as exporting representations to PNG image files).

Hopefully you will enjoy the read as much as I have, and discover a great deal from the experience!

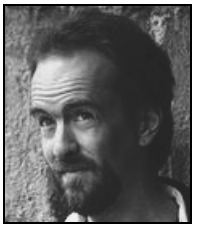
Zane Cahill
Translator

About the Author

HEIKO BÖCK is pursuing his master's degree in informatics at TUM, a technical university in Munich, Germany. He is the author of the book from which this English version was translated. He is a highly respected member of the NetBeans Platform's developer community, and is a member of the NetBeans Dream Team.

About the Technical Reviewers

JAROSLAV TULACH cofounded the NetBeans project, and remains a leading guardian of the NetBeans APIs. He is the author of *Practical API Design: Confessions of a Java Framework Architect* (Apress, 2008) and coauthor of *Rich Client Programming: Plugging into the NetBeans Platform* (Prentice Hall PTR, 2007). He lives in Prague, in the Czech Republic, where he continues to work as a member of the NetBeans team.



GEERTJAN WIELENGA is the technical writer responsible for the documentation relating to the NetBeans APIs. He writes the tutorials and JavaHelp topics relating to this area of the NetBeans project. Like Jaroslav, he coauthored *Rich Client Programming: Plugging into the NetBeans Platform*, and he lives in Prague and works as a member of the NetBeans team.

Acknowledgments

I am very pleased that the original German book is now available in English and thus to a worldwide readership! After long consideration, I almost wanted to reject the concept of a translation project due to lack of time and the enormous expense involved. But then, fortunately, Geertjan Wielenga joined the game, managing to get 11 other industrious translators on board. By joining forces, the book was translated within a single month. What I had considered impossible came true within a few weeks.

For this success, I would like to especially thank Geertjan. Through his hard work, the project was programmed for success from the beginning. In the same way, I would like to express my gratitude to the translators for their passionate cooperation. Many thanks also go to Jim Freeman, the editor of the translation project. Last but not least, I offer many thanks to all participating Apress employees for their great cooperation.

And you, dear reader: I wish you a lot of fun discovering the world of the NetBeans Platform with its numerous features, as well as a lot of success in implementing your projects!

Heiko Böck

Introduction

Over the past several years, rich client desktop platforms have gradually increased in popularity. Leading this trend have been the NetBeans Platform and the Eclipse RCP. The popularization of these desktop platforms has been primarily driven by their related IDEs, which are based on these platforms, providing tools for applications developed on top of their infrastructures. While the Eclipse RCP bases itself, via SWT and JFace, on homegrown idioms and concepts, the NetBeans Platform relies completely on the standard Java APIs, via AWT and Swing, fully integrating the official concepts of the Java Standard Edition.

In the desktop world, rich client platforms are used first and foremost because of the architecture and flexibility they offer to continually growing applications. A significant factor is the increased productivity and flexibility in being able to assemble an application for one purpose and then reassemble it for a different purpose without much extra work, thanks to their flexible modular architecture. Especially for large and professional applications, these concerns are of particular relevance.

It is my opinion that all desktop applications stand to gain from basing themselves on a rich client platform, regardless of their size. The case for this argument can be made by looking, in particular, at the lifecycle management offered by rich client platforms, together with their rich set of APIs, which provide out-of-the-box solutions for the daily challenges faced by desktop application developers. These solutions are tailored specifically to the demands of these kinds of developers, as a result increasing productivity significantly. However, the universal relevance of rich client platforms requires an appropriate handling of the related concepts. At the very least, the developer needs to be comfortable with the main idioms of the platform in question. Only then can the real advantages in increased productivity and improved quality be realized.

The supposed complexity of rich client platform concepts is one of the central reasons why such platforms have, so far anyway, not been adopted as a de facto standard in the development of desktop applications. At the outset, developers often have the impression of standing in the foothills of an overwhelming mountain of new APIs and concepts. However, once developers integrate these APIs and concepts into their mental toolbox, a surprisingly expansive vista of synergies and simplifications is suddenly available, making the learning curve a worthwhile expense.

Consider the most recent enhancements in the Java Platform in relation to desktop applications, such as the improved desktop integration and the performance enhancements, and then examine the plans for the Java Platform in the future. When you do so, you'll notice that the Java Platform is moving in directions that rich client desktop platforms have been exploring from their very beginnings. When I refer to the future, I am referring in particular to the Java Module System (JSR 277), which promises to bring the central concepts of rich client platform development to the Java Platform.

Finally, I'd like to include a note on the NetBeans IDE in relation to the NetBeans Platform. The IDE provides, via its thorough and helpful wizards, effective support for developers getting started with application development on this particular rich client platform. Important for an

easy start is that many of the APIs and concepts you will learn about are directly derived from the Java SE API. Thanks to these factors, you will be able to get started with the NetBeans Platform quite quickly. Reuse of components across different applications will then also rapidly become a possibility.

How This Book Is Structured

This book is aimed at Java developers wanting to create desktop applications on top of the NetBeans Platform. No knowledge of the NetBeans Platform is assumed. The primary goal of this book is the practical explanation of the basic concepts and functionalities of the NetBeans Platform. In the process, you will be introduced to the great support for this kind of development offered by the NetBeans IDE. You will hopefully begin asking yourself why you haven't been developing your desktop applications on top of a platform all along! At the very least, you will learn about the many advantages you could have benefited from in your past Java programming activities.

Firstly, the book discusses the definition of rich clients and rich client platforms. The argument for the general usefulness of these concepts culminates with an examination of the advantages of rich client platforms in general and the NetBeans Platform in particular.

Next, you are introduced to the architecture of the NetBeans Platform. You'll learn how a rich client application is structured, how your application's business logic is integrated into the NetBeans Platform, and how to efficiently use the NetBeans Platform concepts and components. You'll also be shown how to make your applications user- and locale-specific, how to distribute them, and how to update them after distribution.

An important discussion relating to rich client development is that of persistence. This book dives into this topic in some detail, introducing you to the Java Persistence API in combination with Hibernate, as well as with Java DB.

The desktop integration possibilities offered by Java 6 are explained as well. The powerful Visual Library API, which has belonged to the NetBeans Platform since version 6.0, is examined closely, as is the increasingly relevant topic of web services.

This book discusses the similarities and differences between Eclipse RCP and the NetBeans Platform, and walks you through the migration of an existing Eclipse RCP application to the NetBeans Platform.

The individual chapters are structured such that they are as loosely tied to each other as possible. The intent is for you to be able to dive directly into a chapter, without having to be too familiar with the preceding or following parts. I think you will find this approach optimal for the development of rich client applications on top of the NetBeans Platform. To give a practical perspective to each chapter, and to let you use their contents immediately, the explanations in the book are accompanied by small examples, rather than a large overarching application that spans the whole book. At the end of the book, a complete application on the NetBeans Platform is described in some detail, from its starting point to the implementation of business logic, in a tutorial-like format, describing the creation of an MP3 Manager. In this application, you'll integrate the Java Media Framework together with a Java DB database.

All the examples and explanations in this book are based on Java 6, together with the NetBeans Platform 6.5, although Java 5 should in most cases be sufficient, too. You can obtain the Java Development Kit from `http://java.sun.com`, and you can download the NetBeans IDE from `http://netbeans.org`. You can download the examples as complete NetBeans projects from the Source Code section of the Apress web site, at `http://apress.com`.

CHAPTER 1

Introduction

Let's Find Out What This Book Is All About!

This chapter introduces you to the theme of "rich clients." In the process, you will learn what a rich client is and how a rich client platform can help you. In addition, we will briefly touch on the main advantages and characteristics of the NetBeans Platform.

What Is a Rich Client?

In a client server architecture the term "rich client" is used for clients where the data processing occurs mainly on the client side. The client also provides the graphical user interface. Often rich clients are applications that are extendable via plugins and modules. In this way, rich clients are able to solve more than one problem. Rich clients can also potentially solve related problems, as well as those that are completely foreign to their original purpose.

Rich clients are typically developed on top of a framework. A framework offers a basic starting point on top of which the user can assemble logically related parts of the application, which are called modules. Ideally, unrelated solutions (such as those made available by different providers) can work together, so that all the modules appear to have been created as one whole. Software developers and providers can also bundle rich client distributions from distinct modules, with the aim to make these available to specific users.

Above and beyond all that, rich clients have the advantage that they are easy to distribute and update, such as via an automatic online update function within the client itself or through a mechanism that enables the rich client to start over the Internet (for example, via Java Web Start).

Here's an overview of the characteristics of a rich client:

- Flexible and modular application architecture
- Platform independence
- Adaptability to the end user
- Ability to work online as well as offline
- Simplified distribution to the end user
- Simplified updating of the client

What Is a Rich Client Platform?

A rich client platform is an application lifecycle environment, a basis for desktop applications. Most desktop applications have similar features, such as menus, toolbars, status bars, progress visualizations, data displays, customization settings, the saving and loading of user-specific data and configurations, splash screens, About boxes, internationalization, help systems, and so on. For these and other typical client application features, a rich client platform provides a framework with which the features can quickly and simply be put together.

The configurability and extensibility of an application take center stage in a framework of this kind. As a result, you can, for example, declaratively provide the menu entries of an application in a text file, after which the menu will be loaded automatically by the framework. This means that the source code becomes considerably more focused and manageable, and developers are able to concentrate on the actual business needs of the application, while the menu is maximally configurable.

The most important aspect of a rich client platform is its architecture. Applications based on rich client platforms are written in the form of modules, within which logically coherent parts of an application are isolated. A module is described declaratively and automatically loaded by the platform. As a result, there is no explicit binding necessary between the source code and the application. In this way, a relatively loosely coupled relationship is established between independently functioning modules, by means of which the dynamic extensibility of the application and the ability to swap its constituent parts are enormously simplified. In this way it is also very easy to assemble user- or domain-specific applications from individual modules.

A rich client platform also frees the developer from being concerned with tasks that have little to do with the application's business logic. At the end of the development cycle, you achieve a well-deserved and modern application architecture.

Advantages of a Rich Client Platform

Aside from the modularity offered by a rich client architecture, which simultaneously implies a high degree of robustness and end user value, the extensive development support it provides needs to be highlighted as well. These and other advantages of rich client platforms are briefly described here:

- Reduction in development time
- User interface consistency
- Updating
- Platform independence
- Reusability and reliability

We'll look at each in turn.

Reduction in Development Time

A rich client platform provides a multitude of APIs for desktop application development. For example, these can be used by developers to manage windows and menus or support the display of customization options. Through the reusability of many predefined components, developers are able to concentrate very closely on the business logic of the application in question.

User Interface Consistency

Usability of an application is always of crucial concern, in particular when the application is intended to be used by professionals of a particular domain. A rich client platform makes available a framework for the display of user interfaces, while taking particular care of its consistency, accessibility, and usability.

Updating

Using a rich client platform, it becomes possible to quickly and efficiently distribute new or updated modules to end users. As a result, not all the clients of an application need be informed by developers to switch to a new version. Updates can be distributed and installed in the form of modules, so distinct features can be developed and delivered by independently operating teams. The modular architecture of the application ensures that completed modules can be distributed without having to wait for other modules to be finalized.

Platform Independence

Rich client platforms are based on international standards and reusable components. As a result, Java applications based on them can be automatically deployed to multiple different systems, such as Windows or Linux, so long as an implementation of the Java Runtime Environment is available. Since the feature set and the applicability of applications keep changing, it is very important that they are developed in such a way that they are extendable and can be deployed to different target systems. All this is provided by a rich client platform, saving time and money. Applications based on rich client platforms do not require further libraries or components, other than the Java Runtime Environment.

Reusability and Reliability

Rich client platforms make a range of features and modules available, which can be used in the developer's own applications. If the module does not completely match the application's requirements, it is entirely possible to use it as a starting point, while extending it or changing it as needed. Since most platforms also make their source code available, it may also, in some cases, be worth considering changing or extending the platform itself. These factors imply a high degree of reliability and freedom.

Characteristics of the NetBeans Platform

The NetBeans Platform offers, aside from the generic advantages of a rich client platform, numerous frameworks and several further specifics that can be particularly useful to your applications. The important ones, which constitute the main characteristics of the NetBeans Platform, are outlined here:

- User interface framework
- Data editor
- Customization display
- Wizard framework
- Data systems

- Internationalization
- Help system

We'll look at each in turn.

User Interface Framework

Windows, menus, toolbars, and other components are made available by the platform. As a result, you focus on specific actions, which condense your code, making it better and less error-prone. The complete user interface offered by the NetBeans Platform is based 100% on AWT/Swing and can be extended with your own components.

Data Editor

The powerful NetBeans editor within the NetBeans IDE can be used by your own application. The tools and functionality of the editor can quickly and easily be extended and adapted to the purposes of the application.

Customization Display

A display of user- and application-specific settings is needed in every application. The NetBeans Platform makes a framework available, making it extremely simple to integrate your own options dialogs, letting the user save and restore settings in a way that is pleasing to the eye.

Wizard Framework

The NetBeans Platform offers simple tools to create extendable and user-friendly assistants, guiding the user through complex steps in the application.

Data Systems

In terms of the NetBeans Platform, data can be local or available via FTP, CVS, a database, or an XML file. By means of abstraction, data access by one module is transparent to all other modules. Actual data access itself is therefore not a concern, since it is dealt with by the NetBeans Platform's APIs.

Internationalization

The NetBeans Platform provides classes and methods enabling the internationalization of JavaHelp and other resources. You can easily store text constants in properties files. The NetBeans Platform also loads text constants and icons applicable to the current country and language settings.

Help System

By means of the standard JavaHelp system, the NetBeans Platform offers a central system for the integration and display of help topics to the end user. In addition, individual modules can contribute their own topics to the application's help system. On top of all that, the NetBeans Platform lets you provide context-sensitive help as well.

Summary

In this chapter, you learned the difference that a rich client can make. We discussed advantages a rich client brings to the table, including its modular architecture, made possible by a module system unique to rich client platforms. However, a rich client platform offers many other advantages and features. Among these, support for a consistent user interface and the update of applications with new features at runtime. Finally, we examined the most important characteristics of the NetBeans Platform.

CHAPTER 2

Structure of the NetBeans Platform

Let's Find Out What It's Made Of!

To give you an overview of how a rich client application is structured, and to show the relationship between the application that you're creating and the NetBeans Platform, this chapter will discuss the architecture of the NetBeans Platform. You will also be introduced to the independent building blocks of the NetBeans Platform and to the responsibilities that the runtime container handles for you. Finally, the structure of the NetBeans classloader system will be explained, together with the role it plays in applications built atop the NetBeans Platform.

NetBeans Platform Architecture

The size and complexity of modern applications has steadily increased over time. At the same time, professional applications need to be, before anything else, flexible, so that they can quickly and easily be extended. That makes it desirable to divide an application into distinct parts. As a result, each distinct part is a building block making up a modular architecture. The distinct parts must be independent, making available well-defined interfaces that are used by other parts of the same application, with features that other parts can use and extend.

The division of application into modules—that is, as logically interdependent parts—enhances the design of an application enormously. As opposed to a monolithic application, in which every class can make use of code from any other class, the architecture is far more flexible and, more importantly, far simpler to maintain. It is also possible to protect a class from access from the outside world, although such class-level protection is too finely grained to be useful to most applications. It is exactly this central aspect of modern client applications that the NetBeans Platform tackles. Its concepts and structures support the development and conceptualization of flexible and modular applications.

The basic building block of the NetBeans Platform is a *module*. A module is a collection of functionally related classes, together with a description of the interfaces that the module exposes, as well as a description of the other modules that it needs in order to function. The

complete NetBeans Platform, as well as the application built on top of it, is divided into modules. These are loaded by the core of the NetBeans Platform, which is known as the *NetBeans runtime container*. The NetBeans runtime container loads the application's modules dynamically and automatically, after which it is responsible for running the application as well.

In this way, the NetBeans IDE is a very good example of a modular rich client application. The functionality and characteristics of an IDE, such as its Java language support or the code editor, is created in the form of modules on top of the NetBeans Platform (see Figure 2-1). That brings with it the great advantage that the application can be extended by additional modules and that it can be adapted to specific user needs, allowing particular modules that are not used to be deactivated or uninstalled.

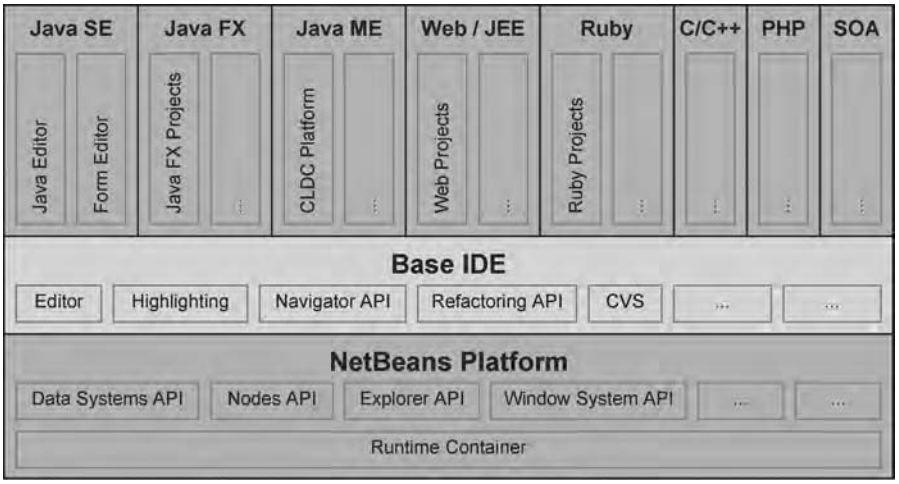


Figure 2-1. *Conceptual structure of the NetBeans IDE*

To enable your applications to attain this level of modularity, the NetBeans Platform on the one hand makes mechanisms and concepts available that enable modules to be extendable by other modules, and on the other hand enables them to communicate with each other without being dependent on each other. In other words, the NetBeans Platform supports a *loose coupling* of modules within an application.

To optimize the encapsulation of code within modules, which is necessary within a modular system, the NetBeans Platform provides its own classloader system. Each module is loaded by its classloader and, in the process, makes a separate independent unit of code available. As a result, a module can explicitly make its packages available, with specific functionality being exposed to other modules. To use functionality from other modules, a module can declare dependencies on other modules. These dependencies are declared in the module's manifest file and resolved by the NetBeans runtime container, ensuring that the application always starts up in a consistent state. More than anything else, this loose coupling plays a role in the declarative concept of the NetBeans Platform. By that we mean that as much as possible is defined in description and configuration files, in order to avoid a hard-wired connection of these concepts with the Java source code. A module is described by its manifest file's data, together with the data specified in related XML files, and therefore does not need to be explicitly added to the NetBeans Platform. Using XML files, the NetBeans Platform knows the modules

that are available to it, as well as their locations and the contracts that need to be satisfied for them to be allowed to be loaded.

The NetBeans Platform itself is formed from a group of core modules (see Figure 2-2), which are needed for starting the application and for defining its user interface. To this end, the NetBeans Platform makes many APIs and service providers available, simplifying the development process considerably. Included in this group (see Figure 2-2) are, for example, the Actions API, which makes available the oft-needed action classes; the powerful Nodes API; and the Options SPI, with whose help your own options dialogs can easily be integrated into the application. Next to these, there are also complete reusable components in the NetBeans Platform, such as the Output window and the Favorites window.

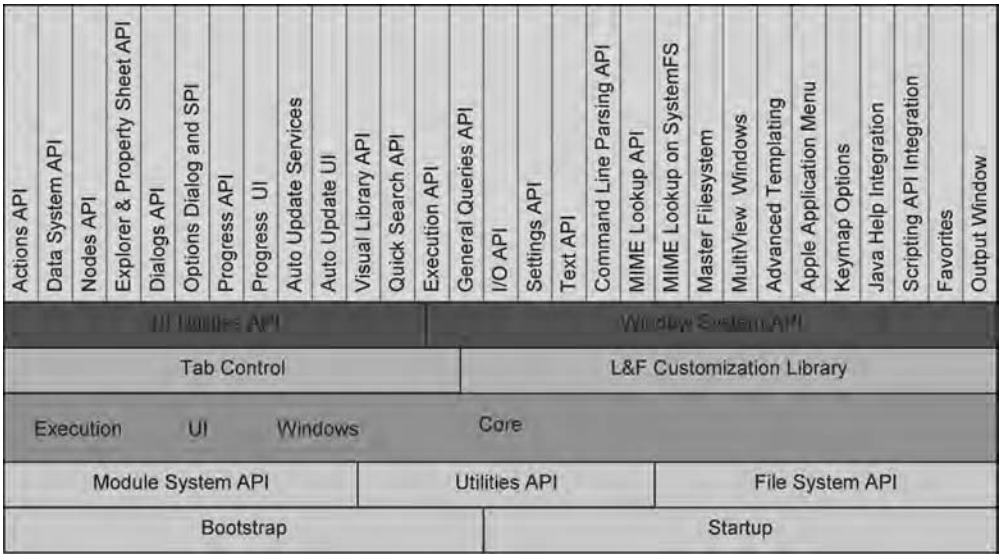


Figure 2-2. *NetBeans Platform architecture*

NetBeans Platform Distribution

Normally you don't need to download a distribution of the NetBeans Platform, since it's already a basic part of the NetBeans IDE, itself a rich client application built on top of the NetBeans Platform. When you develop your application in the NetBeans IDE and then create a distribution for your application, the NetBeans Platform is extracted from the NetBeans IDE distribution you use for development. However, you can also register multiple NetBeans Platforms in the NetBeans IDE. To that end, you can download a separate distribution of the NetBeans Platform from the official site, at `http://platform.netbeans.org`.

Let's now look more closely at the modules that make up a NetBeans Platform distribution:

The modules `org-netbeans-bootstrap`, `org-netbeans-core-startup`, `org-openide-filesystems`, `org-openide-modules`, and `org-openide-util` form the *NetBeans runtime container*. This is the core of the NetBeans Platform and is responsible for the deployment of all the other modules in the application.

The modules `org-netbeans-core`, `org-netbeans-core-execution`, `org-netbeans-core-ui`, and `org-netbeans-core-windows` provide basic underlying functionality for applications based on the NetBeans Platform.

`org-netbeans-core-output2` is a predefined application module that provides a central window for displaying and working with application output messages. More about this module can be read in Chapter 9.

The module `org-netbeans-core-multiview` is a framework for multiview windows, such as those used by the Matisse GUI Builder, and makes an API available for similar views.

The module `org-openide-windows` contains the Window System API, which is probably the API most frequently used by NetBeans Platform applications. In this module you can find foundation classes for the development of application windows and, among others, the `WindowManager`, which gives you access to and information about all the windows available to the application.

A NetBeans Platform application's update functionality is provided by the `org-netbeans-modules-autoupdate-services` module. This module contains all the functionality required for discovery, downloading, and installation of modules into an application. The module `org-netbeans-modules-autoupdate-ui` provides the Plugin Manager, with which the user can choose and control updates to an application.

The `org-netbeans-modules-favorites` module provides a window that lets the user select folders and files—i.e., a filechooser integrated into a NetBeans Platform idiom. The actions added to folders and files via the Data Systems API can also be used in this window.

The module `org-openide-actions` makes a range of important system actions available, such as `Copy`, `Paste`, and `Print`, each of which can be implemented in a context-sensitive way.

`org-openide-loaders` is a very powerful module that contains, among others, the Data Systems API for the integration of data loaders that can be connected to specific types of data.

The Nodes API provided by the `org-openide-nodes` module enables a crucial concept in the NetBeans Platform, that of *nodes*. For example, nodes can be shown in an explorer view, have actions assigned to them, and be supported by property sheets.

The module `org-openide-explorer` provides a framework for the display of explorer views, such as the Projects windows and Files window that are used in the NetBeans IDE.

The module `org-netbeans-modules-editor-mimelookup` provides an API for the discovery of MIME-specific settings, services, and other objects, together with an SPI enabling creation of MIME-specific data providers. The module `org-netbeans-modules-editor-mimelookup-impl` is a special implementation of this SPI, used for the discovery of objects for which the System Filesystem is responsible.

`org-netbeans-modules-javahelp` contains the JavaHelp runtime library, while making an implementation available to the Module System API that enables it to integrate JavaHelp helpsets provided by different modules.

The Master Filesystem module `org-netbeans-modules-masterfs` provides a central wrapper file system to your application on the NetBeans Platform.

The module `org-netbeans-modules-options-api` provides an options window for user customizations and an SPI that enables you to extend the options window very quickly and easily.

The module `org-netbeans-api-progress` lets you control long-running tasks. The module `org-netbeans-modules-progress-ui` offers a visualization feature that enables users to end a task manually.

`org-netbeans-modules-queries` makes the General Queries API available for obtaining information about files that are handled by an application. On top of that is an SPI for creating your own query implementations.

`org-netbeans-modules-sendopts` contains the Command Line Parsing API and SPI, with which you can register your own handlers for dealing with command line arguments.

The module `org-netbeans-modules-settings` provides an API for saving module-specific settings in a user-defined format. To this end, it offers several useful settings formats.

`org-openide-awt` accesses the UI Utilities API, which provides many helper classes for the display of user interface elements in NetBeans Platform applications.

The module `org-openide-dialogs` provides an API for displaying standard and user-specific dialogs. In addition, this module also contains the Wizard framework.

`org-openide-execution` makes available an API for executing long-running tasks asynchronously.

`org-openide-io` provides an API and SPI for the display of input and output coming from data within the application. The module also makes a standard implementation available that enables the application to write to its output window.

The Text API in the module `org-openide-text` offers an extension to the javax.swing.text API.

The modules `org-netbeans-swing-plaf` and `org-netbeans-swing-tabcontrol` are responsible for handling the look and feel and the display of tabs, while the module `org-jdesktop-layout` is a wrapper for the Swing Layout Extensions library.

The `org-netbeans-api-visual` module makes available the Visual Library API, for modeling and displaying visual representations of data.

The module `org-netbeans-spi-quicksearch` contains the Quick Search API and SPI, which are new in NetBeans Platform 6.5. It provides the infrastructure to implement search providers for, e.g., menu entries, actions, or files. A central search field makes the search accessible to the user.

Additionally, it is possible to add modules to your application from the NetBeans IDE's distribution.

NetBeans Runtime Container

The basis of the NetBeans Platform and its modular architecture is the NetBeans runtime container. It consists of the following five modules:

`Bootstrap`: This module is executed before any other. It carries out all registered command handlers and prepares a boot classloader, which loads the `Startup` module.

`Startup`: This module deploys the application, at which point it initializes the module system and the file system.

`Module System`: This module is responsible for the definition of modules, as well as their settings and dependencies.

`File System`: This module makes a virtual data system available, with platform-independent access. Primarily it is used to load module resources into an application.

`Utilities`: This module provides basic components, such as those required for intermodular communication.

Figure 2-3 shows the dependencies between these five basic modules.

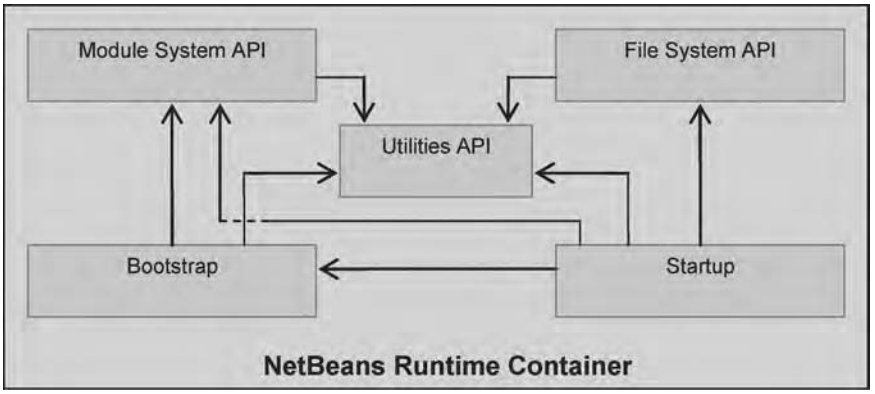


Figure 2-3. *NetBeans runtime container*

The runtime container is a minimal subset of modules that NetBeans Platform applications require. Without other modules or settings being required, an application can be deployed containing these five modules. Directly after deployment it will shut down, since no further tasks have been defined. When the runtime container starts, it finds all available modules and builds from them an internal registry. A module is normally only loaded once it is needed. To that end, it is registered at startup. A module also has the ability to execute tasks at the time that it is loaded by the runtime container. That takes place by means of a module installer, about which more is discussed in Chapter 3. The runtime container also enables dynamic loading, unloading, installing, and uninstalling of modules, all of which occur at runtime. This functionality is for updating an application by the user (via the update feature) or for deactivating unnecessary modules in an application.

For a complete understanding of the deployment of a rich client application, it is also important to mention that the `Bootstrap` module (the first module executed) is started by a platform-specific launcher. The launcher is also responsible for identifying the Java Runtime Environment, which is required for starting the application. The launcher is part of the NetBeans Platform and is platform-specific, so that, for example, on Windows systems it is an `.exe` file.

NetBeans Classloader System

The NetBeans classloader system is an essential part of the NetBeans runtime container and a continuation of the encapsulation of modules and structuring of a modular architecture. This system consists of three different types of classloaders. These are the module classloader, the system classloader, and the original classloader. Most classes are loaded by the module classloader. Only in certain cases, such as when resources must be accessed outside a module, is the system classloader used. The original classloader loads resources from the classpath of the application launcher. The module classloader and the system classloader are multi-parent classloaders, having an infinite number of classloaders as their parents. The relationships between classloader types are displayed in Figure 2-4.

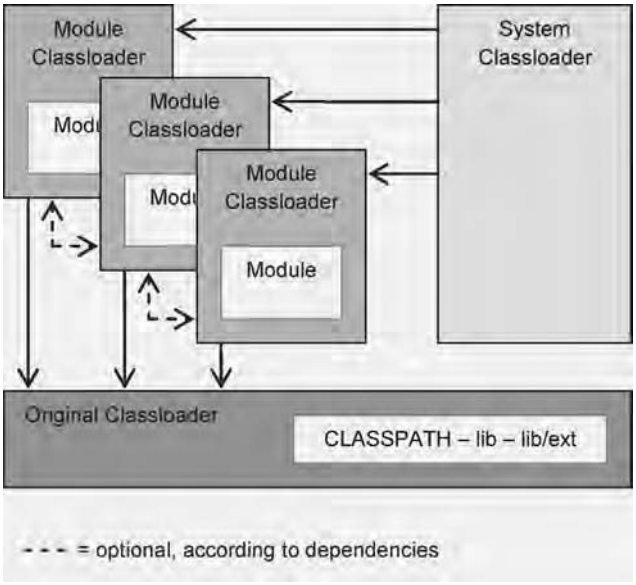


Figure 2-4. *NetBeans classloader system*

Module Classloader

For every module registered in the Module System, an instance of the module classloader is made available, by means of which every module obtains its own namespace. Primarily, this classloader loads classes from the module's JAR archive, by which it may load from multiple archives, as often happens with library wrapper modules. (You will learn more about this in Chapter 3.)

The original classloader is implicitly a parent classloader of every module classloader, and is the first on the parent's list. Further parents are those of related modules, on which dependencies have been set. How dependencies are set is defined in Chapter 3.

This multi-parent module classloader enables classes to be loaded from other modules, while avoiding namespace conflicts. The loading of classes is delegated to the parent classloader, rather than the modules themselves. In addition to the module JAR archive, this classloader is responsible for loading the locale extension archives (see Chapter 10) from the subdirectory locale, as well as the patch archives under the subdirectory `patches`, if these are available.

System Classloader

The system classloader is, by default, (as is the case with the module classloader) a multi-parent classloader. It owns all the instantiated module classloaders as its parents. As a result, it is theoretically possible to load everything provided by a module with this classloader. Even so, obeying the strictures of encapsulation, this approach should only be followed when absolutely necessary.

Access to the system classloader can be obtained in one of two different ways: via *Lookup*, about which you will hear much more later, as well as the context classloader of the current thread. This is the default (insofar as you have not explicitly set other context classloaders) of the system classloader.

```
ClassLoader cl = Lookup.getDefault().lookup(ClassLoader.class);
```

or

```
ClassLoader cl = Thread.currentThread().getContextClassLoader();
```

Original Classloader

The original (application) classloader is used by the launcher of the application. It loads classes and other resources on the original classpath and, after that, from the `lib` directories and their `ext` subdirectories. If a JAR archive is not recognized as a module (i.e., the manifest entries are invalid), it is not added to the module system. These resources are always found first: if the same resource is found here as in the module JAR archive, those found in the module are ignored. This arrangement is important to the branding of modules, as well as to the preparation of multiple language distributions of a particular module.

As before, this classloader is not used for loading related resources. It is much more likely to be used for resources that are needed in the early start phase of an application, such as for the classes required for setting the look and feel.

Summary

This chapter examined the structure of the NetBeans Platform. We began by looking at its architecture, the core of which is provided by the runtime container. The runtime container provides the execution environment of applications created atop the NetBeans Platform and also provides an infrastructure for modular applications. The NetBeans classloader system, which ensures the encapsulation of modules, was introduced and explained. Aside from the runtime container, many modules form parts of the NetBeans Platform, and we looked briefly at each of these. Finally, we noted that the NetBeans IDE is itself a rich client application consisting of modules that are reusable in your own applications.

CHAPTER 3

The Module System

Let's Understand the Basic Building Blocks!

This chapter focuses on the NetBeans module system, which is the central component of the runtime container and is responsible for loading and managing all the modules in the application. You'll learn how a module is structured, how it connects with other modules, how it integrates into the NetBeans Platform, and how its lifecycle is traced and influenced.

Overview

The NetBeans module system is responsible for managing all modules in the application. It is also responsible for tasks such as creating the classloader, the loading of modules, and their activation and deactivation. The concept of the NetBeans module system is based, as far as possible, on standard Java technologies. The basic idea of the module format originates in the Java Extension Mechanism. The fundamental ideas of the Package Versioning Specification are used to describe and manage dependencies between application modules and dependencies of system modules.

Basic properties, such as the description of a module and its dependencies on other modules, are described in a *manifest* file. This file has the standard manifest format with additional NetBeans-specific attributes. The Java Activation Framework as well as JDK internal functions (such as the support of executable JAR files) are used as a design model for the module specification. Besides attributes in the manifest file, most modules do not need special installation code, as they are added to the NetBeans Platform declaratively. The XML file `layer.xml` provides application-specific information and defines the integration of a module into the NetBeans Platform. Everything that a module adds to the NetBeans Platform is specified in this file, ranging from actions to menu items to services.

Module Structure

A module is a simple JAR file, normally consisting of the following parts (see also Figure 3-1):

- Manifest file (`manifest.mf`)
- Layer file (`layer.xml`)
- Class files
- Resources like icons, properties bundles, helpsets, etc.

Only the manifest file is obligatory, since it identifies the module. All other content depends on a module's purpose. In most cases, if the module is only used as a library, the layer file is superfluous.

Figure 3-1. *NetBeans module*

An XML config file (e.g., `com-galileo-netbeans-module.xml`) is needed by each module, located outside the JAR file. This is the first file read—i.e., it announces the module to the platform.

Module Types

All modules are declared in the module system by an XML configuration file, located in the cluster folder `config/Modules`, outside the module file. This folder is read on application startup by the module system and the modules are loaded according to this information. The content of the configuration file describes the module name, version, and location, as well as whether or not and how the module is loaded. See the document structure in Listing 3-1.

Listing 3-1. *Module configuration file: com-galileo-netbeans-module.xml*

```
<module name="com.galileo.netbeans.module">
   <param name="autoload">false</param>
   <param name="eager">false</param>
   <param name="enabled">true</param>
   <param name="jar">modules/com-galileo-netbeans-module.jar</param>
   <param name="reloadable">false</param>
   <param name="specversion">1.0</param>
</module>
```

The `enabled` attribute defines if the module is loaded and the manner in which it is provided to the NetBeans Platform Application. There are three ways to determine at which point a module should be loaded. If the value of both the attributes `autoload` and `eager` is `false`, the module is type `regular`. If one of these values is `true`, the module type is `autoload` or `eager`. The module type is defined in the API Versioning section of the module properties dialog (see Figure 3-7). By default, regular mode is used.

Regular

This is the common type of application modules. They are loaded on application start. The application loading time is extended by the time of module initialization. Therefore, it is recommended to keep the module initialization very short. Normally it is not necessary to run anything during module loading, as many tasks can be defined declaratively.

Autoload

These modules are loaded only when another module requires them. Autoload modules correspond to the principle of lazy-loading. This mode is usually used for those modules acting as libraries.

Eager

Eager modules are only loaded when all dependencies are met. This is another option to minimize starting time. For example, if module X depends on the modules A and B, which are not available, it makes no sense to load module X.

Module Manifest

Each module running within the NetBeans Platform has a manifest file. This file is a textual description of the module and its environment. When loading a module, the manifest file is the first file read by the module system. A NetBeans module is recognized if the manifest file contains the `OpenIDE-Module` attribute. This is the only mandatory attribute. Its value can be any identifier (typically the code name base of the module is used—e.g., `com.galileo.netbeans.module`); therefore, conflicts cannot occur between modules, even if created by various developers. This identifier is used to distinguish a non-ambiguous module, needed for upgrades or dependency definitions.

Attributes

In the following sections, all attributes of a module manifest are described briefly, each with a short example.

Description

Consider the following frequently used attributes by which a module can be textually described:

`OpenIDE-Module`: This attribute defines a unique name for the module used for recognition by the module system. The specification of this attribute is mandatory.

```
OpenIDE-Module: com.galileo.netbeans.module
```

`OpenIDE-Module-Name`: This defines a displayable name of the module, also displayed in the Plugin Manager.

```
OpenIDE-Module-Name: My First Module
```

`OpenIDE-Module-Short-Description`: This represents a short functionality description provided by the module.

```
OpenIDE-Module-Short-Description: This is a short description of my first module
```

`OpenIDE-Module-Long-Description`: This attribute defines a long description of the module-provided functionality. The text is displayed in the Plugin Manager. Setting this attribute is recommended, as it informs the user about features of the module.

```
OpenIDE-Module-Long-Description:
   Here you can put a longer description with more than one
   sentence. You can explain the capability of your module.
```

`OpenIDE-Module-Display-Category`: Modules are summarized into a virtual group with this attribute and thus presented to the user as a functional unit.

```
OpenIDE-Module-Display-Category: My Modules
```

`OpenIDE-Module-Install`: To run actions at certain times in the module lifecycle, this attribute sets a module installer class (see the "Lifecycle" section later in the chapter).

```
OpenIDE-Module-Install: com/galileo/netbeans/module/ModuleLifecycle.class
```

`OpenIDE-Module-Layer`: This is one of the most important attributes. With it, the path is specified to the layer file (see the "Module Layer" section later in the chapter) describing the module integration into the platform.

```
OpenIDE-Module-Layer: com/galileo/netbeans/module/resources/layer.xml
```

`OpenIDE-Module-Public-Packages`: To support encapsulation, access to classes from another module is denied by default. This attribute is used to set visible public packages and allow other modules to use these classes. It is especially essential with libraries.

```
OpenIDE-Module-Public-Packages:
   com.galileo.netbeans.module.actions.*,
   com.galileo.netbeans.module.util.*
```

`OpenIDE-Module-Friends`: If only certain modules are allowed access to these packages, which are declared as `public`, then these may be stated here.

```
OpenIDE-Module-Friends:
   com.galileo.netbeans.module2,
   com.galileo.netbeans.module3
```

`OpenIDE-Module-Localizing-Bundle`: Here, a properties file is defined, which is used as a localizing bundle (see Chapter 8).

```
OpenIDE-Module-Localizing-Bundle:
   com/galileo/netbeans/module/resource/Bundle.properties
```

Versioning and Dependencies

The following attributes are used to define differing versions and dependencies. Descriptions and use of these attributes are detailed in the "Versioning and Dependencies" section later in the chapter.

`OpenIDE-Module-Module-Dependencies`: Dependencies between modules are defined with this attribute. The least-needed module version can also be specified.

```
OpenIDE-Module-Module-Dependencies:
   org.openide.util > 6.8.1,
   org.openide.windows > 6.5.1
```

`OpenIDE-Module-Package-Dependencies`: A module may also depend on a specific package. Such dependencies are defined with this attribute.

```
OpenIDE-Module-Package-Dependencies:
   com.galileo.netbeans.module2.gui > 1.2
```

`OpenIDE-Module-Java-Dependencies`: If a module requires a specific Java version, it can be set with this attribute.

```
OpenIDE-Module-Java-Dependencies: Java > 1.5
```

`OpenIDE-Module-Specification-Version`: This attribute indicates the specification version of the module. It is usually written in the Dewey decimal format.

```
OpenIDE-Module-Specification-Version: 1.2.1
```

`OpenIDE-Module-Implementation-Version`: This attribute sets the module implementation version, usually by a timestamp. This number changes with every change of the module.

```
OpenIDE-Module-Implementation-Version: 200701190920
```

`OpenIDE-Module-Build-Version`: This attribute has only an optional character and is ignored by the module system. Typically, hereby, a date stamp is given.

```
OpenIDE-Module-Build-Version: 20070305
```

`OpenIDE-Module-Module-Dependency-Message`: Here, text is set, which is displayed if a module dependency cannot be resolved. In some cases it's 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 or why none are needed.

```
OpenIDE-Module-Module-Dependency-Message:
   The module dependency is broken. Please go to the following
   URL and download the module.
```

`OpenIDE-Module-Package-Dependency-Message`: The message defined by this attribute is displayed if a necessary reference to a package fails.

```
OpenIDE-Module-Package-Dependency-Message:
   The package dependency is broken. The reason could be...
```

`OpenIDE-Module-Deprecated`: Use this to mark a module as deprecated. A warning is logged if the user tries to load the module into the platform.

```
OpenIDE-Module-Deprecated: true
```

`OpenIDE-Module-Deprecation-Message`: Use this optional attribute 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`.

```
OpenIDE-Module-Deprecation-Message: Module 1 is deprecated, use Module 3 instead
```

Services and Interfaces

The following attributes are used to define dependencies on implementations and certain service provider interfaces. Further information on this topic can be found in Chapter 6.

`OpenIDE-Module-Provides`: Use this attribute to declare a service interface to which this module furnishes a service provider.

```
OpenIDE-Module-Provides: com.galileo.netbeans.spi.ServiceInterface
```

`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.

```
OpenIDE-Module-Requires: org.openide.windows.IOProvider
```

`OpenIDE-Module-Needs`: This attribute is an understated version of the `Require` attribute and does not need any specific order of modules. This may be useful to API modules, which require specific implementation.

```
OpenIDE-Module-Needs: org.openide.windows.IOProvider
```

`OpenIDE-Module-Recommends`: Using this attribute, you can 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-Recommends: java.sql.Driver
```

`OpenIDE-Module-Requires-Message`: Like the two previous attributes, this defines a message displayed if a required token is not found.

```
OpenIDE-Module-Requires-Message:
   The required service provider is not available. For more information go to...
```

PLATFORM-DEPENDENT MODULES

The manifest attribute `OpenIDE-Module-Requires` allows definition of modules that depend on a specific operating system. This attribute 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`
- `org.openide.modules.os.Solaris`

The module system ensures that only the tokens to the operating systems already in use are available. For example, to provide a module that automatically loads on Windows systems but automatically deactivates on all others, set the module type to `Eager` and add the following line to the manifest file:

```
OpenIDE-Module-Requires: org.openide.modules.os.Windows
```

Visibility

With the following attributes, the visibility of modules within the Plugin Manager is controlled. In this way, modules can be displayed clearly and plainly to the end user.

`AutoUpdate-Show-In-Client`: This attribute can be set to `true` or `false`. It determines whether a module is displayed in the Plugin Manager or not.

```
AutoUpdate-Show-In-Client: true
```

`AutoUpdate-Essential-Module`: This attribute can be set to `true` or `false`. `true` means that this module is essential to the application so that it cannot be deactivated or uninstalled.

```
AutoUpdate-Essential-Module: true
```

In conjunction with these attributes, the handling of *kit modules* is introduced in NetBeans Platform 6.5. Each module visible in the Plugin Manager (`AutoUpdate-Show-In-Client: true`) is handled as a kit module. All modules on which the kit module defines a dependency are handled in the same way, 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. You can 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 attribute `AutoUpdate-Show-In-Client` to `false`.

Example

Listing 3-2 shows a manifest file with some typical attributes.

Listing 3-2. *Manifest file example*

```
OpenIDE-Module: com.galileo.netbeans.module
OpenIDE-Module-Public-Packages: -
OpenIDE-Module-Module-Dependencies:
   com.galileo.netbeans.module2 > 1.0,
   org.jdesktop.layout/1 > 1.4,
   org.netbeans.core/2 = 200610171010,
   org.openide.actions > 6.5.1,
   org.openide.awt > 6.9.0,
OpenIDE-Module-Java-Dependencies: Java > 1.5
OpenIDE-Module-Implementation-Version: 200701100122
OpenIDE-Module-Specification-Version: 1.3
OpenIDE-Module-Install: com/galileo/netbeans/module/Install.class
OpenIDE-Module-Layer: com/galileo/netbeans/module/layer.xml
OpenIDE-Module-Localizing-Bundle: com/galileo/netbeans/module/Bundle.properties
OpenIDE-Module-Requires:
   org.openide.windows.IOProvider,
   org.openide.modules.ModuleFormat1
```

Module Layer

In addition to the manifest file, with which the interfaces and the environment of a module are described, there is a *layer* file. This is the central configuration file, in which virtually everything is defined that a module adds to the NetBeans Platform. Partly, it can be seen as the interface between the module and the NetBeans Platform, describing declaratively the integration of the module into the NetBeans Platform.

The existence of the layer file is set in the manifest file with the attribute `OpenIDE-Module-Layer`. This attribute defines the path to the layer file, usually using the file name `layer.xml`.

```
OpenIDE-Module-Layer: com/galileo/netbeans/module/layer.xml
```

The file format is a hierarchical file system containing directories, files, and attributes. During application start, all existing layer files are summarized to a virtual file system (see Figure 3-2). This is the *System Filesystem*, which is the runtime configuration of the NetBeans Platform.

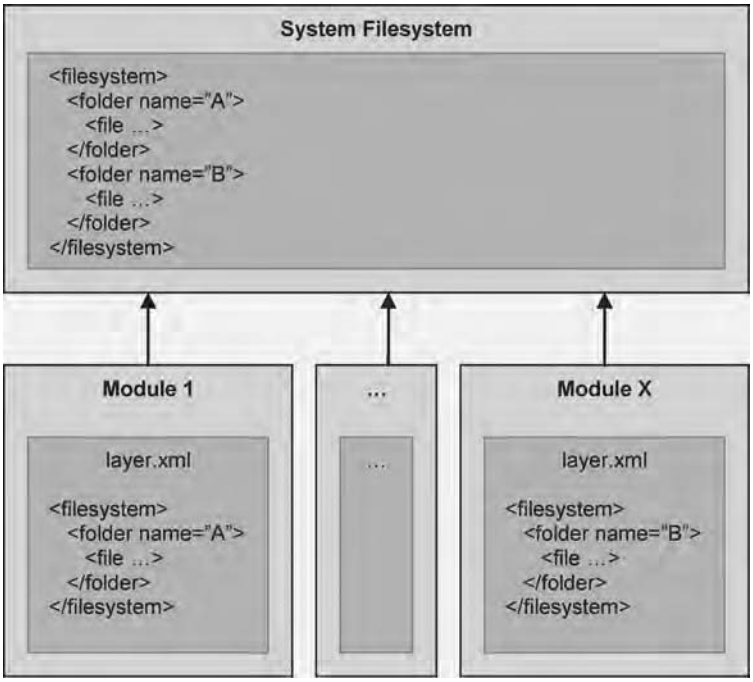


Figure 3-2. *The System Filesystem*

This layer file contains certain default folders. They are defined by different modules, which are *extension points*. For example, the default folder `Menu` looks like Listing 3-3.

Listing 3-3. *Default folder of the layer file*

```
<folder name="Menu">
  <folder name="Edit">
    <file name="MyAction.shadow">
      <attr name="originalFile"
          stringvalue="Actions/Edit/com-galileo-netbeans-module-MyAction.instance"/>
    </file>
  </folder>
</folder>
```

In this example, the action class `MyAction` is added to the Edit menu. Don't worry about exact syntax at this point; it is explained in the context of respective standard folders in later chapters. First of all, we elaborate the basic structure of the layer file. In addition, the NetBeans Platform provides practical features for working with the layer file. That's shown in the following chapters, as our first module is created. An index with important extension points is also found in this book's Appendix.

In this way, every module is able to add new menu items or create new toolbars. As each layer file of a module is merged to the System Filesystem, the entire menu bar content is

assembled. The window system responsible for generation of the menu bar only has to read the `Menu` folder from the System Filesystem to gain the content of the entire menu bar.

This System Filesystem also contributes significantly to the fact that modules can be added or removed at runtime. Filesystem listeners can be registered on this system. For example, this is done by the window system. If any changes occur while a module is loaded, the window system or menu bar notice this and are able to update contents.

Order of Entries

The order in which the entries of the layer file are read (and hence shown in the menu bar) is defined by a `position` attribute, as shown in Listing 3-4.

Listing 3-4. *Determining the order of layer file entries*

```
<filesystem>
  <folder name="Menu">
    <folder name="Edit">
      <file name="CopyAction.shadow">
        <attr name="originalFile"
                stringvalue="Actions/Edit/org-openide-actions-CopyAction.instance"/>
        <attr name="position" intvalue="10"/>
      </file>
      <file name="CutAction.shadow">
        <attr name="originalFile"
                stringvalue="Actions/Edit/org-openide-actions-CutAction.instance"/>
        <attr name="position" intvalue="20"/>
      </file>
    </folder>
  </folder>
</filesystem>
```

Thus, the copy action is shown before the cut action. If necessary, you can also use this attribute to define the order of the folder elements. In practice, positions with greater distance are chosen. This facilitates the subsequent insertion of additional entries. Should the same position be assigned twice, a warning message is logged while running the application.

In order to easily position the layer content, the NetBeans IDE offers a *layer tree* in the Projects window, in which all entries of the layer files are shown. There, their order is defined by drag-and-drop. The respective entries in the layer file are then handled by the IDE.

After we create our first module, the "Creating Modules" section of this chapter shows where to find the layer tree. You determine the order of an action while creating actions with a wizard (see Chapter 4). The respective attributes are then created by the wizard.

Should positions of entries in the layer tree be changed, some entries will be added into the layer file. Those files overwrite the default positions of the entries affected by the change. The position of an entry (also that of entries of a NetBeans Platform module) is overwritten as follows:

```
<attr name="Menu/Edit/CopyAction.shadow/position" intvalue="15"/>
```

Use the complete file path of the affected entry before the attribute name position.

Instance Files

Files of the type `.instance` in the System Filesystem describe objects of which instances can be created. The filename typically describes the full class name of a Java object (e.g., `com-galileo-netbeans-module-MyAction.instance`), which by a default constructor or static method creates an instance. An instance is created by using the File Systems and Data Systems APIs, as follows:

```
public static Object getInstance(String name) {
   FileSystem f = Repository.getDefault().getDefaultFileSystem();
   FileObject o = f.getRoot().getFileObject(name);
   DataObject d = DataObject.find(o);
   InstanceCookie c = d.getCookie(InstanceCookie.class);
   return c.instanceCreate();
}
```

If seeking a more convenient name for an instance, the full class name can be defined by using the attribute `instanceClass`. Thereby much shorter names can be used:

```
<file name="MyWindow.instance">
   <attr name="instanceClass" stringvalue="com.galileo.netbeans.module.MyWindow"/>
</file>
```

In classes not having a parameterless default constructor, create the instance via a static method defined by the attribute `instanceCreate`:

```
<file name="MyWindow.instance">
   <attr name="instanceCreate"
         methodvalue="com.galileo.netbeans.module.MyWindow.getDefault"/>
</file>
```

In doing so, the `FileObject` of the entry is passed to the `getDefault()` method, if declared so in the factory method signature. With this `FileObject` you read self-defined attributes. Assume you want to define the path of an icon or any other resource in the layer file as an attribute:

```
<file name="MyWindow.instance">
   <attr name="instanceCreate"
         methodvalue="com.galileo.netbeans.module.MyWindow.getDefault"/>
   <attr name="icon" urlvalue="nbres:/com/galileo/icon.gif"/>
</file>
```

The method `getDefault()`, creating an instance of the `MyWindow` class, looks as follows:

```
public static MyWindow getDefault(FileObject obj) {
   URL url = (URL) obj.getAttribute("icon");
   ...
   return(new MyWindow(...));
}
```

Notice that we specified the path with a `urlvalue` attribute type. Therefore, a `URL` instance is delivered directly. In addition to the already-known attribute types `stringvalue`, `methodvalue`, and `urlvalue`, there are several others. They are accessed in the Filesystem DTD (`http://netbeans.org/dtds/filesystem-1_2.dtd`).

One or more instances of a certain type can also be generated by Lookup, rather than via an `InstanceCookie`, as shown previously. Thereby we create a lookup for a particular folder of the System Filesystem. By using the `lookup()` or `lookupAll()` method, one or more instances (if several have been defined) can be delivered.

```
Lookup lkp = Lookups.forPath("MyInstanceFolder");
Collection<? extends MyClass> c = lkp.lookupAll(MyClass.class);
```

Such a lookup is used in Chapter 5 to extend the context menu of a `TopComponent` with your own actions defined in the layer file.

The basic class of the interface can be user-defined by the `instanceOf` attribute in the layer file. This allows a more efficient working of Lookup and avoids Lookup having to initiate each object in order to determine from which base class the class will inherit, or which interface it implements. Lookup is able to create directly only instances of the desired object type.

If the class `MyAction` from the prior entry implements, for example, the `Action` interface, we complete the entry as follows:

```
<file name="com-galileo-netbeans-module-MyAction.instance">
   <attr name="instanceOf" stringvalue="javax.swing.Action"/>
</file>
```

Shadow Files

`.shadow` files are a kind of link or reference to an `.instance` file. They are used mainly when singleton instances of objects, as with actions, are used. These are defined by an `.instance` file in the `Actions` folder. An entry in the `Menu` or `Toolbars` folder then refers to the action by using the `.shadow` file (see Listing 3-5). A `.shadow` file refers to files in the System Filesystem as well as to files on disk. In this way, the `Favorites` module stores its entries. The path to the `.instance` file is specified by the attribute `originalFile`.

Listing 3-5. *Connecting a .shadow file with an .instance file*

```
<folder name="Actions">
  <folder name="Window">
    <file name="com-galileo-netbeans-module-MyAction.instance"/>
  </folder>
</folder>
<folder name="Menu">
  <folder name="Window">
    <file name="MyAction.shadow">
      <attr name="originalFile"
        stringvalue="Actions/Window/com-galileo-netbeans-module-MyAction.instance"/>
    </file>
  </folder>
</folder>
```

Settings Files

`.settings` files are an extended version of `.instance` files in the layer file. Information on the type of class and how an instance is created from this class—i.e., information on what these attributes can determine in an `.instance` file—is defined in a separate XML file. The main

difference to an `.instance` file is that the complete class hierarchy—i.e., all superclasses and also all implemented interfaces—can be specified in the separate XML file.

These `.settings` entries are used, for example, with `TopComponents` (see Chapter 5). The associated XML file looks like Listing 3-6.

Listing 3-6. *Type informationen for a .settings file*

```
<!DOCTYPE settings PUBLIC
   "-//NetBeans//DTD Session settings 1.0//EN"
   "http://www.netbeans.org/dtds/sessionsettings-1_0.dtd">
<settings version="1.0">
 <module name="com.galileo.netbeans.module" spec="1.0"/>
 <instanceof class="javax.swing.JComponent"/>
 <instanceof class="org.openide.windows.TopComponent"/>
 <instanceof class="com.galileo.netbeans.module.MyTopComponent"/>
 <instance class="com.galileo.netbeans.module.MyTopComponent" method="getDefault"/>
</settings>
```

The layer file refers to this file by the `url` attribute, specifying the path relative to the XML file:

```
<folder name="Windows2">
   <folder name="Components">
      <file name="MyTopComponent.settings" url="MyTopComponentSettings.xml"/>
   </folder>
</folder>
```

Creating and Using Your Own Contents

Your module may use folders, files, and attributes from the layer file in order to provide extension points to other modules. The readout of entries is accomplished by accessing the System Filesystem:

```
Repository.getDefault().getDefaultFileSystem();
```

This call returns a `FileSystem` object. Its properties and functionality are described in detail in Chapter 7. The "Window: TopComponent" section in Chapter 5 shows you how to define your own entries in the layer file; it also shows how they can be read and, thereby, how they can provide an extension point for other modules in a way that makes them available to additional modules.

Creating Modules

Following the sections on structure and content of modules, we will now create our first module. A good introduction to module development is also offered by the sample applications already integrated in the NetBeans IDE. Here, we will simply design a single module.

First, create a NetBeans Platform Application or a Module Suite. Both are containers for modules. The NetBeans Platform Application project type creates a standalone rich client application, starting from NetBeans Platform modules only, whereas a Module Suite creates a set of related modules, starting from all the modules in the NetBeans IDE. Note, however, that

you can also create a standalone application from a Module Suite. To that end, you need simply use the Project Properties dialog of the Module Suite, accessed via its context menu, and use the Build tab to select the Create Standalone Application option. Choose Exclude when asked whether IDE-specific modules should be excluded. That way, only the modules that are part of the NetBeans Platform remain. View them in the Libraries tab. As you continue creating the application, only those modules are available that are set as needed by the application. However, in both cases, you are able to add modules that are not part of the NetBeans Platform. Do this via the Libraries tab.

The NetBeans IDE provides a wizard to create these projects. Start the NetBeans IDE and select File ➤ New Project. The dialog shows different project categories (see Figure 3-3). Select NetBeans Modules. Now select the project type NetBeans Platform Application on the right side.

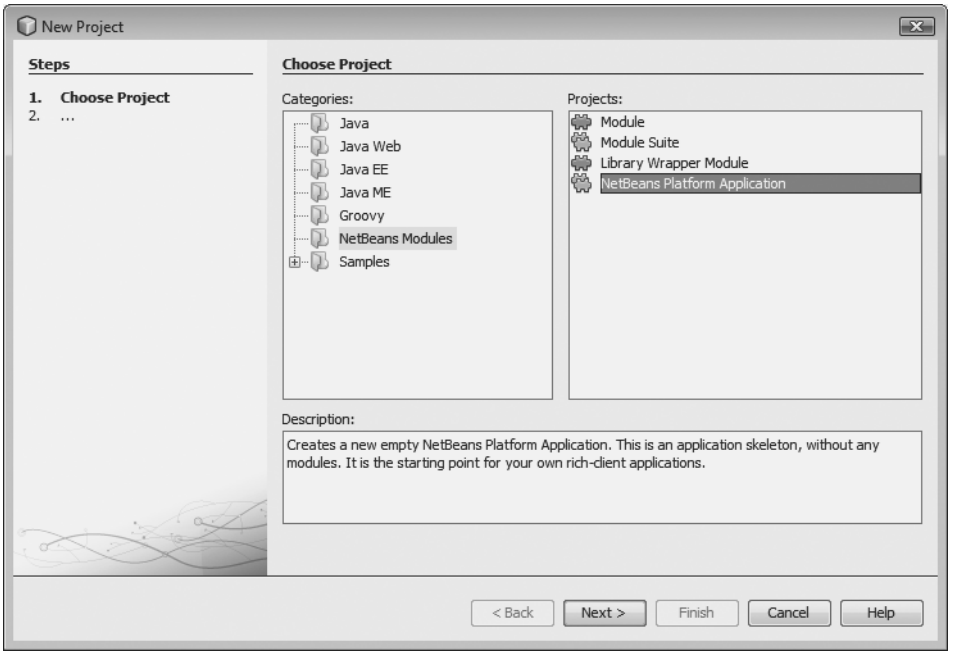


Figure 3-3. *Creating a new NetBeans Platform Application project*

On the next page, name the project, such as My Application, and chose the location where the project is to be saved. The remaining fields can be left blank. Click the Finish button to create the application project.

Now the first module can be created: another wizard is available for this task. Open the File ➤ New Project menu. Choose the category NetBeans Modules, and then the project type Module on the right side. Click the Next button to go to the next page, for naming the project (see Figure 3-4). Enter here, for example, My Module, and then select the option Add to Module Suite, and select the previously created NetBeans Platform Application or Module Suite from the list. On the last page, define the code name base and a module display name. Default values for the localizing bundle and XML layer file can be kept. Then click the Finish button and let the wizard generate the module.

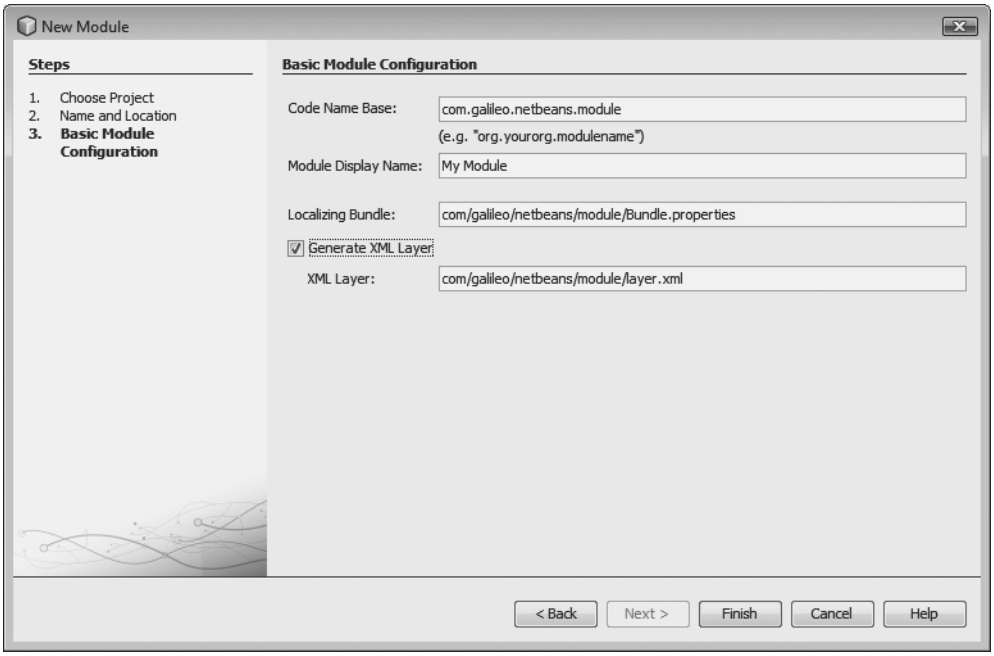


Figure 3-4. *Configuration of a new module*

Looking at the module in the Projects window, you see the folder `Source Packages` (see Figure 3-5). At the moment, this folder contains only the files `Bundle.properties` and `layer.xml`. The file `Bundle.properties` only provides a localizing bundle for the information registered in the manifest file.

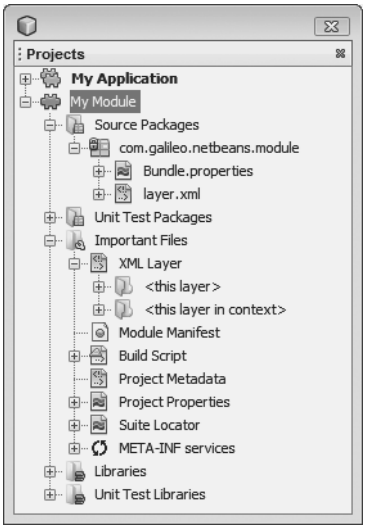


Figure 3-5. *The module in the Projects window*

A special tree structure is presented for layer files within the `Important Files` node. The tree structure gives two different views: first, the folder `<this layer>`, where you view the entries provided by your own layer file; and second, the folder `<this layer in context>`, where you find the entries of the layer files of all the modules that belong to your application. These views represent those parts of the System Filesystem available to the NetBeans Platform at runtime.

In these views, highlighted folders include contributions by the current module. This way, you can see at a glance the most important folders, and you can easily add, delete, or move new entries. The manifest file, created by the wizard that creates the module, is also found in the `Important Files` node.

Without further ado, you can run your module as part of your new rich client application. To start the application, simply choose the Run ➤ Run Main Project (F6) menu item. Your application looks like the one in Figure 3-6.

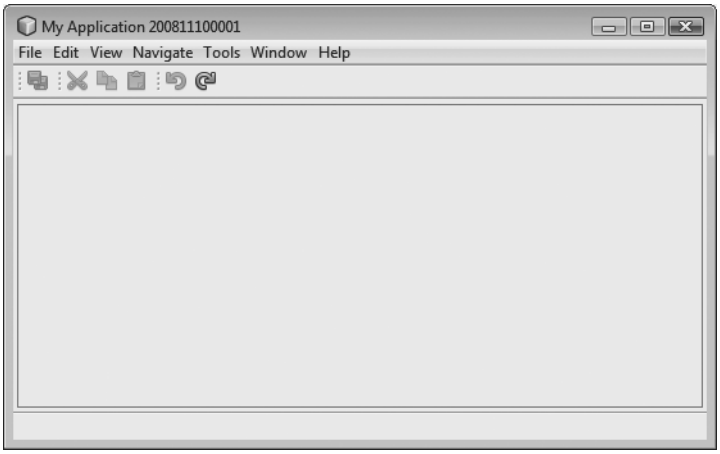


Figure 3-6. *Your first rich client application*

In these very few steps, we now have the basis of our rich client application. In the following chapters, the applications are enriched with new features by adding new modules that contribute windows, menu items, and other business logic.

Versioning and Dependencies

To ensure that a modular system remains consistent and maintainable, it is crucial that the modules within the system prescribe the modules they need to use. To that end, the NetBeans Platform allows definition of dependencies on other modules. Only by defining a dependency can one module access the code from another module. Dependencies are set in the manifest file of a module. That information is then read by the module system when the module is loaded.

Versioning

To guarantee compatibility between dependencies, you must define versions. In that regard, there is the *major release version*, the *specification version*, and the *implementation version*. These versions are based on the Java Package Versioning Specification and reflect the basic concepts of dependencies. You can define and edit dependencies in the Properties dialog of modules, which you can access via Properties ➤ API Versioning (see Figure 3-7).

First, define the major release version in this window. This is the version notifying the user of incompatible changes, compared to the previous version of the module. Here, the slash is used to separate the code name base from the version within the manifest file:

```
OpenIDE-Module: com.galileo.netbeans.module/1
```

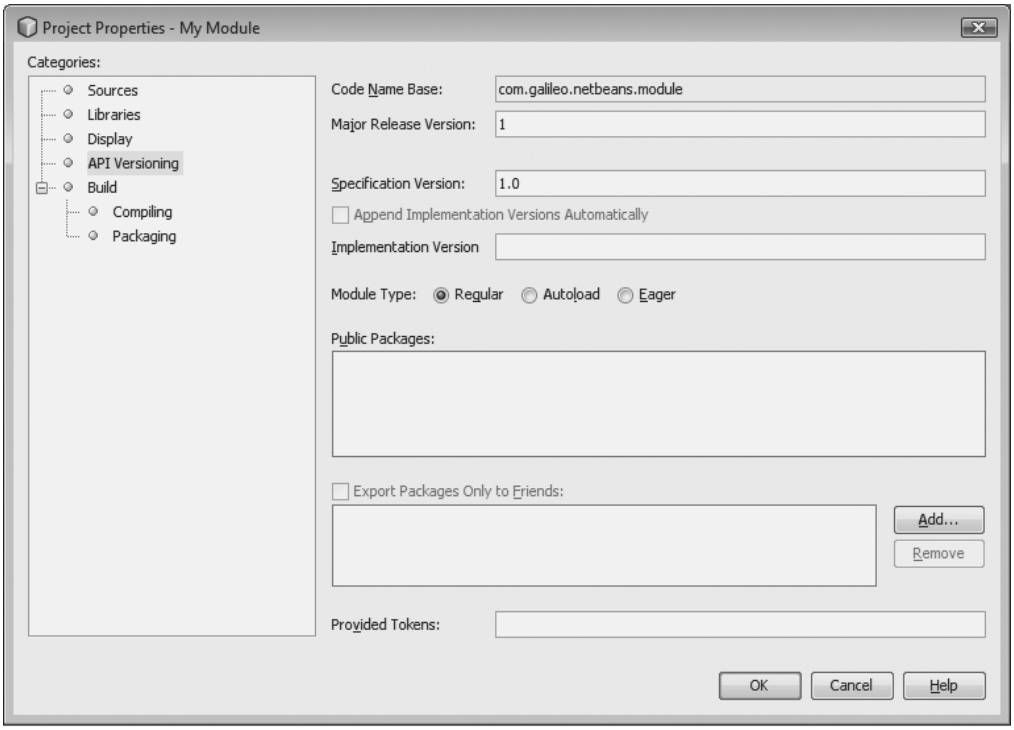


Figure 3-7. *Setting the module version*

The most important version is the specification version. The Dewey decimal system is used to define this version:

```
OpenIDE-Module-Specification-Version: 1.0.4
```

The implementation version is freely definable text. Typically, a timestamp is used, providing the date and time. In that way, you determine it is unique. If not explicitly set in the

Properties dialog of the module, the IDE adds the implementation version when the module is created, using the current timestamp, set within the manifest file:

```
OpenIDE-Module-Implementation-Version: 200701231820
```

On the other hand, if you define your own implementation version in the Properties dialog, the IDE adds the `OpenIDE-Module-Build-Version` attribute with the current timestamp.

In the list of public packages, all packages in your module are listed. To expose a package to other modules, check the box next to the package you want to expose. In doing so, you define the API of your module. Exposed packages are listed as follows in the manifest file:

```
OpenIDE-Module-Public-Packages:
   com.galileo.netbeans.module.*,
   com.galileo.netbeans.module.model.*
```

To restrict access to the public packages (e.g., to allow only your own modules to access the public packages), you can define a module's friends. You define them beneath the list of public packages in the API Versioning section of the Properties dialog. These are then listed as follows in the manifest file:

```
OpenIDE-Module-Friends:
   com.galileo.netbeans.module2,
   com.galileo.netbeans.module3
```

Defining Dependencies

Based on these various versions, define your dependencies. To that end, three different types of dependencies are available: a module depends on a module, a package, or a version of Java.

Module Dependencies

You define and edit module dependencies via Properties ➤ Libraries, as shown in Figure 3-8.

NO DEPENDENCY? NO ACCESS!

To use classes from another module, including the NetBeans Platform's own modules, you must first define a dependency, as described in the following sections. That means, if you use a NetBeans Platform class in your module and the code editor cannot find the desired class, the problem can normally be fixed by simply setting a dependency on the module that provides the class.

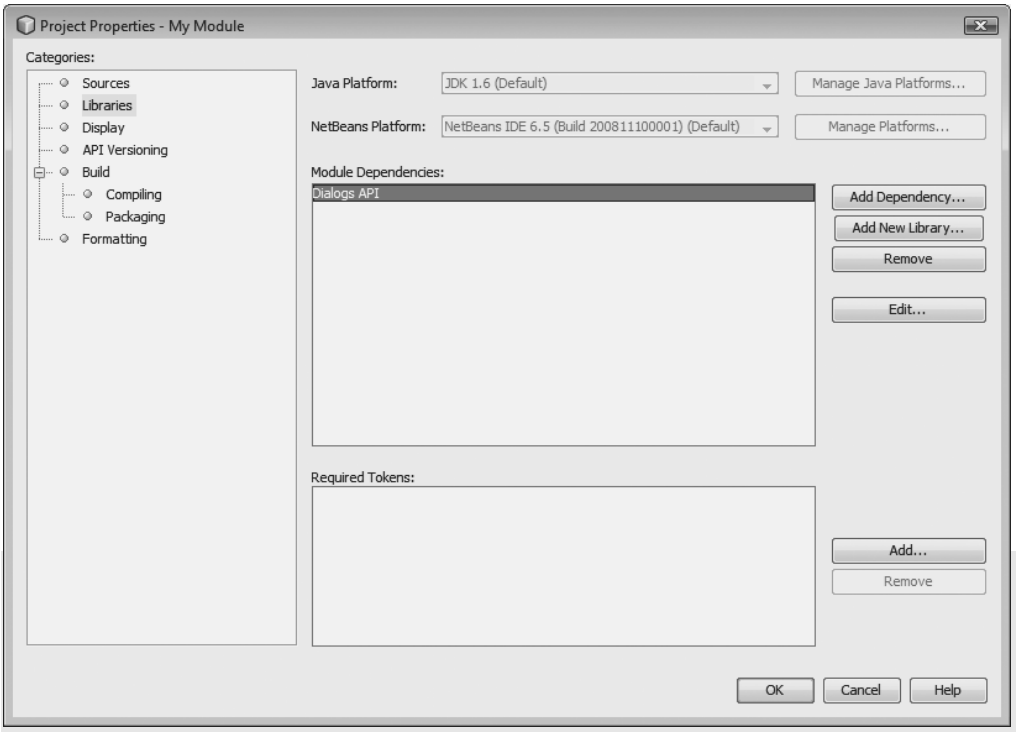


Figure 3-8. *Definition of module dependencies*

In this window, use Add Dependency to add dependencies to your module. The NetBeans module system offers different methods to connect dependencies to a particular module.

In the simplest case, no version is required. That means there should simply be a module available, though not a particular version; although where possible you still specify a version:

```
OpenIDE-Module-Module-Dependencies: com.galileo.netbeans.module2
```

In addition, you may require a certain version. In this case, the module version should be greater than 7.1. This is the most common manner in which to define dependencies:

```
OpenIDE-Module-Module-Dependencies: org.openide.dialogs > 7.1
```

If the module on which you want to depend has a major release version, it must be specified via a slash after the name of the module:

```
OpenIDE-Module-Module-Dependencies: org.netbeans.modules.options.api/1 > 1.5
```

Finally, you may also specify a range of major release versions:

```
OpenIDE-Module-Module-Dependencies: com.galileo.netbeans.module3/2-3 > 3.1.5
```

To create tight integration to another module, set an implementation dependency. The main reason for this approach is to make use of all the packages in the module, regardless of whether the module has exposed them or not. A dependency of this kind must be set with care, since it negates the principle of encapsulation. To enable the system to guarantee the consistency of the application, the dependency must be set precisely on the version of the given implementation version. However, this version changes with each change to the module.

```
OpenIDE-Module-Module-Dependencies: com.galileo.netbeans.module2 = 200702031823
```

Select the required dependency in the list (see Figure 3-8) and click the Edit button. In this window (shown in Figure 3-9), you set various types of dependencies.

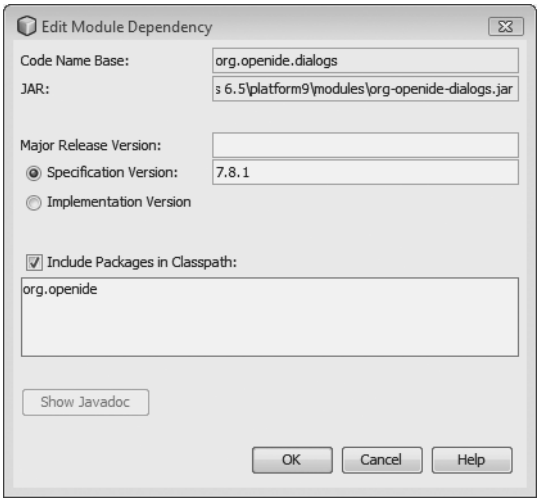


Figure 3-9. *Editing module dependencies*

Java Package Dependency

NetBeans lets you set a module dependency on a specific Java version. A dependency of this kind is set in the manifest file:

```
OpenIDE-Module-Package-Dependencies: javax.sound.midi.spi > 1.4
```

Java Version Dependency

If the module depends on a specific Java version, such as Java 5, specify that in the module properties under Properties ➤ Sources, using the Source Level setting. Aside from that, you can require a specific version of the Java Virtual Machine:

```
OpenIDE-Module-Java-Dependencies: Java > 1.5, VM > 1.0
```

You can require an exact version using the equal sign or require a version that is greater than the specified version.

Lifecycle

To influence the lifecycle of a module, implement a *module installer*. The Module System API provides the `ModuleInstall` class, from which we derive our own module installer class. The following methods can be overridden in the module installer class:

`validate()`: This method is 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()`: This method is called when an installed module is loaded. Here, actions required to initialize the module are called.

`uninstalled()`: When the module is removed from the application, this method is called.

`closing()`: Before a module is ended, this method is called. Here, you 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()`: If all modules are ready to end, this method is called. Here, you call the actions for the successful verification of the module in question.

ONLY USE THE MODULE INSTALLER WHEN ABSOLUTELY NECESSARY

When using these methods, consider whether the actions you're calling could be set declaratively instead. In particular, in the cases of the methods `validate()` and `restored()`, consider that these methods influence the startup time of the whole application. For example, when services are registered, you could instead use entries in the layer file or the Java Extension Mechanism (see Chapter 6). These enable the code to be invoked as needed, without impacting the startup time of the application as a whole.

Listing 3-7 shows the structure of the module installer class.

Listing 3-7. *Structure of a module installer class*

```
public class ModuleLifecycleManager extends ModuleInstall {
   public void validate() throws IllegalStateException {
      // e.g., check for a license key and throw an
      // IllegalStateException if this is not valid.
   }
   public void restored() {
      // called when the module is loaded.
   }
   public void uninstalled() {
```

```
        // called when the module is uninstalled.
    }
    public boolean closing() {
        // called to check if the module can be closed.
    }
    public void close() {
        // called before the module will be closed.
    }
}
```

To record the state of the module installer class over different sessions, override the methods `readExternal()` and `writeExternal()` from the `Externalizable` interface, which is implemented by the `ModuleInstall` class. There you store and retrieve necessary data. When doing so, it is recommended to first call the methods to be overridden on the superclass.

To let the module system know right from the start that a module provides a module installer, and where to find it, register it in the manifest file:

```
OpenIDE-Module-Install: com/galileo/netbeans/module/ModuleLifecycle.class
```

Look at how the module installer is created. The NetBeans IDE provides a wizard to create this file (see Figure 3-10). Go to File ➤ New File and choose Module Development ➤ Module Installer.

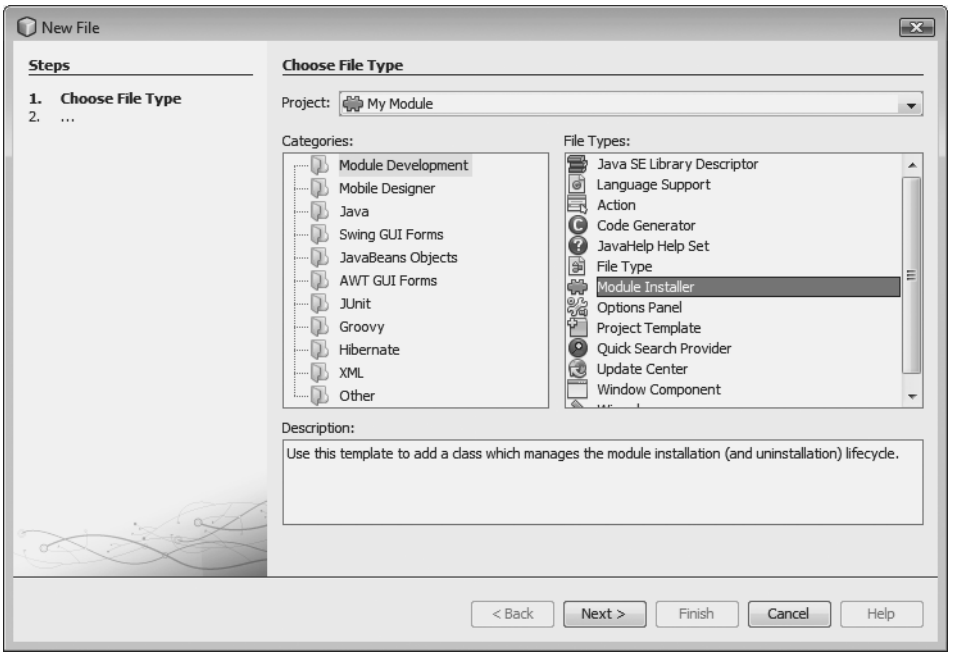


Figure 3-10. *Creating a module installer*

Click Next and then click Finish to complete the wizard. Now the `ModuleInstall` class is created in the specified package and registered in the manifest file. You now need only override

the required methods. For example, override the `closing()` method to show a dialog confirming whether the application should really shut down, as shown in Listing 3-8.

Listing 3-8. *Dialog for shutting down an application*

```
import org.openide.DialogDisplayer;
import org.openide.NotifyDescriptor;
import org.openide.modules.ModuleInstall;
public class Installer extends ModuleInstall {
   public boolean closing() {
      NotifyDescriptor d = new NotifyDescriptor.Confirmation(
         "Do you really want to exit the application?",
         "Exit",
         NotifyDescriptor.YES_NO_OPTION);
      if (DialogDisplayer.getDefault().notify(d) == NotifyDescriptor.YES_OPTION) {
         return true;
      } else {
         return false;
      }
   }
}
```

Be aware that this module requires a dependency on the Dialogs API to be able to use the NetBeans dialog support. Defining dependencies was described previously in the "Defining Dependencies" section of this chapter, while information about the Dialogs API can be found in Chapter 8.

To try this new functionality, invoke Run ➤ Run Main Project (F6). When the application shuts down, the dialog is shown and you can confirm whether or not the application should actually be shut down.

Module Registry

Modules normally need not worry about other modules. Nor should they need to know whether other modules exist. However, it might sometimes be necessary to create a list of all available modules. The module system provides a `ModuleInfo` class for each module, where all information about modules is stored. The `ModuleInfo` objects are available centrally via the Lookup, and can be obtained there as follows:

```
Collection<? extends ModuleInfo> modules =
   Lookup.getDefault().lookupAll(ModuleInfo.class);
```

The class provides information such as module name, version, dependencies, current status (activated or deactivated), and the existence of service implementations for the current module. Use the `getAttribute()` method to obtain this information from the manifest file.

To be informed of changes, register a `PropertyChangeListener`, which informs you of the activation and deactivation of modules in the system. You can also register a `LookupListener` that informs you of the installation and uninstallation of modules. For example, a listener could be defined as shown in Listing 3-9.

Listing 3-9. *Reacting to changes to the module system*

```
Lookup.Result<ModuleInfo> res = Lookup.getDefault().lookupResult(ModuleInfo.class);
result.addLookupListener(new LookupListener() {
   public void resultChanged(LookupEvent lookupEvent) {
      Collection<? extends ModuleInfo> c = res.allInstances();
      System.out.println("Available modules: " + c.size());
   }
});
res.allItems(); // initialize the listener
```

Using Libraries

When developing rich client applications, you'll more than likely need to include external libraries in the form of JAR files. Since the whole application is based on modules, it is desirable to integrate the external JAR file in the form of a module. That has the advantage of setting dependencies on the module, enhancing the consistency of the application as a whole. You can also bundle multiple JAR files into a single module, after which you need no longer put the physical JAR files on the application classpath, as is normally done when developing applications.

Library Wrapper Module

To achieve the scenario just outlined, create a *library wrapper module*. The NetBeans IDE provides a project type and a wizard for this purpose. To create a new Library Wrapper project, go to File ➤ New Project, and use the dialog shown in Figure 3-11 to choose the category NetBeans Modules, followed by the project type Library Wrapper Module.

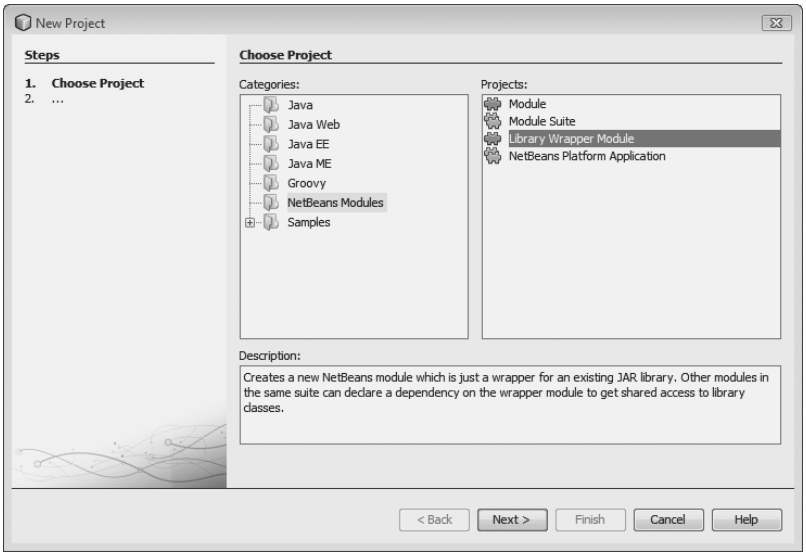


Figure 3-11. *Creating a Library Wrapper project*

Click Next to choose the required JAR files. You can choose one or more JAR files here (hold down the Ctrl key to select multiple JAR files). You are also able to add a license file for the JAR you are wrapping as a module. In the next step, provide a project name, as well as a location to store the new module. Specify the Module Suite or Platform Application to which the library wrapper module belongs. Click Next again to fill out the Basic Module Configuration dialog, as shown in Figure 3-12.

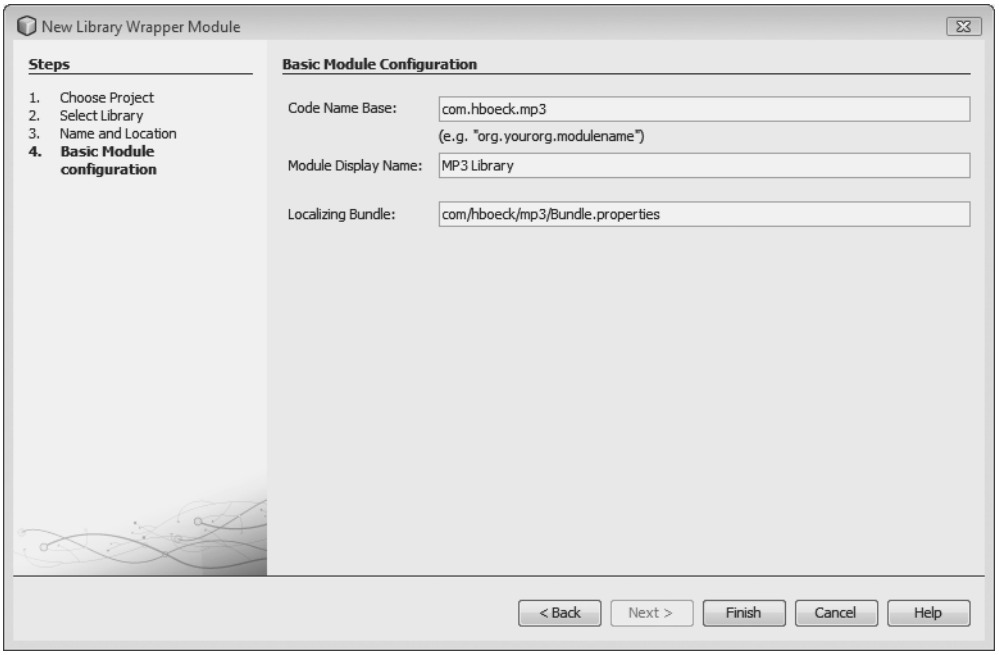


Figure 3-12. *Library wrapper module configuration*

Here, you define the code name base. Normally this field is prefilled with the name of the JAR file. Provide the module with a name and a localizing bundle (see the "Module Manifest" section earlier in the chapter) to localize the manifest file. With a click of the Finish button, you create your new module project.

When expanding the `Source Packages` folder in the Projects window, observe that there is only a `Bundle.properties` file and a manifest file. The library, which is encapsulated by the module, is found in the folder `release/modules/ext`, which is seen in the Files window.

To understand how a library wrapper module works, take a look at the related manifest file. It's found in the Projects window, within the `Important Files` file. Note that the following information (in Listing 3-10) may not reflect exactly what is found in your specific manifest file. Certain information, such as the public packages, are only created when you build the project. To see the entire manifest file, create the module and then look at the manifest file within the created JAR file, found in the `build/cluster/modules` folder of your application. The exposed

packages are also seen in the Properties dialog of the library wrapper modules, under the API Versioning tab. There, you can easily hide packages you do not want to have exposed.

Listing 3-10. *Manifest file of a library wrapper module*

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.6.0-b105 (Sun Microsystems Inc.)
OpenIDE-Module: com.hboeck.mp3
OpenIDE-Module-Public-Packages:
   com.hboeck.mp3.*,
   com.hboeck.mp3.id3.*,
   ...
OpenIDE-Module-Java-Dependencies: Java > 1.4
OpenIDE-Module-Specification-Version: 1.0
OpenIDE-Module-Implementation-Version: 070211
OpenIDE-Module-Localizing-Bundle: com/hboeck/mp3/Bundle.properties
OpenIDE-Module-Requires: org.openide.modules.ModuleFormat1
Class-Path: ext/com-hboeck-mp3.jar
```

Two very important things have been accomplished by the wizard. First, it marked all packages in the third-party library with the attribute `OpenIDE-Module-Public-Packages`, making all these packages publicly accessible. That's useful because it means the library can now be used by other modules. Second, the wizard marked the library it found in the `ext/` folder with the `Class-Path` attribute, putting it on the module classpath. In this way, the classes in the library can be loaded by the module classloader. The library wrapper module is automatically assigned the `Autoload` type (see the "Module Types" section near the beginning of the chapter), so that it is only loaded when needed.

Adding a Library to a Module

It is advisable to always use a library wrapper module when integrating a third-party library into an application. Creating a new module in this way for a third-party library adds to the value and maintainability of the application as a whole, because you can then set dependencies on the library via the module that wraps it.

In some cases, it is desirable to add the third-party library directly to the module using it. Taking this approach is simple and similar to creating library wrapper modules.

Open the `Project Metadata` file (`project.xml`) in the `Important Files` node in your module project, within the Projects window. For each library you want to include in your module, create a `class-path-extension` entry (see Listing 3-11). Via the `runtime-relative-path` attribute, define the path where the library is found in the distribution. Before, this was done automatically by the wizard. Use the `binary-origin` attribute to specify the location where the original library is found. As you see, this is the same approach as taken with library wrapper modules.

Listing 3-11. *Project metadata file with classpath extension*

```
<project xmlns="http://www.netbeans.org/ns/project/1">
  <type>org.netbeans.modules.apisupport.project</type>
```

```
  <configuration>
    <data xmlns="http://www.netbeans.org/ns/nb-module-project/3">
      <code-name-base>com.galileo.netbeans.module</code-name-base>
      <class-path-extension>
        <runtime-relative-path>ext/org-hboeck-mp3.jar</runtime-relative-path>
        <binary-origin>release/modules/ext/org-hboeck-mp3.jar</binary-origin>
      </class-path-extension>
    </data>
  </configuration>
</project>
```

Via this entry in the project metadata file, the creation of the module results in the library being copied to the `ext/` folder. In the manifest file, the entry `Class-Path: ext/com-hboeck-mp3.jar` is added. In contrast to a library wrapper module, the packages of the library are not exposed. As a result, they can only be used by the module where the library is found, which is normally the reason for taking this approach in the first place. It is also possible to define the packages as being public, which is automatically the case with library wrapper modules.

WHEN SHOULD I FOLLOW WHICH APPROACH?

Bear in mind that putting a third-party library into a module works against its modularity and maintainability. Put a library directly into a module only when the library will be used solely by the module in question, and if it is not a problem to distribute the third-party library together with the module that uses it.

Finally, avoid loading the same library from two different modules. This can lead to problems that are difficult to solve because their causes are hard to identify. Also, do not use the `Class-Path` attribute to reference module JAR files or libraries found in the `lib/` folder.

Summary

In this chapter, you learned how the underlying module system of NetBeans Platform applications is structured and how it functions. The module system is part of the runtime container. First, we looked at the structure of a NetBeans module. You learned about the many configuration options that are defined in the manifest file. In addition to the manifest file, a module optionally provides a layer file. You learned how to make contributions to the whole application, via registration entries in a module layer file.

You created your first module, learned how modules use code from other modules, and explored the lifecycle of modules and how third-party libraries integrate in a module via a library wrapper module. Finally, you discovered how those kinds of modules work, and you got some hands-on experience with them.

CHAPTER 4

Actions

Let's Make the NetBeans Platform Do Something!

An application's actions are among its most important and central components. In this chapter, you'll learn how to provide actions and how to integrate them into the application's menu bar, toolbar, and pop-up menus. You'll be introduced to specific NetBeans Platform classes that simplify the development and integration of action classes into your application. Moreover, the concepts are explained relevant to the creation of context-sensitive actions.

Overview

The NetBeans Platform bases its actions on the Swing Action Framework. Ultimately, every action rests upon Swing's `Action` interface. In the simplest case, an action class implements the `Action` interface or, even more straightforward, extends the `AbstractAction` class. Indeed, actions that derive from these standard base classes can be integrated into NetBeans Platform applications.

In addition to these standard Swing base classes, the NetBeans Platform has some of its own base classes from which you can derive your own actions. These base classes offer better integration with the NetBeans Platform for purposes of delivering standard action representatives for entries in menu bars, toolbars, and pop-up menus. Besides these, you can support these classes by the implementation of asynchronous tasks, as well as with the implementation of context-sensitive actions. Figure 4-1 illustrates the class hierarchy of these abstract base classes, while the following sections explain the purpose of each.

Actions are registered centrally in the layer file of your module, within the `Actions` folder. Other classes reference these registration entries, as you will learn in Chapter 5, which discusses the creation of menu bars and toolbars. The advantage is that actions can be used multiple times in different places, such as in the menu bar or toolbar, or even bound to a Swing component. However, though the action is displayed in multiple ways, only one instance is created.

Another reason for centralized declaration of an action is the possibility of toolbar customization. When removing an entry from the toolbar, the action itself is not removed, only the reference to the action.

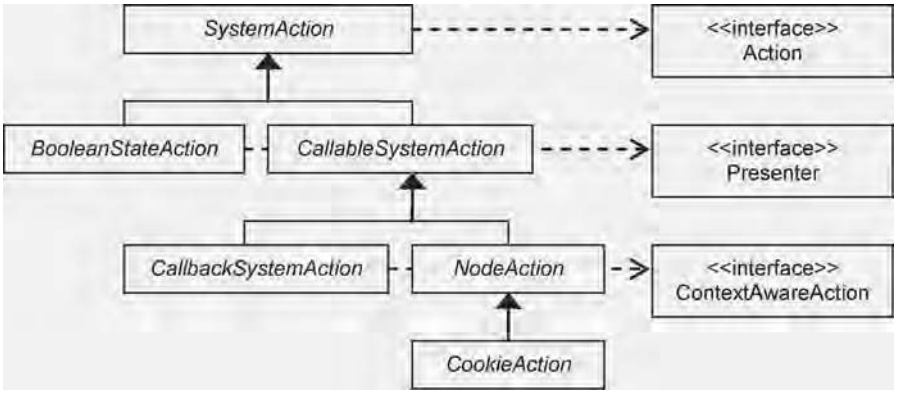


Figure 4-1. *Hierarchy of the NetBeans Platform's action base classes*

Providing Action Classes

The NetBeans IDE provides a wizard for creating new action classes. This is very practical, particularly for the insertion and organization of actions in the menu bar or toolbar, as well as for the insertion of separators between them. Additionally, it is handy that the registration of all entries in the layer file is taken care of for you, as the wizard handles this task by adding the necessary registration entries to the layer file.

Always Enabled Actions

Choose File ➤ New File ➤ Module Development ➤ Action to invoke the wizard for creating a new action class. In the first step, select the action type. The choice is between an *always enabled* and a *conditionally enabled* action.

We will use an always enabled action in this section. In the next step, specify how to integrate your action class into the menu bar, toolbar, or both, and set a shortcut, if necessary (see Figure 4-2).

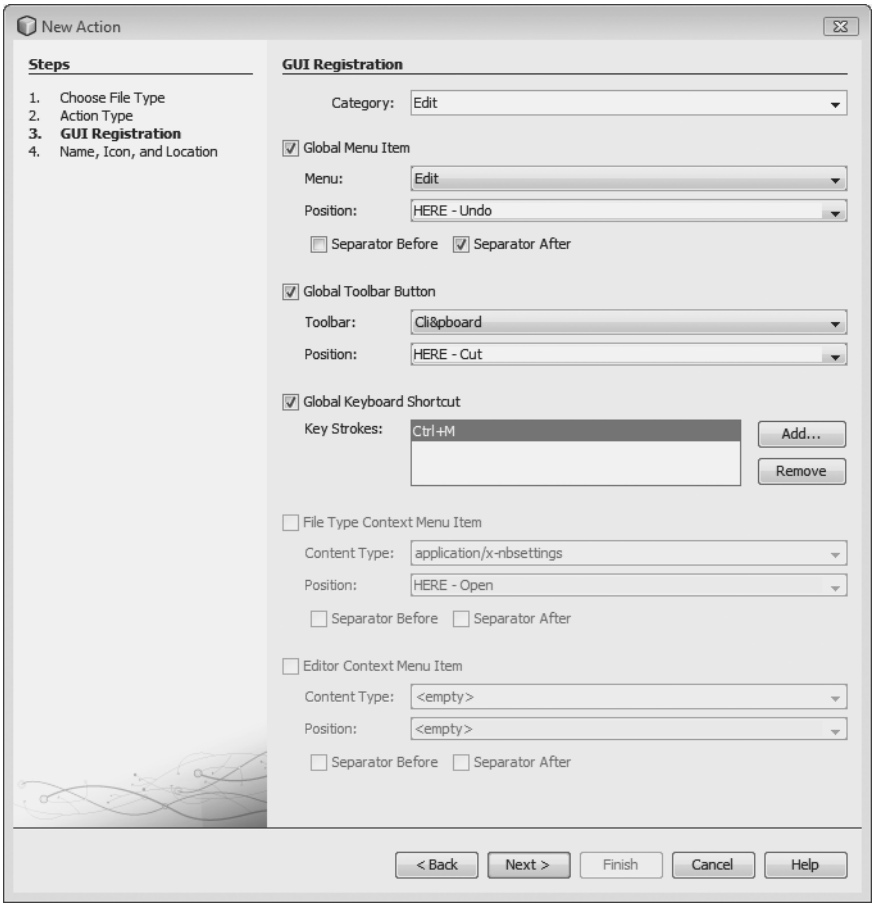


Figure 4-2. *Creating an action via the New Action wizard*

Next, select a category for the action. The categories represent semantic groupings of the actions. Either select a preexisting category or create a new one. In addition, assign your action to a menu bar, a toolbar, or both, and set the position where the action will be displayed. Drop-down menus show you possible locations for display. HERE identifies the location where display of your action will be inserted.

Although you can allow the action to be added to an existing menu bar or toolbar, it is also possible to tweak the layer file later, so the action will be displayed in a different position (see Chapter 5). The wizard also makes possible inserting a separator before or after the action (or both).

The dialog allows you to specify a keyboard shortcut that invokes the action. Once you approve of your settings, click Next to reach the final step of the wizard. There you set the name of the action class and the display name that will be shown in the menu bar. You must add an icon for the action. Typically, this should be 16×16 pixels in size. Since the user can show the toolbar in two different dimensions, you should provide the same icon in a different size, preferably 24×24 pixels. You need not specifically select this 24×24 pixel icon. It should merely be found in the same folder, with the same file name as the 16×16 pixel icon.

For example, if the 16×16 icon is named `icon.gif`, the accompanying 24×24 icon must be named `icon24.gif`. (Later, you can also add icons named `icon_pressed.gif`, `icon_disabled.gif`, and `icon_rollover.gif` for the related states, in the folder where `icon.gif` is found.) Afterward, click Finish to end the wizard and allow the action class to be created. Let's now look at the newly created action class (see Listing 4-1).

Listing 4-1. *Example of an always enabled action class*

```
public final class MyFirstAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // TODO implement action body
    }
}
```

While earlier NetBeans versions used the NetBeans Platform `CallableSystemAction` (see Figure 4-1, shown previously), an always enabled action is created and handled as an `ActionListener` interface from NetBeans Platform 6.5 onward. Thus, we use a plain Java action class, making the integration of existing action classes into a NetBeans Platform application very easy. The integration into the NetBeans Platform is done in a declarative manner, via the layer file. The required entries are generated by the Action wizard. The entries that are generated are shown in Listing 4-2, within the `Actions` folder.

Listing 4-2. *Integration of an always enabled action into the NetBeans Platform*

```
 <file name="com-galileo-netbeans-module-MyFirstAction.instance">
   <attr name="SystemFileSystem.localizingBundle"
           stringvalue="com.galileo.netbeans.module.Bundle"/>
   <attr name="delegate" newvalue="com.galileo.netbeans.module.MyFirstAction"/>
   <attr name="displayName"
           bundlevalue="com.galileo.netbeans.module#CTL_MyFirstAction"/>
   <attr name="iconBase" stringvalue="com/galileo/netbeans/module/icon.gif"/>
   <attr name="instanceCreate" methodvalue="org.openide.awt.Actions.alwaysEnabled"/>
   <attr name="noIconInMenu" stringvalue="false"/>
</file>
```

These attributes have the following meaning:

`SystemFileSystem.localizingBundle`: Specifies the bundle where localized string literals are found.

`delegate`: Specifies the default constructor of the action class. It is also possible to specify a factory method.

`displayName`: Specifies the key of the action's name property in the properties bundle.

`iconBase`: Specifies the path and the base name of an icon for this action.

`instanceCreate`: Specifies the factory method responsible for the creation of the action class. For this purpose, the utility class `Actions` provides the method `alwaysEnabled()`.

CallableSystemAction

The previously described approach to creating and integrating an action into a NetBeans Platform application is the approach taken since the NetBeans Platform 6.5. It is the recommended approach to take. Previously, a NetBeans Platform `CallableSystemAction` class was created by the wizard for always enabled actions. It is the superclass of the action classes that are described later in this chapter.

Listing 4-3 shows an example. The class is derived from the `CallableSystemAction` class. Rather than defining attributes in the layer file, as is done from the NetBeans Platform 6.5 onward, several methods need to be overridden to set attributes.

Listing 4-3. *Always enabled action with CallableSystemAction as superclass*

```
public final class MyFirstAction extends CallableSystemAction {
   public void performAction() {
      // TODO implement action body
   }
   public String getName() {
      return NbBundle.getMessage(MyFirstAction.Class, "CTL_MyFirstAction");
   }
   protected String iconResource() {
      return "com/galileo/netbeans/module/icon.gif";
   }
   public HelpCtx getHelpCtx() {
      return HelpCtx.DEFAULT_HELP;
   }
   protected boolean asynchronous() {
      return false;
   }
}
```

In the `performAction()` method, you add the logic of your action implementation. Using the `getName()` method, you are able to provide the name displayed in the menu item or as the toolbar button's tooltip. This name is loaded via the bundle key `CTL_MyFirstAction` (further information about this is found in Chapter 10). In the resource bundle, you can also set a mnemonic for your action (see the section "Shortcuts and Mnemonics").

The `iconResource()` method merely provides the base path of the icons you want to display. At the appropriate time, these are loaded in a relevant size and state as required. If you do not want an icon at all, simply do not override the `iconResource()` method.

The toolbar button displays the name of the action. If you want to ensure the icon assigned to the action is not shown in the menu, set the property `noIconInMenu` for the action. This is done by calling `putValue("noIconInMenu", Boolean.TRUE)` on the action. Setting this property substitutes the real icon of the action with a placeholder, which is an empty icon. Using this feature is only useful if none of the items in the menu show their icons, so that all the items have a similar appearance.

The `getHelpCtx()` method is provided by the interface `HelpCtxProvider` and defines the help ID of a specific help topic relating to the action. This topic is addressed in detail in Chapter 9.

Finally, you should override the `asynchronous()` method. That method allows the action to be created asynchronously, simply by returning `true` from this method. In turn, this causes `performAction()` to be run in a separate thread, instead of on the event dispatch thread (EDT). Therefore, by means of the asynchronous implementation, you ensure that access to GUI components is put into the EDT. Returning `false` here, the action logic is executed synchronously in the EDT. In this case, you should ensure that no long-running processes are handled in the duties of the `performAction()` method, because the GUI would otherwise be blocked.

CallbackSystemAction

The abstract action class `CallbackSystemAction` is a subclass of `CallableSystemAction`. It differs from its parent by the fact that it can delegate to another action, which is typically a context-sensitive action.

`CallbackSystemAction` contains no action logic, but delegates to an *action performer*. These action classes are used especially by global actions—that is, actions that provide different behavior depending on their context. For example, these are actions such as performing searches, copying, or pasting. Depending on the current context, such actions behave in different ways, depending on what needs to happen in the context of the current objects. Such global actions are offered out of the box by the NetBeans Actions API.

The action performer is made available by a Java `ActionMap`. It is registered in a map, together with the key of the `CallbackSystemAction` class that is delivered via the method `getActionMapKey()`. All classes that derive from `JComponent` can make use of an `ActionMap`. That means the NetBeans superclass `TopComponent`, which creates windows that integrate into NetBeans Platform applications (see Chapter 5 for details), also has an `ActionMap`.

These `ActionMaps` are made available via a Lookup. It is the task of the `CallbackSystemAction` class to look in the global Proxy Lookup to determine whether an `ActionMap` exists and, if so, whether an action performer has been registered for a given action. If this is the case, the action representatives (menu bar or toolbar entries) are automatically activated. If no action performer is available, the related entries are deactivated.

In the following code listing, a refresh action is shown. This action will behave differently, depending on which window is currently selected. Therefore, the `performAction()` method is not needed, since action handling will be provided by the selected window. The class should look like Listing 4-4.

Listing 4-4. *The global action class that inherits from CallbackSystemAction and delegates to an action performer*

```
public final class RefreshAction extends CallbackSystemAction {
   public String getName() {
      return NbBundle.getMessage(RefreshAction.class, "CTL_RefreshAction");
   }
   protected String iconResource() {
      return "com/galileo/netbeans/module/icon.gif";
```

```
    }
    public HelpCtx getHelpCtx() {
       return HelpCtx.DEFAULT_HELP;
    }
    protected boolean asynchronous() {
       return false;
    }
}
```

Running the module, you see the action deactivated in the menu bar, as well as in the toolbar. That is because no action performer currently exists. A general outline for creating an action performer is described following, although the full discussion of `TopComponent`s comes later, in Chapter 5. Once we discuss how `TopComponent`s are created, you can return here and try out the action class described in Listing 4-5.

Listing 4-5. *Registration of an action performer for a CallbackSystemAction*

```
final class MyTopComponent extends TopComponent {
   private MyTopComponent() {
      ...
      RefreshAction ra = RefreshAction.get(RefreshAction.class);
      getActionMap().put(ra.getActionMapKey(), new AbstractAction() {
         public void actionPerformed(ActionEvent event) {
           // refresh content of TopComponent
         }
      });
   }
}
```

First obtain the singleton instance of the previously created `RefreshAction` class to determine the key needed for the `ActionMap`. Next, use the `getActionMap()` method, defined by the `JComponent` class, to get the `ActionMap` of the `TopComponent` and to add an instance of the action implementation as the key of the `RefreshAction`. Then the action created by the class `AbstractAction` becomes the action performer in the context of `MyTopComponent`.

You can insert into the `ActionMap` any action that implements the `Action` interface. Therefore, you can also use the `CallableSystemAction` class of your choice. Now, as soon as the `MyTopComponent` window is active, `RefreshAction` becomes active. And when the action is invoked, `MyTopComponent`'s `actionPerformed()` method is invoked.

For a comprehensive understanding, you should know how a `CallbackSystemAction` class obtains an action performer. Simply adding it to the `ActionMap` is not enough. After all, the `RefreshAction` class must somehow be connected to the `ActionMap`. The Lookup handles this connection. The `TopComponent`'s local Lookup is made available to `CallbackSystemAction` via the global Proxy Lookup.

Therefore, the `TopComponent` needs to ensure that the `ActionMap` is added to its own Lookup. Only then will the `CallbackSystemAction` be able to find the action performer. By default, the `ActionMap` is already in the `TopComponent`'s local Lookup, which means that you don't need to worry about it yourself. However, you do need to be aware of this when you create your own Lookup via `associateLookup()`, or if you override `getLookup()`. In these cases, pay attention to the fact that the `ActionMap` must be added to the `TopComponent`'s Lookup.

Let's leave this discussion at this point. You are referred to Chapter 6 in this book, in which many more details are found about the NetBeans Lookup, together with examples that illustrate its usage.

Note, however, that in the manner described in this chapter, you can add many action performers to a `CallbackSystemAction`. These are then handled according to their context. Possibly you've been wondering how to use the actions that are typically default members of NetBeans Platform applications, such as `CopyAction`, `CutAction`, and `DeleteAction`. Well, all these classes are simply `CallbackSystemAction`s, which you can use by providing a related action performer, as shown in this section, via the `RefreshAction` example.

CookieAction

The abstract superclass `CookieAction` is, just like `CallbackSystemAction`, a class you can use to create context-sensitive actions. First, let's talk about *node actions*. As indicated by its name, actions of the type `NodeAction` are dependent on nodes. A node is, as you will discover in detail in Chapter 7, the visual representation of a particular piece of data. For example, a node can be shown in a tree structure or opened in an editor. Each `TopComponent` (and therefore each window within the NetBeans Platform) can make use of one or many activated nodes.

It is precisely these activated nodes that form the context of a `CookieAction`. Context-sensitivity is constructed from interfaces, which are called *cookies*. And now you begin to understand how to create context-sensitive implementations of an action. The node on which the action is to operate implements an interface specifying the method that should be invoked by the action. The action can specify a set of cookies, the presence of which in the active node (if the active node implements one of these interfaces) determines whether the action is enabled.

To create an action of the type `CookieAction`, you again use the Action wizard. However in this case, select the type Conditionally Enabled. Moreover, you can immediately specify the cookies for which the action should be enabled. Later, you can change these in your code. In the wizard, you also specify whether several nodes can be active at the same time. In the next step of the wizard, define integration of the action into the menu bar or toolbar, as explained in the previous section. Upon completion of the wizard, your action should look something like Listing 4-6.

Listing 4-6. *Action class of type CookieAction that is enabled if the active node implements the EditCookie interface*

```
public final class MyCookieAction extends CookieAction {
   protected void performAction(Node[] activatedNodes) {
      EditCookie ec = activatedNodes[0].getLookup().lookup(EditCookie.class);
      ec.edit();
   }
   protected int mode() {
      return CookieAction.MODE_EXACTLY_ONE;
   }
   protected Class[] cookieClasses() {
      return new Class[] {
         EditCookie.class
      };
   }
```

```
    protected boolean surviveFocusChange() {
        return false;
    }
    public String getName() {
        return NbBundle.getMessage(MyCookieAction.class, "CTL_MyCookieAction");
    }
    protected String iconResource() {
        return "com/galileo/netbeans/module/icon.gif";
    }
    public HelpCtx getHelpCtx() {
        return HelpCtx.DEFAULT_HELP;
    }
    protected boolean asynchronous() {
        return false;
    }
}
```

As you can see, the method `performAction()` distinguishes itself by the fact that it passes in the activated nodes, which together form the context of the action. Via these nodes, you gain access to cookies, which invoke the action. The `mode()` method lets you specify the conditions for activation. On that note, the constants listed in Table 4-1 are available to you.

Table 4-1. *Constants for setting conditions under which the action is enabled*

Constant	Condition for Activating the Action
`MODE_ALL`	Action is enabled if one or several nodes are selected that implement all cookies
`MODE_ANY`	Action is enabled if one or several nodes are selected, while at least one of them implements cookies
`MODE_EXACTLY_ONE`	Action is enabled if exactly one node is active that implements cookies
`MODE_ONE`	Action is enabled if one or several nodes are selected, of which exactly one implements cookies
`MODE_SOME`	Action is enabled if one or several nodes are selected, of which some, though not all, implement cookies

Via the method `cookieClasses()`, specify the cookies—that is, the interfaces—that the active node must implement in order for the action to be enabled. Let's take `EditCookie` as an example. Though the `surviveFocusChange()` method is not added to the code by the wizard, you should normally override this and let it return `false`. Since the default implementation of the method provides `true`, the action will remain enabled even if the applicable `TopComponent` is no longer current. The remaining methods should be familiar to you from the sections that covered the `CallableSystemAction` and `CallbackSystemAction` classes.

We still need a node that permits an action to become enabled. The node must derive from the abstract superclass `Node`. However, in general you should instead feel free to implement the `Node` subclass `AbstractNode`. It offers a base implementation of a node and is usually exactly what you need.

Following, as an example, you see the `MyNode` class, without any other logic. In a real-life scenario, the node represents a file of a certain type. However, here the class exists for no reason other than to clarify the relationship between the `CookieAction` and `Node` classes. In our `CookieAction` class (created in Listing 4-4), we specified that the node should implement the `EditCookie` interface.

Let's do so at this point (see Listing 4-7). The interface specifies the `edit()` method, which consists of an empty implementation. The method will be called later by the action class and exists to make the context-sensitive action logic available.

Listing 4-7. *Node that provides the context of the action*

```
public class MyNode extends AbstractNode implements EditCookie {
   public MyNode() {
     super(Children.LEAF);
   }
   public void edit() {
      // edit something depending on the data this node represents
   }
}
```

Now, in your `TopComponent` (which could be a file editor in which the file that the node represents is opened), use the `setActivatedNodes()` method to define the `TopComponent`'s activated node. Typically, you would do this at the time when the file is opened. Rather than using this method, you can also add the activated node to the local Lookup (see Listing 4-8). The exact connection between the node and its various representations is described in Chapter 7.

Listing 4-8. *Defining the active node, by means of which the action is enabled, to the extent the applicable cookies of the node have been implemented*

```
final class MyTopComponent extends TopComponent {
   private MyTopComponent() {
      MyNode node = new MyNode();
      ...
      setActivatedNodes(new Node[]{node});
      // or more general with
      // associateLookup(Lookups.fixed(node, getActionMap()));
   }
}
```

Let's look again at the `performAction()` method of the action:

```
protected void performAction(Node[] activatedNodes) {
   EditCookie ec = activatedNodes[0].getLookup().lookup(EditCookie.class);
   ec.edit();
}
```

When the action is used, we receive the active node as parameters, which thereby provide the context of the action. Since we defined the action as only active (and thereby selectable) when the active node has implemented the `EditCookie` interface, we can assume that this cookie is now available to us. A node has access to a local Lookup, which we obtain via the `getLookup()` method. The Lookup gives us access to the `EditCookie` interface of the node,

which in this case is an instance of the `MyNode` type. Now only the `edit()` method needs to be called, which delegates to the node. Therefore, we need not know the actual action logic of the class in order to create our context-sensitive action.

General Context-Sensitive Action Classes

In the two previous sections you learned how to create a context-sensitive action based on the `CallbackSystemAction` class and the `CookieAction` class. Take particular note of the fact that the `CallbackSystemAction` is dependent on an `ActionMap`, while the `CookieAction` has a relationship with the `Node` class.

Next, yet another approach will be introduced. You will be given a starting point for providing generic context-sensitive action classes with the help of the `Lookup` class (see Chapter 6 for details about Lookup). You will see how to use Lookup to create a general context-sensitive action, by means of which you will (similar to how it is done with the `CookieAction` class) use a `Class` object to determine when an action should be enabled. You will not, however, be dependent upon a `Node` class, nor any other NetBeans superclass, since you will observe how any class or interface can be used to determine the applicable context of an action.

First, we provide a generic abstract superclass that inherits from `AbstractAction`, while implementing `LookupListener` and `ContextAwareAction`. As private data elements, we have `Lookup`, which sets up the context, and `Lookup.Result`, with which we can monitor our Lookup. The default constructor uses the global Proxy Lookup as the context. Through this Proxy Lookup we get access to the local Lookup of the currently active window (see Chapter 6). With an additional constructor, you are able to provide a special context. If the action (such as a pop-up menu) has a node added to it, the `createContextAwareInstance()` method is automatically called, which receives the context of the node.

From this context, we receive a `Lookup.Result` for the class type to which the action should be sensitive. We attach a listener to the `Lookup.Result` so that we are informed about the presence or absence of an instance of the class type and provide appropriate handling via the `resultChanged()` method, which enables or disables the action.

We implement the `actionPerformed()` method and call the abstract `performAction()` method, which must be implemented by the subclass. We pass the instance, which forms the context of the action, to this method (see Listing 4-9). In this way, usage and access to the context in the subclasses is simplified.

As you can see, this is how the context is passed via the Lookup to the concrete action classes. Moreover, a subclass (the actual action class) must implement the `contextClass()` method, which specifies the class type that determines whether the action should be enabled.

Listing 4-9. *Abstract superclass for a general contextual action*

```
public abstract class ContextAction<T> extends AbstractAction
   implements LookupListener, ContextAwareAction {
   private Lookup context = null;
   private Lookup.Result<T> result = null;
   public ContextAction(Lookup context) {
      init(context);
```

```
    }
    private void init(Lookup context) {
        this.context = context;
        result = context.lookupResult(contextClass());
        result.addLookupListener(this);
        resultChanged(null);
    }
    public void resultChanged(LookupEvent ev) {
        setEnabled(result.allItems().size() != 0);
    }
    public void actionPerformed(ActionEvent e) {
        performAction(result.allInstances().iterator().next());
    }
    public abstract Class<T> contextClass();
    public abstract void performAction(T context);
}
```

Next, let's look at an example action that derives from the `ContextAction` defined previously (see Listing 4-10). First, we need to override the methods `performAction()` and `contextClass()`. In `performAction()`, we call the method `doSomething()` of the `MyInterface` interface, which is an example of a context-sensitive action. Using the `contextClass()` method, we provide a class object of the `MyInterface` interface to which the action should react. Besides these requirements, we must still implement `createContextAwareInstance()`, with which we create a new instance for the general context. As mentioned, this method can be used (for example) at the creation of a context-sensitive menu for a node.

Listing 4-10. *Example of a context-sensitive action class that becomes active if an instance of the interface MyInterface is in the global Proxy Lookup*

```
public final class MySensitiveAction extends ContextAction<MyInterface> {
    public MySensitiveAction() {
        this(Utilities.actionsGlobalContext());
    }
    public MySensitiveAction(Lookup context) {
        super(context);
        putValue(NAME,
            NbBundle.getMessage(MySensitiveAction.class,"CTL_MySensitiveAction"));
        putValue(SMALL_ICON,new ImageIcon(
            ImageUtilities.loadImage("com/galileo/netbeans/module/icon.gif", true)));
    }
    public Class<MyInterface> contextClass() {
        return MyInterface.class;
    }
    public void performAction(MyInterface context) {
        context.doSomething();
    }
    public Action createContextAwareInstance(Lookup context) {
        return new MySensitiveAction(context);
    }
}
```

The example action `MySensitiveAction` should be active if the `MyInterface` interface is available in the global Proxy Lookup (`Utilities.actionsGlobalContext()`).

This interface provided in the example is, as in most other cases, a window that implements the `TopComponent` class. In the `doSomething()` method of the interface, we implement the action logic that will be invoked by the action. We must make sure that the instance of the `MyInterface` interface is in the local Lookup, which will be represented by the global Proxy Lookup.

This is achieved by using the `Lookups` factory to create a `Lookup` containing the `MyTopComponent` instance, while defining this `Lookup` as our local Lookup via a call to `associateLookup()` (see Listing 4-11). As soon as the window is activated, the instance of the `MyInterface` interface is in the global Proxy Lookup, causing the `resultChanged()` method in `ContextAction` to be called, activating the action. The action is deactivated when the window is no longer current.

Listing 4-11. *The class MyTopComponent is a window that shows the context for the action. If this window has the focus, the action should be active.*

```
final class MyTopComponent extends TopComponent implements MyInterface {
   private MyTopComponent() {
      associateLookup(Lookups.fixed(getActionMap(), this));
   }
   public void doSomething() {
      // called by the context-sensitive action
   }
}
```

The advantage of this approach is that you can use the `contextClass()` method of the action class to provide a class type of your choice. In this case, for example, we provided the `MyInterface` interface. It would be as easy to provide the `TopComponent` class instead, so the action would be enabled for each and every window. Another solution would be to provide only the `MyTopComponent` class, resulting in the action being active only when the `MyTopComponent` window is current.

Registering Actions

In Listing 4-2, you already saw how to integrate an action into your application. Except that with the new approach the action's attributes are also specified, the registration with an `.instance` file is the same for all mentioned action classes.

Actions are registered in the central folder `Actions`. There they can be divided into different groups, essentially different folders in the layer file (see Listing 4-12). From this central actions registry, the menu bar and toolbar are created (in Chapter 5, you'll learn how to build menu bars and toolbars from the actions).

Listing 4-12. *Registration of actions in the layer file*

```
<folder name="Actions">
   <folder name="Edit">
      <file name="com-galileo-netbeans-module-MyCookieAction.instance"/>
      <file name="com-galileo-netbeans-module-MyFirstAction.instance"/>
```

```
      <file name="com-galileo-netbeans-module-MySensitiveAction.instance"/>
   </folder>
   <folder name="Window">
      <file name="com-galileo-netbeans-module-MyAction.instance"/>
   </folder>
</folder>
```

The System Filesystem now has the necessary information required for the creation of actions. Therefore, you need not instantiate them yourself. They are simply integrated in this way—that is, declaratively. The options available for the definition and instantiation of actions are explained in Chapter 3, which covers the general definitions of classes in the layer file and how they are instantiated.

Shortcuts and Mnemonics

Shortcuts are also centrally defined and administered via the layer file. This is done in the `Shortcuts` folder. The file element defines the shortcut, with a reference to an action class as a file attribute. You see from this that a shortcut is not created for a menu entry, but an action. A shortcut consists of one or more modifiers and an identifier, separated by a minus sign:

```
modifier-identifier
```

Be aware that the following keys are represented by characters in the layer file, where they are used as modifiers for shortcuts:

C: Ctrl

A: Alt

S: Shift

M: Cmd/Meta

In addition, there are two wildcards that guarantee the platform independence of the shortcuts. These should be used as shown here:

D: Ctrl or Cmd/Meta (on the Mac)

O: Alt or Ctrl (on the Mac)

As identifiers, all constants defined by the `KeyEvent` class are possible. For example, for `KeyEvent.VK_M`, you simply omit `VK_`. The identifier would simply be `M`, in this case.

For example, to create the shortcut Ctrl+M for the action `MyCookieAction` that we created in the "CookieAction" section, we need the code shown in Listing 4-13 in the layer file.

Listing 4-13. *Definition of shortcuts in the layer file*

```
<folder name="Shortcuts">
  <file name="D-M.shadow">
    <attr name="originalFile" stringvalue=
            "Actions/Edit/com-galileo-netbeans-module-MyCookieAction.instance"/>
```

```
  </file>
</folder>
```

Tip It can be helpful to look in the Javadoc for the functions `Utilities.keyToString()` and `Utilities.stringToKey()`. These are used for encoding shortcuts. In Table 4-2, you see example combinations for shortcuts. And if you do not know how to write the definition of a shortcut for a certain key, you can simply use the New Action wizard to help you (see Chapter 3 and Figure 4-2).

Table 4-2. *Examples of shortcuts and their corresponding entries in the layer file*

Shortcut	Entry in the Layer File
`Ctrl++`	`<file name="D-PLUS.shadow">`
`Ctrl+Shift+S`	`<file name="DS-S.shadow">`
`F3`	`<file name="F3.shadow">`
`Alt+Enter`	`<file name="O-ENTER.shadow">`
`Alt+O`	`<file name="O-O.shadow">`
`Alt+Shift+S`	`<file name="OS-S.shadow">`

Mnemonics are inserted directly into the name of an action, via the insertion of an ampersand (&). This can be coded directly into an action, or within its related properties file:

```
CTL_OpenMyWindow=Open MyWind&ow
```

Note that the mnemonics are only shown if the user holds down the Alt key.

Summary

In this chapter, we discussed actions. You learned how to quickly and efficiently create actions via a wizard in the NetBeans IDE. You also saw the various types of actions that are available and learned how to make effective use of them. For example, some actions are always available, while others are only available within specific contexts. Finally, we looked at how actions are registered, how they integrate into applications, and how to set shortcuts and mnemonics for actions.

CHAPTER 5

User Interface Design
Let's Create Some Windows!

In this chapter, the structure of the user interface of NetBeans Platform applications is discussed. You'll learn about menu construction, toolbar creation, and window system usage. As well, this chapter shows how windows and components are created, integrated, and displayed within the window system.

Overview

The NetBeans Platform provides a *window system*, which is a container that manages the menu bar, toolbars, and status bar, as well as the windows your modules make available (see Figure 5-1). The following sections discuss these components in detail.

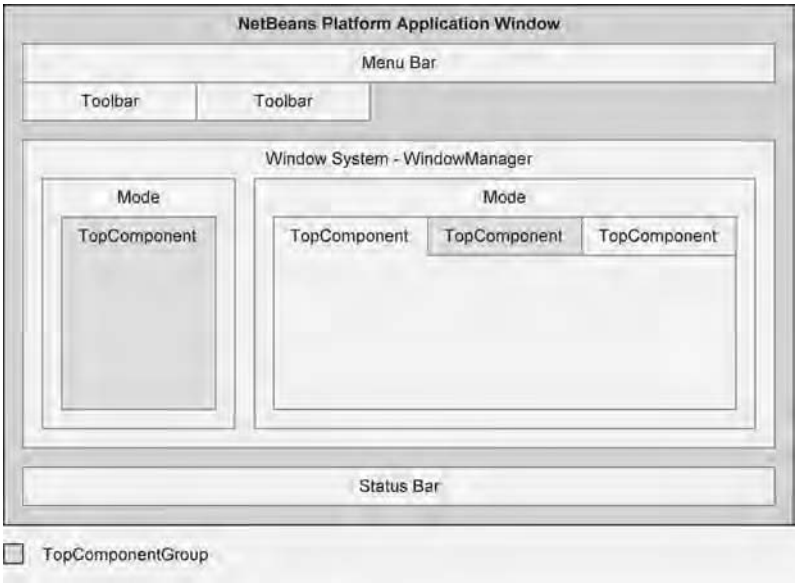


Figure 5-1. *Structure of the NetBeans application window*

Menu Bar

The menu bar of an application based on the NetBeans Platform is created by the NetBeans Platform via the System Filesystem. Every menu (as well as the menu entries) is defined in a module layer file. This allows each module to declaratively add its menu entries to the menu bar. They implement the action performed when selecting a menu entry.

Creating and Adding Menus and Menu Entries

As an example, we will add a menu entry to the menu. The simplest way to do so is to use the Action wizard as described in Chapter 4. The class shown in Listing 5-1 is used to show how an action is added to the menu bar.

Listing 5-1. *Creating an action class for a menu entry*

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyFirstMenuAction implements ActionListener {
   public void actionPerformed(ActionEvent evt) {
      // do something
   }
}
```

The action class is registered in the layer file (see Listing 5-2), as shown in the "Always Enabled Actions" and "Registering Actions" sections in Chapter 4.

Listing 5-2. *Adding a menu entry in the layer file*

```
<filesystem>
  <folder name="Actions">
    <folder name="Edit">
      <file name="com-galileo-netbeans-module-MyFirstMenuAction.instance">
         <attr name="SystemFileSystem.localizingBundle"
                stringvalue="com.galileo.netbeans.module.Bundle"/>
         <attr name="delegate"
                newvalue="com.galileo.netbeans.module.MyFirstMenuAction"/>
         <attr name="displayName"
                bundlevalue="com.galileo.netbeans.module#CTL_MyFirstMenuAction"/>
         <attr name="iconBase" stringvalue="com/galileo/netbeans/module/icon.gif"/>
         <attr name="instanceCreate"
                methodvalue="org.openide.awt.Actions.alwaysEnabled"/>
         <attr name="noIconInMenu" stringvalue="false"/>
      </file>
    </folder>
  </folder>
  <folder name="Menu">
    <folder name="Edit">
      <file name="MyFirstMenuAction.shadow">
```

```
        <attr name="originalFile" stringvalue="
           Actions/Edit/com-galileo-netbeans-module-MyFirstMenuAction.instance"/>
      </file>
    </folder>
  </folder>
</filesystem>
```

The preceding listing adds a menu entry to the Edit menu. A folder element named `Edit` appears in the default folder `Menu`. In this folder, a file element is added, which in turn adds the menu entry. The attribute `originalFile` references the action class declared in the default folder `Actions`. Since the module system combines all layer files, all menu entries declared in the `Edit` folder are displayed in the Edit menu. Basically, a menu is created by the simple act of declaring a folder element. This allows arbitrarily cascaded menus. It is possible to move the preceding action to a submenu of Edit (see Figure 5-2). The resulting layer file looks like Listing 5-3.

Listing 5-3. *Creating a submenu*

```
<folder name="Menu">
  <folder name="Edit">
    <folder name="My Actions">
      <file name="MyFirstMenuAction.shadow">
        <attr name="originalFile" stringvalue="
          Actions/Edit/com-galileo-netbeans-module-MyFirstMenuAction.instance"/>
      </file>
    </folder>
  </folder>
</folder>
```

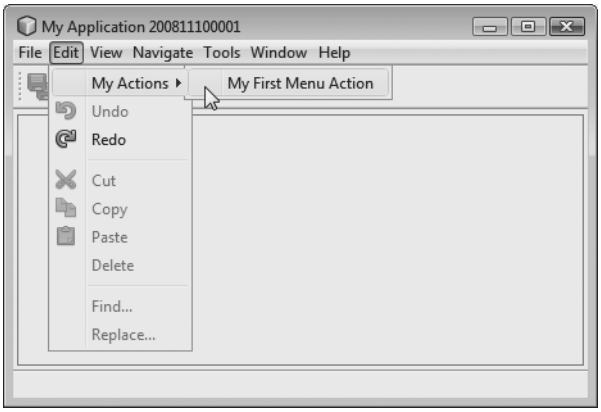


Figure 5-2. *Menu ➤ submenu ➤ menu entry*

The order of menus and menu entries is defined in the layer tree. It allows the developer to drag and drop entries into desired positions. Doing that defines the entries' positions via the `position` attribute in the layer file. For further information, see Chapter 3.

Inserting Separators

Separators are displayed between menu entries and added directly in the layer file. To add a separator below the menu entry of the preceding section, a layer file is modified as shown in Listing 5-4.

Listing 5-4. *Inserting a separator in the menu*

```
<file name="MyFirstMenuAction.shadow">
   <attr name="originalFile" stringvalue="
          Actions/Edit/com-galileo-netbeans-module-MyFirstMenuAction.instance"/>
   <attr name="position" intvalue="10"/>
</file>
<file name="javax-swing-JSeparator.instance">
   <attr name="position" intvalue="20"/>
</file>
```

When creating an action with the help of the Action wizard (see Chapter 4), the developer has the option of letting the wizard add a separator above as well as below the menu entry. The wizard then modifies the layer file accordingly.

Hiding Existing Menu Entries

To hide existing menus or menu entries originating from either the NetBeans Platform or other application modules, simply use the layer tree (see Chapter 3). Open the `Important Files` ➤ `XML Layer` ➤ `<this layer in context>` folder of a module. The module's entries are displayed along with all entries of the application and the NetBeans Platform modules. All menus and menu entries are displayed in the `Menu Bar` folder. The desired entry is selected and then deleted via the context menu. The entry is not physically deleted, but made invisible in the layer file. When the View menu and the Edit ➤ Find menu entry are deleted, the entries shown in Listing 5-5 are added to the layer file.

Listing 5-5. *Hiding menu entries*

```
<folder name="Menu">
  <folder name="View_hidden"/>
  <folder name="Edit">
    <file name="org-openide-actions-FindAction.instance_hidden"/>
  </folder>
</folder>
```

The suffix `_hidden` was added to the corresponding entries. To make these hidden entries available again, the entries are simply removed from the layer file.

Creating a Custom Menu Bar

When creating a menu bar to be used in a module, use the NetBeans APIs. The Data Systems API provides the `MenuBar` class, which is a subclass of `JMenubar`, and has the ability to create its content from a `DataFolder` object. This enables the developer to define custom menus the same way as default menus in the layer file.

To do that, create a `DataFolder`. The System Filesystem is accessed with the method `getDefaultFileSystem()` (see Listing 5-6) and searched for the root folder of the menu, in this case `MyModuleMenu`. Then a `DataFolder` object is created for the root folder by calling the static method `findFolder()` and passing it directly to the `MenuBar` constructor.

Listing 5-6. *Creating a menu bar that reads its content from the System Filesystem*

```
FileSystem sfs  = Repository.getDefault().getDefaultFileSystem();
FileObject menu = sfs.findResource("MyModuleMenu");
MenuBar    bar  = new MenuBar(DataFolder.findFolder(menu));
```

Toolbars

The application window of the NetBeans Platform contains a toolbar area. There you can place your own toolbars. How to create, configure, and modify toolbars is described in the following sections.

Creating Toolbars

Adding actions to the toolbar is accomplished the same way actions are added to the menu bar. Toolbars are defined in the default `Toolbars` folder in the layer file, as shown in Listing 5-7.

Listing 5-7. *Creating a toolbar and adding an action*

```
<folder name="Toolbars">
  <folder name="MyToolbar">
    <file name="MyFirstMenuAction.shadow">
      <attr name="originalFile" stringvalue="
        Actions/Edit/com-galileo-netbeans-module-MyFirstMenuAction.instance"/>
    </file>
  </folder>
</folder>
```

Using this entry, the new toolbar `MyToolbar` is defined, and a reference to the previously created and declared action in the `Actions` folder is added.

Configuring Toolbars

Which and in what order toolbars are displayed is configured in a toolbar configuration file. The DTD for that XML format is found in this book's Appendix. By default, there are three toolbars. These are defined in the `Standard.xml` file of the `Core-UI` module and look like Listing 5-8.

Listing 5-8. *Default NetBeans Platform toolbar configuration: Standard.xml*

```
<Configuration>
   <Row>
      <Toolbar name="File"/>
      <Toolbar name="Edit"/>
      <Toolbar name="Memory"/>
```

```
   </Row>
</Configuration>
```

A custom configuration can be created. For example, this allows the hiding of toolbars based on context. In the configuration in Listing 5-9, the previously created toolbar named `MyToolbar` and the default `Edit` toolbar are displayed. The `File` toolbar is hidden. The resulting configuration looks like Listing 5-9.

Listing 5-9. *Custom toolbar configuration*

```
<!DOCTYPE Configuration PUBLIC
  "-//NetBeans IDE//DTD toolbar//EN"
  "http://www.netbeans.org/dtds/toolbar.dtd">
<Configuration>
   <Row>
      <Toolbar name="Edit"/>
      <Toolbar name="MyToolbar"/>
   </Row>
   <Row>
      <Toolbar name="File" visible="false"/>
   </Row>
</Configuration>
```

This newly created configuration can be stored under any given name. The configuration is referenced in the layer file to announce its existence to the NetBeans Platform. There, the storage path relative to the layer file is declared in an attribute named `url`, as shown in Listing 5-10.

Listing 5-10. *Registering a toolbar configuration*

```
<folder name="Toolbars">
   <file name="MyToolbarConfig.xml" url="Toolbars/MyToolbarConfig.xml"/>
</folder>
```

What remains to be done is to insert a line into the source code to activate the configuration displaying the desired toolbars. The module `UI Utilities` provides a practical API for that purpose:

```
ToolbarPool.getDefault().setConfiguration("MyToolbarConfig");
```

This call happens when a window is selected displaying a context-based toolbar. That is done in the "Window: TopComponent" section, when a custom window is created.

The class `ToolbarPool` is responsible for managing toolbars registered in the System Filesystem. Via the call to `getDefault()`, you get the `ToolbarPool` object created by the system. It is responsible for toolbars registered in the default folder `Toolbars`. The provided methods are described in Table 5-1. Additionally, you can create a separate `ToolbarPool` object to manage toolbars registered in a separate folder. The constructor simply requires a `DataFolder` object. How that is achieved is shown in the "Creating Custom Toolbars" section.

Table 5-1. *Useful methods of the ToolbarPool class*

Method	Functionality
`findToolbar(String name)`	Returns a specific toolbar
`getToolbars()`	Returns all toolbars available in this pool
`getConfiguration()`	Returns the name of the currently active configuration
`getConfigurations()`	Returns an array of all available configurations
`setConfiguration(String c)`	Changes the current toolbar configuration
`setPreferredIconSize(int s)`	Allows sizing icons in the toolbar; valid values are 16 and 24 pixels

Modification by the User

Clicking the right mouse button in the application's toolbar opens a context menu allowing the user to toggle the visibility of a toolbar (see Figure 5-3). Additionally, toolbars can be configured at runtime via the View ➤ Toolbars ➤ Customize menu. Via drag-and-drop, actions can be added or removed.

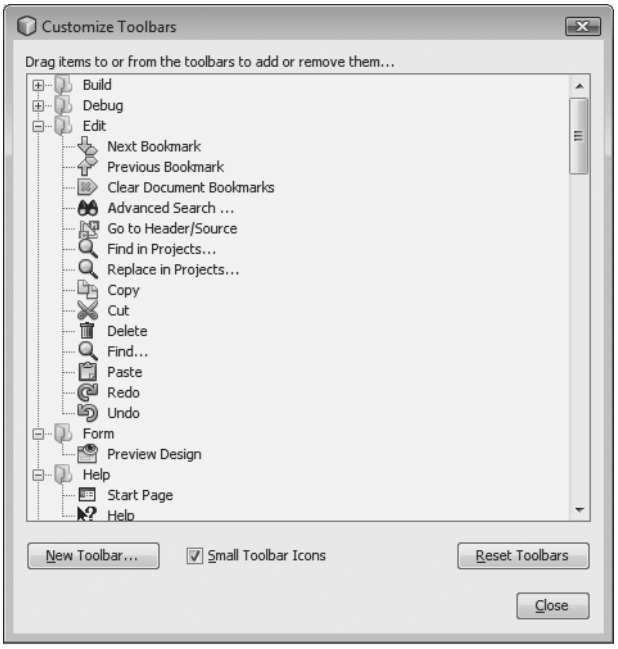


Figure 5-3. *User-defined configuration of toolbars*

Creating Custom Toolbars

Like the menu bar (see the previous "Creating a Custom menu Bar" section), developers can create a custom toolbar—even a pool of toolbars. For example, these toolbars can be used in a `TopComponent`. The `ToolbarPool` class therefore offers (like the `MenuBar` class) a constructor to pass in a `DataFolder` object representing a folder of toolbars in the System Filesystem. That allows the developer to define toolbars in the same way as default toolbars. The necessary steps are shown in Listing 5-11.

Listing 5-11. *Creating toolbars with content read from the System Filesystem*

```
FileSystem  sfs  = Repository.getDefault().getDefaultFileSystem();
FileObject  tbs  = sfs.findResource("MyToolbars");
ToolbarPool pool = new ToolbarPool(DataFolder.findFolder(tbs));
```

Further information about what components can be added to the toolbars via the System Filesystem is located in the API documentation of the `ToolbarPool` class.

Using Custom Control Elements

In Chapter 4, the NetBeans superclasses for actions implementing `Presenter` interfaces were shown. These interfaces specify methods providing graphic representations for menus, toolbars, and pop-up menus. These representations (menu entries or buttons) are regularly provided by the `CallableSystemAction` superclass. If a representation other than the default toolbar action is required, the method `getToolbarPresenter()` from the `Presenter.Toolbar` interface can be overridden. Listing 5-12 shows an action class utilizing a combo box as a control element to, for example, set a zoom level.

Listing 5-12. *User-defined control item for the toolbar action*

```
public class MyComboBoxAction extends CallableSystemAction {
   JComboBox box = new JComboBox(new String[]{"100%", "200%"});
   public MyComboBoxAction() {
      box.setMaximumSize(box.getPreferredSize());
      box.setAction(this);
   }
   public void performAction() {
      System.out.print("Adjust zoom to: ");
      System.out.println(box.getSelectedItem());
   }
   ...
   public Component getToolbarPresenter() {
      return box;
   }
}
```

The action class extends from either `CallableSystemAction` or from `CookieAction`. The custom control item is added as a private field. The maximum size of the combo box is set to its preferred size to avoid taking over the entire space of the toolbar. The connection between the action and the control item is important. That connection is achieved via the method `setAction()`, while passing a reference to the instance with the operator `this`. Upon activating the combo box, the action is executed. Lastly, the method `getToolbarPresenter()` returns the combo box. This way, rather than a default button, the combo box is displayed.

Window System

The window system is a framework provided by the NetBeans Platform. It is responsible for the administration and display of all application windows. It allows the user to customize the layout of the user interface.

Introduction

The window system is document based. That means the central section—that is, the editor section—is all about the display of several files in tabs. View sections are placed around the editor section. Components are arranged within the view sections. Usually, these supporting windows offering edit functionality to the documents. In the case of the NetBeans IDE, these windows provide the structure of the project, the Properties dialog, and the Output window.

By default, all windows are displayed in the NetBeans main application window. Since version 6 of the NetBeans Platform, undocking windows by using the context menu or dragging the window from the application window is possible. What that looks like is shown in Figure 5-4, where the project window is undocked. Docking and undocking allows for flexible window positioning. The floating window feature is especially useful when you are using multiple monitors.

The window system is comprised of *modes*. A mode is a NetBeans Platform class that provides a container for windows, displayed like a tab. The windows must be subclasses of `TopComponent`. Every displayed window is managed by the `WindowManager`. Windows can be grouped as well. The assembly of the window system is described in the layer file. This entails a description of the available modes, the windows that are displayed within them, and a definition of which window belongs to which group of windows. In the following sections, these windows are described in detail. How a developer uses these is illustrated.

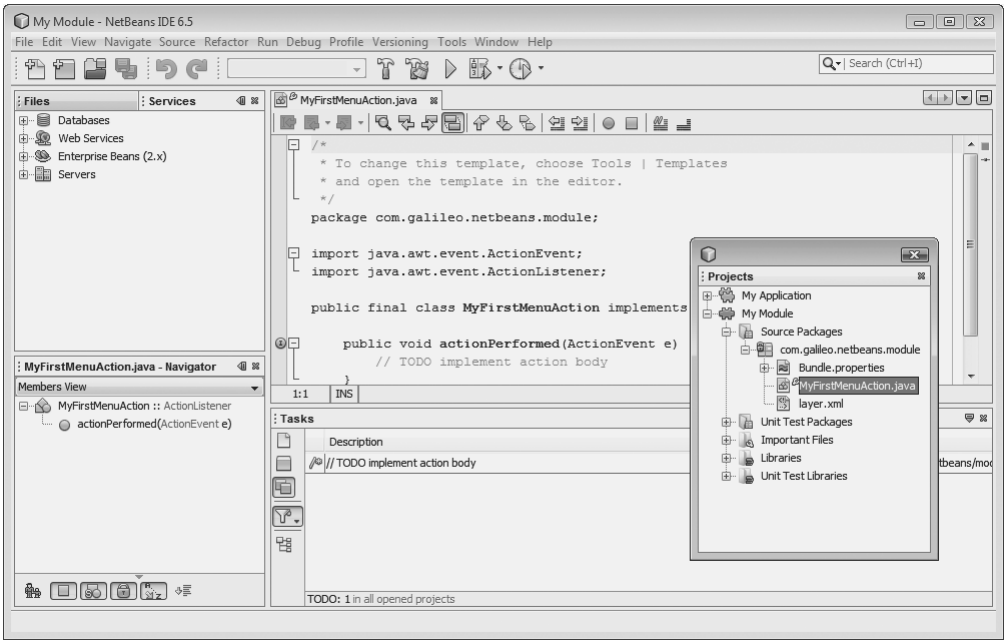


Figure 5-4. *NetBeans window system with floating windows*

Configuration

A module configures its windows, modes, and groups in the layer file, within the folder `Windows2`, as outlined in Listing 5-13.

Listing 5-13. *A window system configuration in the layer file*

```
<folder name="Windows2">
   <folder name="Components">
      <file name="MyTopComponent.settings" url="MyTopComponent.settings"/>
      <file name="MyTopComponent2.settings" url="MyTopComponent2.settings"/>
   </folder>
   <folder name="Modes">
      <folder name="explorer">
         <file name="MyTopComponent.wstcref" url="MyTopComponent.wstcref"/>
      </folder>
      <file name="MyMode.wsmode" url="MyMode.wsmode"/>
      <folder name="MyMode">
         <file name="MyTopComponent2.wstcref" url="MyTopComponent2.wstcref"/>
      </folder>
   </folder>
   <folder name="Groups">
      <file name="MyGroup.wsgrp" url="MyGroup.wsgrp"/>
      <folder name="MyGroup">
```

```
        <file name="MyTopComponent.wstcgrp" url="MyTopComponent.wstcgrp"/>
        <file name="MyTopComponent2.wstcgrp" url="MyTopComponent2.wstcgrp"/>
      </folder>
    </folder>
</folder>
```

In this example, there are

- Definitions for the windows `MyTopComponent` and `MyTopComponent2` in the `Components` folder
- A declaration that the window `MyTopComponent` is associated with the Mode `explorer` mode, which is defined by the platform
- A declaration that the window `MyTopComponent2` is associated with the newly created mode `MyMode`
- A group of windows named `MyGroup` that were added to the windows `MyTopComponent` and `MyTopComponent2`

This way, a module defines its windows, associates them with modes, and groups them together. In the following sections, further details concerning different reference file types will be shown.

This configuration is the default configuration a module defines. The default configuration is used by the window system upon the first start. When exiting the application, any changes made to the layout of the application (e.g., moving a window to another mode or closing a window group) are stored in the user directory folder `config/Windows2Local` in a hierarchy identical to the layer file (see Figure 5-5). Upon restarting, the application settings are read first. If none exist (as is the case when starting the application for the very first time), the settings are read from the layer file.

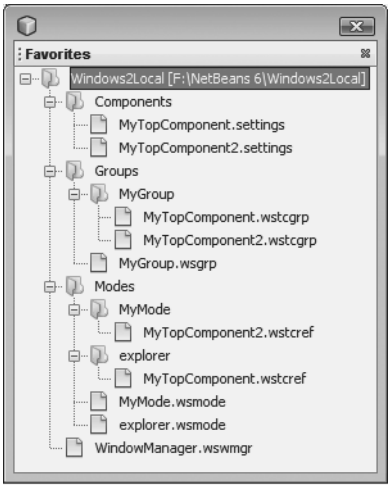


Figure 5-5. *Window system configuration*

Customization

From NetBeans Platform 6.5 onward, you have the possibility to customize the window system. You can disable several features to prevent undesired changes by the end user, such as the undocking or closing of windows.

To enable or disable these window system features, go to the Properties dialog of your NetBeans Platform Application (or Module Suite) and open the category Build ➤ Window System. The effect of disabling theses features is described in Table 5-2.

Table 5-2. *Window system features and the effects of disabling them*

Feature	Effect
`Window Drag and Drop`	The position of all windows is fixed. The user cannot move them to another position.
`Floating Windows`	The Undocking feature is disabled. The user cannot move windows out of the main window.
`Sliding Windows`	The windows cannot be minimized to the left/right side or to the bottom.
`Maximized Windows`	The windows cannot be maximized (neither by double-clicking nor via the context menu).
`Closing of Non-document Windows`	`TopComponent`s in view modes cannot be closed (neither over the symbol nor via the context menu).
`Closing of Document Windows`	`TopComponent`s in editor modes cannot be closed (neither over the symbol nor via the context menu).
`Window Resizing`	The size of all windows/modes is fixed and cannot be resized by the user.
`Respect Minimum Size When Resizing Windows`	Windows can be resized to less than their minimum size.

This new feature of window system customization allows you to define a consistent application layout. While in NetBeans Platform 6.5, the previously described features apply to the window system as a whole, you will be able to define them at the `TopComponent` level in the next NetBeans Platform release.

Window: TopComponent

The Window System API provides the class `TopComponent` for windows that integrate into the NetBeans Platform. A subclass of `JComponent`, it provides optional support for window interactions with the window system. A `TopComponent` always exists inside a mode, and as such is dockable, is automatically managed by the `WindowManager`, and receives lifecycle events.

Creating a TopComponent

The NetBeans IDE provides a wizard for creating `TopComponent`s. It creates a basic framework for new `TopComponent`s and registers them in the layer file. It is started by calling File ➤ New File, and selecting the category Module Development and file type Window Component (see Figure 5-6).

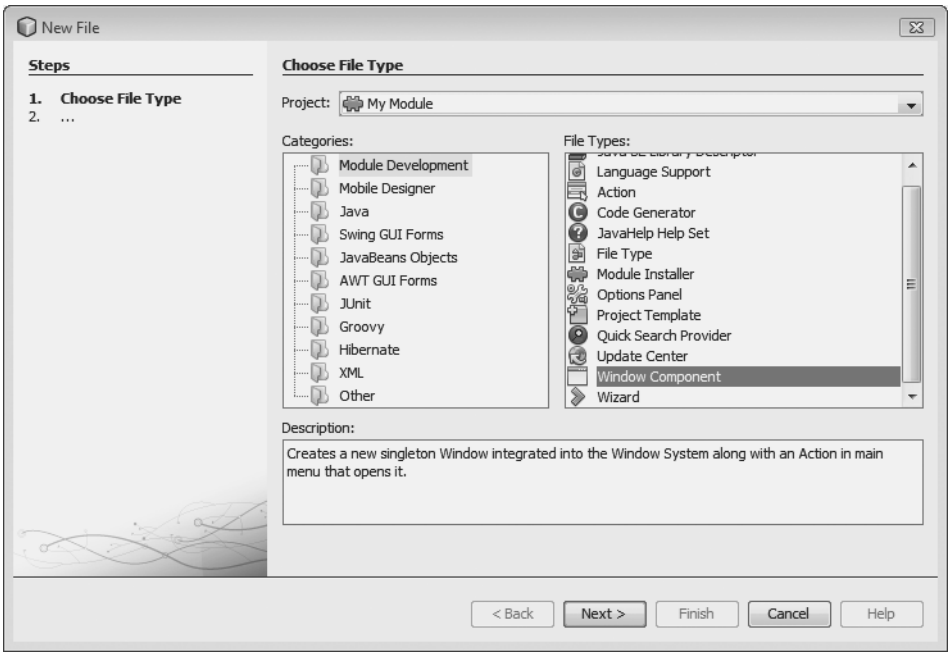


Figure 5-6. *Creating a TopComponent: Step 1*

Next (step 2), the wizard prompts developers to select the mode in which the `TopComponent` will display. Additionally, `TopComponent` display can be declared upon launching the application. Initially, only modes provided by the NetBeans Platform can be selected. That selection can be replaced in the layer file by custom modes. The last step (see Figure 5-7) of the wizard has the developer declare a prefix for the class name and select an icon for the `TopComponent`.

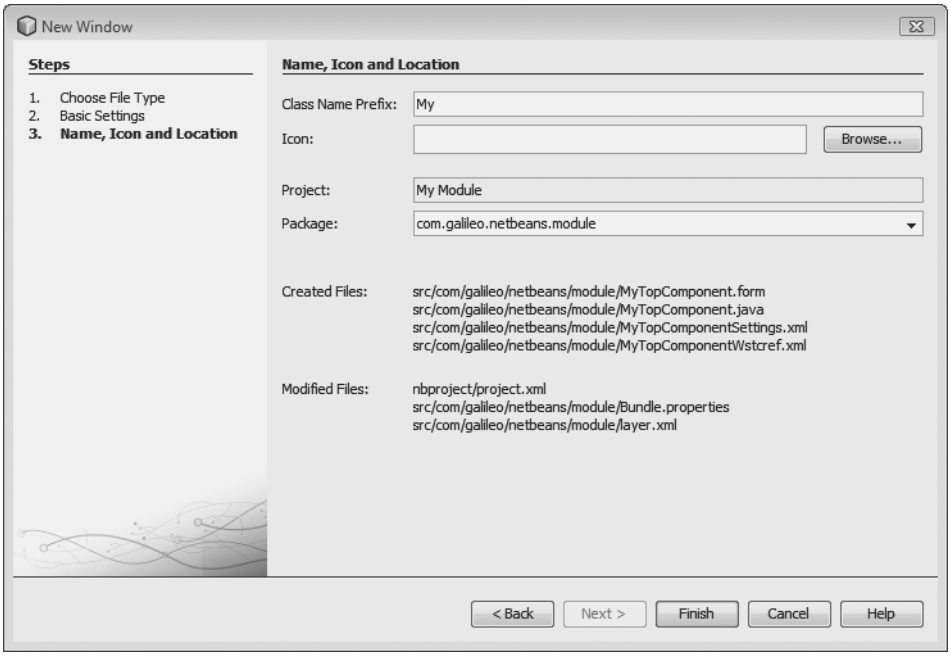


Figure 5-7. *Creating a TopComponent: Step 3*

The files the wizard creates are displayed in this dialog. These files comprise the `TopComponent` itself and two XML files that associate the `TopComponent` with a mode. Those files can be examined further once the wizard is finished. Clicking Finish prompts the wizard to initiate creation of the files.

Once the wizard finishes, everything necessary has been done. The developer is free to edit the `TopComponent` with the Form Editor to fit the `TopComponent` to its desired functionality. The component can be tested by selecting Run ➤ Run Main Project.

Examine the files the wizard created and the entries made to the layer file. First, the `TopComponent` is defined in the folder `Windows2/Components` (see Listing 5-14). This required mapping to a mode. The mapping is done in the folder `Windows2/Modes`.

Listing 5-14. *Definition and mapping of a TopComponent in the layer file*

```
<folder name="Windows2">
   <folder name="Components">
      <file name="MyTopComponent.settings" url="MyTopComponentSettings.xml"/>
   </folder>
   <folder name="Modes">
      <folder name="editor">
         <file name="MyTopComponent.wstcref" url="MyTopComponentWstcref.xml"/>
      </folder>
   </folder>
</folder>
```

Defining a `TopComponent` in the folder `Windows2/Components` necessitates a `.settings` file. Information required by the window system to create the `TopComponent` is in that file. Listing 5-15 defines the method `getDefault()` to create an instance of the `TopComponent`.

Listing 5-15. *Settings file to declaratively add a TopComponent*

```
<!DOCTYPE settings PUBLIC
 "-//NetBeans//DTD Session settings 1.0//EN"
 "http://www.netbeans.org/dtds/sessionsettings-1_0.dtd">
<settings version="1.0">
   <module name="com.galileo.netbeans.module" spec="1.0"/>
   <instanceof class="org.openide.windows.TopComponent"/>
   <instanceof class="com.galileo.netbeans.module.MyTopComponent"/>
   <instance class="
      com.galileo.netbeans.module.MyTopComponent" method="getDefault"/>
</settings>
```

Mapping a `TopComponent` to a mode is done using a TopComponent Reference file (see Listing 5-16). This file is created by the wizard.

Listing 5-16. *TopComponent Reference file mapping a TopComponent to a mode*

```
<!DOCTYPE tc-ref PUBLIC
 "-//NetBeans//DTD Top Component in Mode Properties 2.0//EN"
 "http://www.netbeans.org/dtds/tc-ref2_0.dtd">
<tc-ref version="2.0" >
   <module name="com.galileo.netbeans.module" spec="1.0"/>
   <tc-id id="MyTopComponent"/>
   <state opened="true"/>
</tc-ref>
```

This file contains only the unique identifier that references the `TopComponent`. This identifier correlates with the name given in `Windows2/Components` (`MyTopComponent.settings`) and the `PREFERRED_ID`. The stated element attribute `opened` specifies whether the `TopComponent` must be opened.

With NetBeans Platform 6.5, there is no need for a separate action class to open the `TopComponent`. Instead, the `TopComponent` class has the new static method `openAction()`. This method can be used in the same way as the `Actions.alwaysEnabled()` method (described in Chapter 4). This is a very nice approach because there is no need for even a single line of code. Instead, some tags need to be added in the layer file. If you create the `TopComponent` with the wizard as shown previously, the necessary tags (see Listing 5-17) are added automatically.

Listing 5-17. *Definition of an action for opening a TopComponent*

```
<file name="com-galileo-netbeans-module-MyAction.instance">
   <attr name="SystemFileSystem.localizingBundle"
           stringvalue="com.galileo.netbeans.module.Bundle"/>
   <attr name="component"
           methodvalue="com.galileo.netbeans.module.MyTopComponent.findInstance"/>
```

```
   <attr name="displayName"
         bundlevalue="com.galileo.netbeans.module.Bundle#CTL_MyAction"/>
   <attr name="iconBase" stringvalue="com/galileo/netbeans/module/icon.gif"/>
   <attr name="instanceCreate"
         methodvalue="org.openide.windows.TopComponent.openAction"/>
</file>
```

When looking at these tags in the layer file, it is evident that the `TopComponent` instance is created via the method `findInstance()`. This method first queries the `WindowManager` to check whether an instance of the `TopComponent` is opened. In that case, the previously created instance is returned. By default, a `TopComponent` is implemented as a singleton instance by the wizard. To create multiple instances of the `TopComponent`, it is easiest to use the wizard to create a JPanel form and change the superclass from `JPanel` to `TopComponent`. A `TopComponent` can be created via its constructor as well. Opening and activating remains the same as before:

```
TopComponent tc = new MyTopComponent();
tc.open();
tc.requestActive();
```

With the call to `open()`, the `TopComponent` is opened and added to the `WindowManager` for administration purposes. This mainly entails the process of storing and restoring the `TopComponent` when exiting or starting the application. The call to `requestActive()` leaves the `TopComponent` focused.

A `TopComponent` can dock into a specific mode directly, as outlined in Listing 5-18.

Listing 5-18. *Docking a TopComponent into a specific mode programmatically*

```
TopComponent tc = new MyTopComponent();
Mode m = WindowManager.getDefault().findMode("explorer");
if(m != null)
   m.dockInto(tc);
tc.open();
tc.requestActive();
```

States

A `TopComponent` can have several states, as outlined in Table 5-3.

Table 5-3. *Different states of a TopComponent*

State	Condition
`opened`	A `TopComponent` has the state opened when it is displayed in a tab inside one of the window system modes.
`closed`	A `TopComponent` has the state closed either after it is closed or if it hasn't yet been opened. Even closed, a `TopComponent` continues to exist.
`visible`	If a `TopComponent` is alone in its mode or is in top position, it remains in the visible state.

Table 5-3. *Different states of a TopComponent (Continued)*

State	Condition
invisible	If one TopComponent is covered by another inside a mode, it is in the invisible state.
active	A TopComponent is in the active state when it or one of its components is focused. In this state, the global selection context is provided by the TopComponent.
inactive	A TopComponent that is unfocused is in the inactive state.

Entering a specific state is announced via a call to one of the methods shown in Table 5-4. If a window has to perform an action in a specific state, the corresponding method simply has to be overridden.

Table 5-4. *Methods for the different states*

State	Method
opened	protected void componentOpened()
closed	protected void componentClosed()
visible	protected void componentShowing()
invisible	protected void componentHidden()
active	protected void componentActivated()
inactive	protected void componentDeactivated()

The "Configuring Toolbars" section earlier in the chapter showed how to create toolbar configurations, using them to display application-specific toolbars. Display on the currently active TopComponent is done by using two previously described methods (see Listing 5-19).

Listing 5-19. *Displaying and hiding toolbars based on context*

```
public class MyTopComponent extends TopComponent {
   private String origConfig = "Standard";
   private String myConfig   = "MyToolbarConfig";

   protected void componentActivated() {
      origConfig = ToolbarPool.getDefault().getConfiguration();
      ToolbarPool.getDefault().setConfiguration(myConfig);
   }

   protected void componentDeactivated() {
      ToolbarPool.getDefault().setConfiguration(origConfig);
   }
}
```

If the TopComponent is focused, the method componentActivated() is called. The current configuration is stored for later reactivation. Then the toolbar configuration MyToolbarConfig (from the "Configuring Toolbars" section of the chapter) is set. When another TopComponent is selected, the TopComponent loses focus and the method componentDeactivated() is called. The stored configuration is set there to restore previous toolbars.

Context Menu

When clicking with the right mouse button on the title bar of a TopComponent, a context menu is displayed, with actions like, e.g., Undock Window or Close Window. These actions are obtained from the TopComponent class via its getActions() method. To add actions to this context menu, override this method (see Listing 5-20). It is useful to add the actions declaratively. In Chapter 3, we mentioned possibly adding a folder and extension points to the layer file. That is done here. The actions are declared in the layer file and read on demand in the getActions() method.

Listing 5-20. *Reading actions for a context menu from the layer file*

```
public class MyTopComponent extends TopComponent {
   private List<Action> ca = null;
   @Override
   public Action[] getActions() {
      if (ca == null) {
         ca = new ArrayList<Action>(Arrays.asList(super.getActions()));
         ca.add(null); /* add separator */
         Lookup lkp = Lookups.forPath("ContextActions/MyTC");
         ca.addAll(lkp.lookupAll(Action.class));
      }
      return ca.toArray(new Action[ca.size()]);
   }
}
```

Firstly, the superclass's getActions() method is called to obtain default actions. With the help of the method Lookups.forPath(), a Lookup for the declared folder ContextActions/MyTC is created. The method lookupAll() then obtains all registered actions implementing the Action interface. When creating the menu, a null value is automatically replaced by a separator in the platform. The assembled list is returned as an array. The entry with the declared folder in the layer file looks like Listing 5-21.

Listing 5-21. *Defining context menu actions in the layer file*

```
<folder name="ContextActions">
  <folder name="MyTC">
    <file name="MyAction1.shadow">
      <attr name="originalFile"
         stringvalue="Actions/Edit/com-galileo-netbeans-module-MyAction1.instance"/>
    </file>
    <file name="MyAction2.shadow">
```

```
      <attr name="originalFile"
         stringvalue="Actions/Edit/com-galileo-netbeans-module-MyAction2.instance"/>
    </file>
  </folder>
</folder>
```

This method creates an extension point. Other modules add actions to the context menu by adding actions to the layer files folder `ContextActions/MyTC`. That allows flexibly extending the context menu from different modules without dependency.

Persistence

The window system is capable of storing opened `TopComponent`s upon exiting the application and restoring them upon restart. But there are use cases where storing the `TopComponent`s is undesired. Defining a stored `TopComponent` is done via the method `getPersistenceType()`. This method should always be overridden. The constants listed in Table 5-5 are available as return values.

Table 5-5. *Possible persistence types of a TopComponent*

Constant	Property
`PERSISTENCE_ALWAYS`	When returning this constant, the `TopComponent` is always stored.
`PERSISTENCE_ONLY_OPENED`	This constant defines a `TopComponent` stored only when opened in a mode.
`PERSISTENCE_NEVER`	With this constant, the `TopComponent` is never stored.

The window system calls the `Externalizable` interface methods `writeExternal()` and `readExternal()` upon storing or restoring a `TopComponent`. Override these methods to inject `TopComponent`-specific data for persistence. Don't forget to call the corresponding methods from the superclass. Admittedly, this is not the preferred way of making application data persistent. A far more flexible way is offered by the Preferences API in the form of the class `NbPreferences`. In Chapter 9, this approach is dealt with in detail in connection with administering options and settings.

Registry

Every `TopComponent` from the window system is centrally managed in a registry. The interface of this registry is specified by the `TopComponent.Registry` interface. An instance of this registry is obtained either directly via the `TopComponent` class:

```
TopComponent.Registry registry = TopComponent.getRegistry();
```

or via the `WindowManager`:

```
TopComponent.Registry registry = WindowManager.getDefault().getRegistry();
```

This registry will return, e.g., the currently activated `TopComponent`s via `getActivated()` or all opened `TopComponent`s via `getOpened()`. Further, a `PropertyChangeListener` can be registered to globally react to, e.g., state changes of a `TopComponent` (see Table 5-6).

Table 5-6. *Publicly accessing a TopComponent's state*

Property	Condition
`PROP_ACTIVATED`	If a `TopComponent` is being activated
`PROP_TC_CLOSED`	If a `TopComponent` has been closed
`PROP_TC_OPENED`	If a `TopComponent` has been opened

The code in Listing 5-22 adds a listener to the registry to react if a `TopComponent` is opened.

Listing 5-22. *Globally following changes of TopComponent states*

```
public class MyTopComponent
   extends TopComponent implements PropertyChangeListener {

   private MyTopComponent() {
      TopComponent.Registry reg = TopComponent.getRegistry();
      reg.addPropertyChangeListener(WeakListeners.propertyChange(this, reg));
   }

   public void propertyChange(PropertyChangeEvent evt) {
      if(evt.getPropertyName().equals(TopComponent.Registry.PROP_OPENED))
         // TopComponent opened
   }
}
```

Docking Container: Mode

The entire window system of the NetBeans Platform comprises sections, where multiple components are displayed docked in tabs. These are the previously mentioned editor and view sections. Such a section is called a mode. A mode as such is not a displayed component, but acts as controller and container for those components displayed therein. These components are of the type `TopComponent`. A mode is defined by the interface `Mode` from the Window System API.

Creating a Mode

A mode is not a fixed section, but can be defined individually via an XML file. Some important sections (e.g., the central editor section or the section where the NetBeans IDE usually opens the project view) are defined in NetBeans Platform modules. Any and all user-defined modes can be defined and added. The configuration file for a mode has the structure shown in Listing 5-23.

Listing 5-23. *Mode configuration file: MyMode.wsmode*

```
<!DOCTYPE mode PUBLIC
 "-//NetBeans//DTD Mode Properties 2.3//EN"
 "http://www.netbeans.org/dtds/mode-properties2_3.dtd">
<mode version="2.3">
   <module name="com.galileo.netbeans.module" spec="1.0"/>
   <name   unique="MyMode"/>
   <kind   type="view"/>
   <state  type="joined"/>
   <constraints>
      <path orientation="vertical" number="0" weight="0.2"/>
      <path orientation="horizontal" number="0" weight="1.0"/>
   </constraints>
   <empty-behavior permanent="true"/>
</mode>
```

First of all, a `module` element declares the module to which the mode belongs. Most important is the `name` element, whose value is a unique identifier and matches the file name. Additionally, the way the mode displays its components is specified in the `kind` element. There are three different kinds: `editor`, `view`, and `sliding`. Figure 5-8 displays the appearance of each kind via the NetBeans IDE.

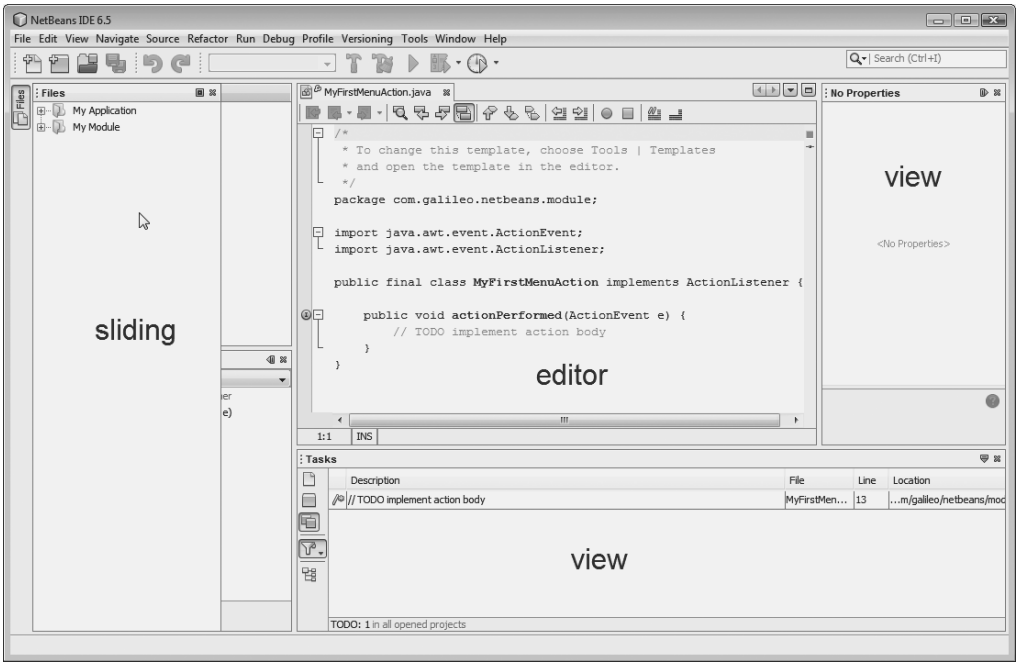


Figure 5-8. *Different types of modes*

A mode of the type `editor` is usually centrally arranged in the application. The `TopComponent`s arranged surrounding this `editor` mode are typically displayed in modes of the type `view`.

These windows are often called *helper windows*, because they offer features to, e.g., edit documents in the `editor` mode. Aside from differently displaying the tabs in the modes, the editor and view modes differ in that the `editor` type has control elements in the top-right corner for easier navigation between documents and `TopComponent`s.

Further, there is the type `sliding`. The window system moves or minimizes `TopComponent`s to the right, left, or bottom border of the application window. This is often useful when working with windows that are seldom or sporadically used. When hovering above the button of a minimized `TopComponent`, it opens above the opened windows and hides automatically when exiting the control element. Those windows are in a mode of the type `sliding`.

A mode of the type `sliding` additionally defines the element `slidingSide`. It selects borders upon which the mode is located. The following values are allowed:

```
<slidingSide side="left"/>
<slidingSide side="right"/>
<slidingSide side="bottom"/>
```

The element state defines whether the mode is docked in the application window or undocked in a separate window. Admissible values are `joined` for docked and `separated` for undocked display. When a `TopComponent` is undocked, its mode changes to `separated`.

The `constraints` element allows definition of dimension and position in relation to other modes. The preceding example would display the mode on the top border of the application window. Should it be on the bottom border, a bigger number (e.g., 30) is put into the attribute `number`. Since this number controls the position of all modes, it is helpful to take a look at the configuration files of the predefined modes for the NetBeans Platform. Some of them are in the module `Core-UI`.

This configuration file is added to the platform via the module's layer file. The `.wsmode` file is referenced in the folder `Windows2/Modes` (see Listing 5-24).

Listing 5-24. *Adding a new mode to the layer file*

```
<folder name="Windows2">
   <folder name="Modes">
      <file name="MyMode.wsmode" url="MyMode.wsmode"/>
      <folder name="MyMode">
         <file name="MyTopComponent.wstcref" url="MyTopComponentWstcref.xml"/>
      </folder>
   </folder>
</folder>
```

The `TopComponent` is added to the new mode via the TopComponent Reference file the wizard created. This allows for flexible declarative change in the arrangement of `TopComponent`s.

Maximize `TopComponent` by double-clicking the title bar in the application window. By default, all other components are changed to a `sliding` mode. If a component must stay in place and not move to the border, an attribute can be added to the corresponding TopComponent Reference file (`.wstcref`):

```
<docking-status maximized-mode="docked">
```

Modifying a Mode

At runtime, the user retains ability to move `TopComponent`s to different modes or change the dimension of a mode. The changes are stored in the user directory and restored upon restarting the application (as shown in "Configuration" subsection of the "Window System" section earlier in this chapter). Configurations are read from the layer file and the module's configuration files only if no data was stored. Invoke the Clean & Build Project command when changing configuration files during development, to refresh the user directory.

Groups of Windows: TopComponentGroup

Usually, more than one window is required for certain tasks. One such case is creation of a GUI inside the NetBeans IDE. The Inspector, Palette, and Properties windows are displayed. Upon leaving the Form Editor mode, these windows are hidden. The NetBeans Platform provides the ability to assemble `TopComponent`s into a *group* that enables toggling the visibility of them all. The Window System API provides the interface `TopComponentGroup` for this purpose. Though a group does not change the layout of the windows, nor the assembly or dimension of modes, it is responsible for opening and closing the groups' windows.

Groups manage windows in accordance with user settings, allowing the following cases:

- When a group is opened, all windows not already open will be opened, if the `open` attribute is set to `true`.
- Upon closing a group, all windows will be closed that were not open prior to opening the group, and their `close` attribute will be set to `true`. That means windows the user had open prior to opening the group will remain open.
- If a window of a group is closed by the user, the `open` attribute is set to `false` when closing the group. When the group is reopened, the window will not be opened.
- If during the time a group is open, the user opens a window from the group he previously closed, the `open` attribute is set to `true`, opening the window when the group is reopened.

The user influences the content of a group. The preceding describes the logic behind groups somewhat confusingly, but it is more easily understood with a bit of experimenting.

Creating a TopComponentGroup

Groups are defined via a Group Configuration file. They are declared in the layer file in the folder `Windows2/Groups`, announcing its existence to the platform. A configuration looks like Listing 5-25.

Listing 5-25. *The Group Configuration file: MyGroup.wsgrp*

```
<!DOCTYPE group PUBLIC
 "-//NetBeans//DTD Group Properties 2.0//EN"
 "http://www.netbeans.org/dtds/group-properties2_0.dtd">
<group version="2.0">
```

```
   <module name="com.galileo.netbeans.module" spec="1.0"/>
   <name unique="MyGroup"/>
   <state opened="false"/>
</group>
```

The optional `module` attribute declares the module groups belong to. The `name` attribute defines unique identifiers that must correspond to the file name. Whether the group is currently displayed is set in the `state` attribute. Creating the group in the layer file and referencing it to the Group Configuration file looks like Listing 5-26.

Listing 5-26. *Adding a group to the layer file*

```
<folder name="Windows2">
   <folder name="Groups">
      <file name="MyGroup.wsgrp" url="MyGroup.wsgrp"/>
      <folder name="MyGroup">
         <file name="MyTopComponent.wstcgrp" url="MyTopComponent.wstcgrp"/>
      </folder>
   </folder>
</folder>
```

Observe that this chapter's `TopComponent` was added to the newly created group. It was done by declaring a Group Reference Configuration file (`.wstcgrp`), where the behavior of the `TopComponent` inside the group is declared as shown in Listing 5-27.

Listing 5-27. *The Group Reference Configuration file: MyTopComponent.wstcgrp*

```
<!DOCTYPE tc-group PUBLIC
 "-//NetBeans//DTD Top Component in Group Properties 2.0//EN"
 "http://www.netbeans.org/dtds/tc-group2_0.dtd">
<tc-group version="2.0">
   <module name="com.galileo.netbeans.module" spec="1.0" />
   <tc-id id="MyTopComponent"/>
   <open-close-behavior open="true" close="true" />
</tc-group>
```

This file references a `TopComponent` via its unique identifier. The `TopComponent` must be declared in the layer file in the folder `Windows2/Components` with a `.settings` file, as occurs automatically when using the wizard from the "Window: TopComponent" section to create `TopComponent`s. Behaviors considered when opening and closing are defined. Those are attributes covered in the previous section.

For each window added to the group, create such a file and make an entry in the folder of your group in the layer file. The `WindowManager` obtains the group, as shown in Listing 5-28.

Listing 5-28. *Opening and closing a TopComponentGroup*

```
TopComponentGroup group =
   WindowManager.getDefault().findTopComponentGroup("MyGroup");
if(group != null) { /* group found */
   group.open();
}
```

Administration: WindowManager

The `WindowManager` is the central component of the window system. It manages modes, components, and groups, and provides an API to access its administrated components. The methods for locating components, as described in Table 5-7, are very useful.

Table 5-7. *Methods locating components from the window system*

Method	Description
`findMode(String name)`	Find a mode via its name.
`findMode(TopComponent t)`	Find the mode into which the `TopComponent` is docked.
`findTopComponent(String id)`	Find a `TopComponent` via its unique ID.
`findTopComponentID(TopComponent t)`	Get a `TopComponent`'s unique ID.
`findTopComponentGroup(String name)`	Find a `TopComponentGroup` via its name.

A `PropertyChangeListener` can be added to the `WindowManager` allowing notification events when, e.g., a mode is activated. Additionally, a set of all available modes in the window system can be obtained via a call to `getModes()`. The main application window is accessed via the following call:

```
Frame main = WindowManager.getDefault().getMainWindow();
```

Window System Architecture

The architecture of the window system classes is outlined in Figure 5-9.

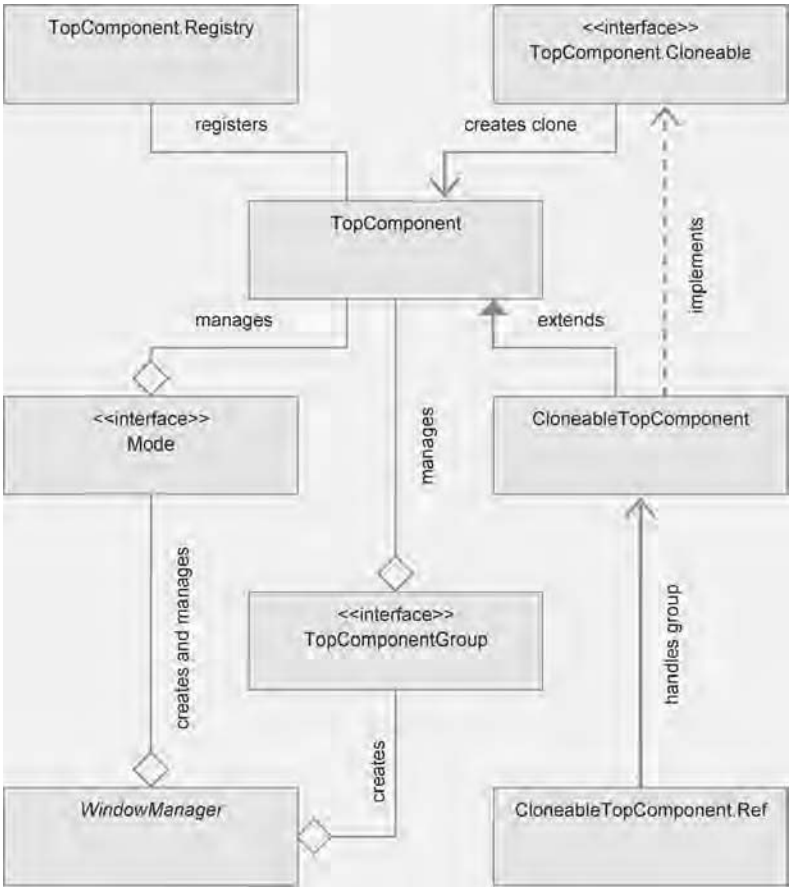


Figure 5-9. *Architecture of NetBeans window system*

Status Bar

The application window of the NetBeans Platform has an integrated status bar. It can be used over a global service class. Additionally, the status bar can be extended with your own components.

Using the Status Bar

The status bar is accessed via the abstract class `StatusDisplayer`. Obtain the default implementation via the method `getDefault()`. If there is none provided, the default NetBeans Platform status bar is returned. To provide a custom implementation of a status bar, see Chapter 6.

The method `setStatusText()` displays text on the status bar:

```
StatusDisplayer.getDefault().setStatusText("my first status");
```

To react to changes concerning the status bar, a `ChangeListener` can be registered with the status bar.

When working with a status bar, declare a module dependency to the module `UI Utilities`.

Extending the Status Bar

To extend the status bar, use the service interface `StatusLineElementProvider` from the UI Utilities API. The interface declares the method `getStatusLineElement()`, which returns the component added to the status bar.

The service implementation is accessed via, e.g., the folder `META-INF/services`. The service provider implementation is shown in Chapter 6. Setting the position of the component via the attribute `#position` is shown as well. The code in Listing 5-29 adds a clock to the status bar.

Listing 5-29. *Extending the status bar with a clock*

```
public class MyStatusLineClock
   implements StatusLineElementProvider {
   private static DateFormat format =
      DateFormat.getTimeInstance(DateFormat.MEDIUM);
   private static JLabel time = new JLabel(" " + format.format(new Date()) + " ");
   private JPanel panel = new JPanel(new BorderLayout());
   public MyStatusLineClock() {
      Timer t = new Timer(1000, new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            time.setText(" " + format.format(new Date()) + " ");
         }
      });
      t.start();
      panel.add(new JSeparator(SwingConstants.VERTICAL), BorderLayout.WEST);
      panel.add(time, BorderLayout.CENTER);
   }
   public Component getStatusLineElement() {
      return(panel);
   }
}
```

The implementation must be made known publicly to be found by the status bar. Create a file with the name of the interface in the folder `META-INF/services`:

```
org.openide.awt.StatusLineElementProvider
```

Add the name of the implementation to that file:

```
com.galileo.netbeans.module.MyStatusLineClock
```

This adds the clock to the Lookup by declaratively allowing the status bar to find and add it.

Progress Bar

By default, the NetBeans status bar has an integrated progress bar. It is used via the Progress API. There are classes available for visualizing the progress of simple tasks as well as monitoring multiple tasks that have their progress displayed as one. The progress of separate tasks can be monitored as well.

Displaying the Progress of a Task

There are three displays available for a progressing task (see Figure 5-10):

- A finite display of percentile progress until completion, if the number of required steps is known
- A finite display of remaining seconds until completion, if the number of required steps and their total duration are known
- An infinite display if neither the number nor the total duration of required steps are known

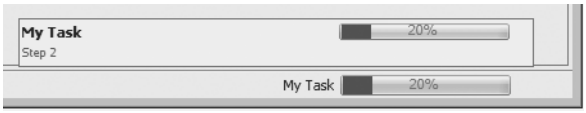


Figure 5-10. *Different types of progress bars*

The most basic use case entails the use of `ProgressHandleFactory`, creating an instance of `ProgressHandle` for a specific task (see Listing 5-30). The `ProgressHandle` provides control of the display of progress.

Listing 5-30. *Using the progress bar for separate tasks*

```
Runnable run = new Runnable() {
   public void run() {
      ProgressHandle p = ProgressHandleFactory.createHandle("My Task");
      p.start(100);
         // do some work
      p.progress("Step 1", 10);
         // do some more work
      p.progress(100);
      p.finish();
   }
};
Thread t = new Thread(run);
t.start(); // start the task and progress visualization
```

There are three different ways to start the display of a progress bar. These are outlined in Table 5-8.

Table 5-8. *Methods starting the different display types*

Method	Description
`start()`	Lets the progress bar run until a call to the `finish()` method is made
`start(int workunits)`	Displays the progress of execution in percentiles
`start(int workunits, long sec)`	Displays the remaining time in seconds

The methods shown in Table 5-9 allow switching between finite and infinite progress bars during runtime.

Table 5-9. *Methods changing the display type*

Method	Description
`switchToDeterminate(int workunits)`	Switches to percentile progress display
`switchToDeterminate(int workunits, long estimate)`	Switches to time progress display
`switchToIndeterminate()`	Switches to infinite mode

ALWAYS EXECUTE TASKS IN A SEPARATE THREAD!

A common error working with progress bars concerns the real task performed in the event dispatch thread (EDT) responsible for updating the GUI. Executing the task there blocks the thread, which in turn blocks the update of the GUI. Because of this, the EDT updates the progress bar when the task is finished. To separately execute the task, the `SwingWorker` class of the Java API can be used. Its use is shown via an asynchronous initialization in Chapter 17.

There are several methods for creating a `ProgressHandle` with `ProgressHandleFactory`. One of these allows passing the `Cancellable` service interface, allowing the user to abort the task with a button displayed next to the progress bar.

```
createHandle(String displayName, Cancellable allowToCancel)
```

The `suspend(String message)` method pauses the progress bar and displays a message.

Displaying the Progress of Many Related Tasks

Additionally, the Progress API provides an extended method for monitoring progress. An `AggregateProgressHandle` is created via the `AggregateProgressFactory`. With its help, you can assemble the progress of multiple tasks and display them in a single progress bar. The class `ProgressContributor` is additionally required. Every task requires an instance of it to communicate current progress to the `AggregateProgressHandle`.

The following example shows the use of this type of progress display. A number of tasks with different durations are created for execution and display in the progress bar.

The abstract class `AbstractTask` extending from `Thread` is created. This allows the parallel execution of tasks. Executing tasks sequentially requires not extending from `Thread`. This abstract class takes care of creating and managing the instance of `ProgressContributor` and communicating current progress.

```
public abstract class AbstractTask extends Thread {
   protected ProgressContributor p = null;
   public AbstractTask(String id) {
      p = AggregateProgressFactory.createProgressContributor(id);
   }
   public ProgressContributor getProgressContributor() {
      return p;
   }
}
```

The class `MyTask` (which creates an example task that takes ten steps to finish) is created. The `run()` method is implemented, in which the task is executed and progress communicated.

```
public class MyTask extends AbstractTask {
   public MyTask(String id) {
      super(id);
   }
   public void run() {
      p.start(10);
      // do some work
      p.progress(5);
      // do some work
      p.progress(10);
      p.finish();
   }
}
```

With the `MyTask2` class, another example task is created that takes more steps to finish than class `MyTask`.

```
public class MyTask2 extends AbstractTask {
   public MyTask2(String id) {
      super(id);
   }
   public void run() {
      p.start(30);
      // do some more work
      p.progress(2);
```

```
      // do some more work
      p.progress(15);
      p.finish();
   }
}
```

Located in the `MyProgram` class are a list of the tasks and the `processTaskList()` method to execute the tasks. The constructor creates a showcase of three tasks, adding them to the task list. By calling the method `processTaskList()` via, e.g., a button, an array of `ProgressContributor` is created, and every task's `ProgressContributor` is added to that array. The array is then passed to the `createHandle()` method of `AggregateProgressFactory`, creating an `AggregateProgressHandle`. When started, the progress bar is displayed and ready to receive progress notifications from the tasks. What remains is to start the tasks. The progress bar automatically terminates when the last task is finished.

```
public class MyProgram {
   private Vector<AbstractTask> tasks = new Vector<AbstractTask>();
   public MyProgram() {
      tasks.add(new MyTask("Task1"));
      tasks.add(new MyTask2("Task2"));
      tasks.add(new MyTask2("Task3"));
   }
   public void processTaskList() {
      ProgressContributor cps[] = new ProgressContributor[tasks.size()];
      int i = 0;
      for(AbstractTask task : tasks) {
         cps[i] = task.getProgressContributor();
         i++;
      }
      AggregateProgressHandle aph =
         AggregateProgressFactory.createHandle(
            "MyTasks", // displayed name
            cps,       // progress contributors
            null,      // not cancelable
            null);     // no output
      aph.start();
      for(AbstractTask task : tasks) {
         task.start();
      }
   }
}
```

If notification events of a task's execution are required, a monitor is passed to the instance of `AggregateProgressHandle`. Therefore, the interface `ProgressMonitor` has to be implemented (see Listing 5-31) and one of its instances passed to the `AggregateProgressHandle`.

Listing 5-31. *Examining tasks via a ProgressMonitor*

```
public class MyProgressMonitor implements ProgressMonitor {
   public void started(ProgressContributor pc) {
      System.out.println(pc.getTrackingId() + " started");
```

```
    }
    public void progressed(ProgressContributor pc) {
        System.out.println(pc.getTrackingId() + " progressed");
    }
    public void finished(ProgressContributor pc) {
        System.out.println(pc.getTrackingId() + " finished");
    }
}
AggregateProgressHandle aph = AggregateProgressFactory.create...
aph.setMonitor(new MyProgressMonitor());
```

Integrating a Progress Bar into Your Component

When integrating a progress bar into a component, `ProgressHandleFactory` and `AggregateProgressFactory` offer three methods to get a label with the name, a label with the details, and the progress bar for a certain `ProgressHandle` or `AggregateProgressHandle`:

```
JLabel     createMainLabelComponent(ProgressHandle ph)
JLabel     createDetailLabelComponent(ProgressHandle ph)
JComponent createProgressComponent(ProgressHandle ph)
```

Summary

In this chapter, we examined user interface design on the NetBeans Platform. We began with a birds-eye view of available components, the first of which was the menu bar. You added your own menu items to the menu bar and learned to hide those available by default. Similarly, you now understand that the toolbar can also be customized via configuration files.

Looking at the window system in detail, you can understand it to be the most important part of the user interface of the NetBeans Platform, and you'll continue to encounter it throughout this book. You learned what a `TopComponent` is and how to work with it. In this context, we looked at modes and `TopComponentGroups`.

Finally, the chapter discussed usage of the status bar and progress bar, as well as their related APIs.

CHAPTER 6

Lookup
Let's Talk to Other Modules!

Lookup is a concept that is as important as it is simple. Used in many places within NetBeans Platform applications, it allows modules to communicate with each other. This chapter shows how the concept works.

Functionality

Lookup is a central component and a commonly used concept in the NetBeans Platform for the management of object instances. Simplified, Lookup is a `Map`, with `Class` objects as keys and instances of those `Class` objects as values.

The main idea behind Lookup is decoupling components. It lets modules communicate with each other, which plays an important role in component-based systems, such as applications based on the NetBeans Platform. Modules provide objects via Lookup, as well as searching for or using objects.

The advantage of Lookup is its type safety, achieved by using `Class` objects instead of strings as keys. With this, the key defines the type of retrieved instance. So, it is impossible to request an instance whose type is unknown in the module. This pattern results in a more robust application, since errors like `ClassCastException` do not occur. Lookup is also used to retrieve and manage multiple instances for one key—i.e., of one type. This central management of specific instances is used for different purposes. Lookup is used to discover service providers, for which declarative adding and lazy-loading of instances is supported. In addition to this, you may pass instances via Lookup from one module to another, without the modules knowing each other. Intermodule communication is established. Even context-sensitive actions can be realized using the Lookup component.

To clear a common misconception, within a single application it's possible to have more than one Lookup. The most commonly used Lookup is global, provided by default in the NetBeans Platform. In addition, there are components, such as `TopComponent`, that have their own Lookup. These are local Lookups. As described in the "Intermodule Communication" section, it is possible to create your own Lookups and equip your components with a Lookup.

The Lookup concept is simple, efficient, and convenient. Once familiar with this pattern, it applies to many different areas. In the following sections, the usage of Lookup in its main use cases is shown.

Services and Extension Points

A main application of Lookup is the discovery and provision of services. The role of Lookup in this scenario is a function of a dynamic service locator, allowing separation of the service interface and the service provider. A module makes use of functionality without knowing anything about implementation. With this, loose coupling is achieved between modules.

By means of Lookup and a service interface, it is simple to define extension points for graphic components. A good example is the NetBeans status bar, defining the interface `StatusLineElementProvider`. With this interface and a service provider registration, the status bar is extended with user-defined components (an example for this is described in the "Status Bar" section of Chapter 5), without the status bar knowing about or having a dependency on those components.

For a dynamic and flexible provision and exchange of services, these are added declaratively to the Lookup, rather than programmed in the source code. This is achieved by either of two methods: adding the implementation of a service using a Service Provider Configuration file in the `META-INF/services` directory, or using the layer file of your module. Both are shown in the "Registering Service Providers" section later in the chapter.

The NetBeans Platform provides a global Lookup, which is retrieved using the static method `Lookup.getDefault()`. This global Lookup is used to discover services, added by using one of the available declarative registrations. Use this approach to register more than one implementation for a single service. The declarative registration allows instantiation of implementations on the first request. This pattern is known as *lazy-loading*.

To achieve a better understanding of this pattern for providing and requesting services, and to get a more practical perspective, we illustrate the creation of a search list for an MP3 file.

Defining the Service Interface

Module A is a module providing a user interface allowing the user to search for MP3 files by special search criteria. The search results are shown in a list. To remain independent of the search algorithm and ensure the dynamic use of multiple search variants (which may be switched at runtime), we specify the service interface `Mp3Finder` in module A. This service defines the search interface for MP3 files. The actual search algorithm is implemented in a separate module, B, provided via declarative registration.

Loose Service Provisioning

Module B is a service provider for implementation of the interface `Mp3Finder`. In this example, assume the module is searching for MP3 files in a database. This allows multiple implementations of the service provider to be registered. All implementations can be in either one or separate modules. To create an `Mp3DatabaseFinder` implementation of the interface `Mp3Finder` from module A, module B must define a dependency on module A. However, module A, the search list user interface, needs no dependency on module B. This is because Lookup provides the service based on the interface (living in module A as well) rather than the implementation (residing in module B). Thus, module A is completely independent of the implementation of the service (see Figure 6-1) and can use it transparently.

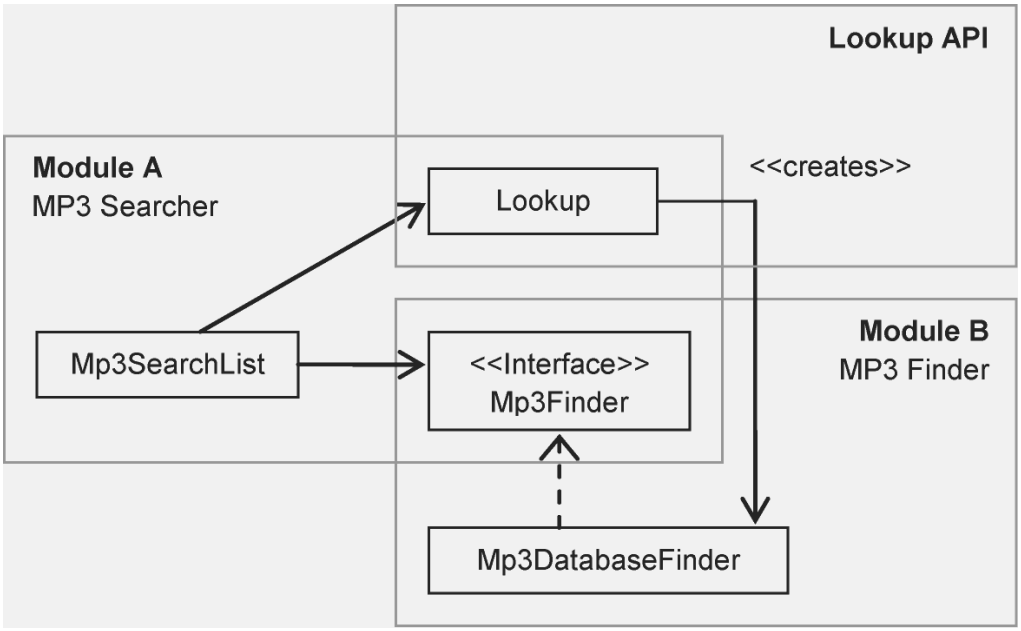


Figure 6-1. *Service Lookup pattern*

In module A, the service interface `Mp3Finder` is specified and a user interface is implemented for search and display of MP3 files (see Listing 6-1). A service provider is retrieved by passing the `Class` object of the interface `Mp3Finder` to Lookup, returning an instance matching the requested type. The interface `Mp3Finder` is also known as an extension point of module A. Any module can register implementations for it.

Listing 6-1. *Module A: MP3 searcher*

```
public interface Mp3Finder {
   public List<Mp3FileObject> find(String search);
}
public class Mp3SearchList {
   public void doSearch(String search) {
      Mp3Finder finder =
         Lookup.getDefault().lookup(Mp3Finder.class);
      List<Mp3FileObject> list = finder.find(search);
   }
}
```

Module B provides a service provider allowing the search of a database for MP3 files. This is done by implementing the interface `Mp3Finder`, specified by module A (see Listing 6-2). So, module B is an extension of module A at the extension point `Mp3Finder`.

Listing 6-2. *Module B: MP3 finder*

```
public class Mp3DatabaseFinder implements Mp3Finder {
   public List<Mp3FileObject> find(String search) {
      // search in database for mp3 files
   }
}
```

The newly created service provider must be registered, so it can be discovered with Lookup. This is done via a new file in the `META-INF/services` directory, named after the interface (`com.galileo.netbeans.modulea.Mp3Finder`) and using its fully qualified name. To associate an implementation with the interface, add a line to the file containing the fully qualified name of the implementation:

```
com.galileo.netbeans.moduleb.Mp3DatabaseFinder
```

Providing Multiple Service Implementations

It is useful to be able to register multiple MP3 search implementations. This is easy. Simply create further implementations of the interface `Mp3Finder`—for example

```
public class Mp3FilesystemFinder implements Mp3Finder {
   public List<Mp3FileObject> find(String search) {
      // search in local filesystem for mp3 files
   }
}
```

The name of this service provider is added to the previously created Service Provider Configuration file in `META-INF/services`:

```
com.galileo.netbeans.moduleb.Mp3FilesystemFinder
```

To use all registered implementations of a service, discovery of the services using Lookup must be adopted. Rather than using the `lookup()` method to retrieve a single implementation, use `lookupAll()` to retrieve all registered implementations of the service. Call the `find()` method of all discovered services as follows:

```
public class Mp3SearchList {
   public void doSearch(String search) {
      Collection<? extends Mp3Finder> finder =
         Lookup.getDefault().lookupAll(Mp3Finder.class);
      List<Mp3FileObject> list = new ArrayList<Mp3FileObject>();
      for(Mp3Finder f : finder) {
         list.addAll(f.find(search));
      }
   }
}
```

Ensuring Service Availability

A search module is of no use to the user if no search service is available allowing a search for MP3 files. To enable module A, ensuring that at least one implementation of a service is available, the NetBeans module system provides two attributes: `OpenIDE-Module-Provides` and `OpenIDE-Module-Requires`, which allow definition in the manifest file of a module if a special service implementation is provided or required. These and further attributes of the manifest file are described in more detail in the "Module Manifest" section of Chapter 3.

Within the manifest file of module A, the existence of at least one provider of the `Mp3Finder` service is required, with the following entry:

```
OpenIDE-Module-Requires: com.galileo.netbeans.modulea.Mp3Finder
```

To inform the module system during loading of the modules that module B provides the service `Mp3Finder`, add the following entry to the manifest file of module B:

```
OpenIDE-Module-Provides: com.galileo.netbeans.modulea.Mp3Finder
```

If no module declares such an entry in its manifest file (i.e., there is no service provider available), the module system announces an error and does not load module A.

Global Services

Global services—i.e., services that can be used by multiple modules and that are only provided by one module—are typically implemented using abstract (singleton) classes. With this pattern, the services manage the implementation on their own and provide an additional trivial implementation (as an inner class) in case there is no other implementation registered in the system. This has the advantage that the user always gets a valid reference to a service and never a `null` value.

An example would be an MP3 player service (see Listing 6-3) used by different modules—e.g., a search list or playlist. The implementation of the player is exchangeable.

Listing 6-3. *MP3 player as a global service in the MP3 Services module*

```
public abstract class Mp3Player {
   public abstract void play(Mp3FileObject mp3);
   public abstract void stop();
   public static Mp3Player getDefault() {
      Mp3Player player = Lookup.getDefault().lookup(Mp3Player.class);
      if(player == null) {
         player = new DefaultMp3Player();
      }
      return(player);
   }
```

```
    private static class DefaultMp3Player extends Mp3Player {
        public void play(Mp3FileObject mp3) {
            // send file to an external player or
            // provide own player implementation or
            // show a message that no player is available
        }
        public void stop() {}
    }
}
```

This service, implemented as an abstract class, specifies its interface via the abstract methods, and at the same time provides access to the service via the static method `getDefault()`. The advantage of this pattern is that there is no need for users of the service to know anything about the Lookup API. This keeps the application logic lean, as well as independent from Lookup API.

The abstract class should normally be part of a module, which is, in turn, part of the standard distribution of the application (in the example, this would be the `MP3 Services` module). The service provider (i.e., the classes that contain the real code for playing MP3 files) can be encapsulated in a separate module (see Listing 6-4). In the example, this is the class `MyMp3Player`, for which we subsequently create a skeleton and add it to module C.

Listing 6-4. *MP3 player service provider in the MP3 Player module*

```
public class MyMp3Player extends Mp3Player {
    public void play(Mp3FileObject mp3) {
        // play file
    }
    public void stop() {
        // stop player
    }
}
```

Now the `MyMp3Player` service provider must be registered. This is done—as shown in the previous section—via a Service Provider Configuration file in the name `com.galileo.netbeans.mp3services.Mp3Player` in the `META-INF/services` directory (see the "Registering Service Providers" section) with the following content:

```
com.galileo.netbeans.mp3player.MyMp3Player
```

The relationships and dependencies of the modules are shown in Figure 6-2.

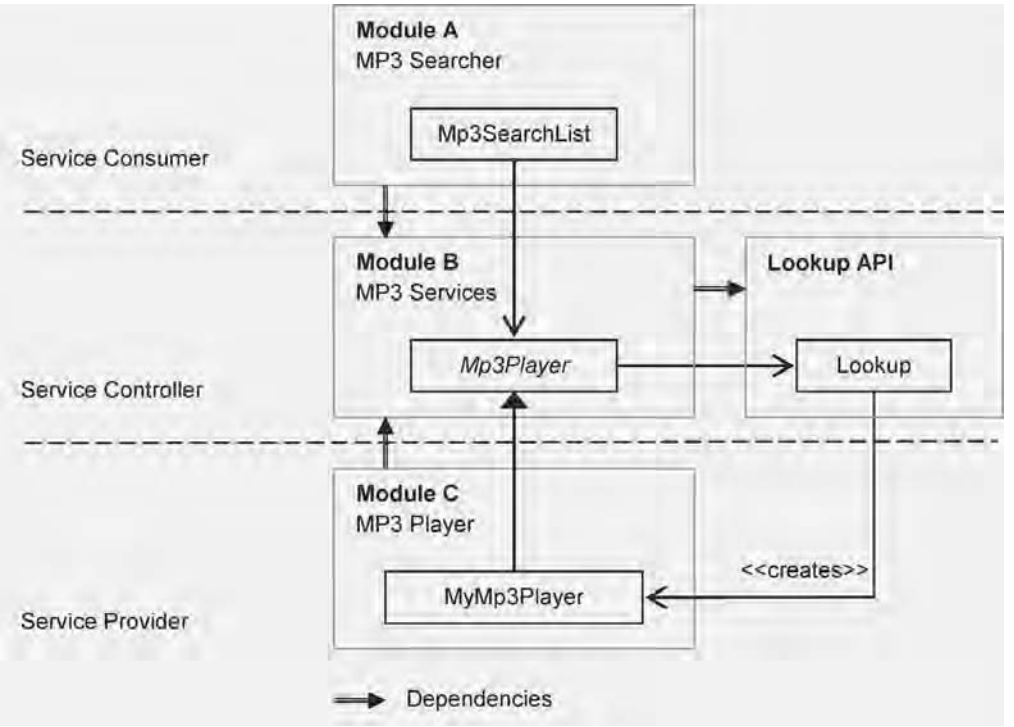


Figure 6-2. *Dependencies and relationships of global service, service provider, and application module*

Good examples for global services inside the NetBeans Platform are `StatusDisplayer` and `IOProvider`. The class `IOProvider` grants access to the Output window. The service provider actually writing the data to the Output window is in a separate class, `NbIOProvider`, in a separate module. If the module is available and the service provider registered, its implementation is retrieved via the static method `IOProvider.getDefault()`. If the module is not available, the default implementation is provided, which writes the output data to the default output (`System.out` and `System.err`).

Registering Service Providers

To allow a dynamic and flexible registration of service providers, even after delivering the application, and to ensure those are loaded only if needed, the registration is done declaratively, using configuration files.

Services available inside a NetBeans Platform–based application and accessible via Lookup, and can be registered using two different mechanisms. Both will be shown in detail in the following sections.

Service Provider Configuration File

The preferred method of registration of service providers is using a Service Provider Configuration file. This approach is part of the Java JAR File Specification. A file is named after its service and lists in its content all service providers. The file must be placed in the `META-INF/services` directory, which is part of the `src/` directory of a module, or must be part of the classpath of a module.

`src/META-INF/services/com.galileo.netbeans.module.Mp3Finder`

```
com.galileo.netbeans.module.Mp3DatabaseFinder
com.galileo.netbeans.module.Mp3FilesystemFinder
```

In this example, two service providers are registered for the service (i.e., the interface of abstract class, `Mp3Finder`). The provided services and providers are discovered using the META-INF services browser of the NetBeans IDE. This feature is part of the project view for NetBeans module projects. The information is shown under Important Files ➤ META-INF services. There, the services registered within your module are listed in `<exported services>`. `<all services>` lists all services and service providers available inside the Module Suite (i.e., those registered by NetBeans Platform modules, as well as independently created modules). Using this view, it is possible to get an overview of services provided by the NetBeans Platform. Additionally, you can use the context menu of an existing service to add another provider for this service.

The global Lookup (i.e., the standard Lookup) discovers the services in the `META-INF/services` directory and instantiates the providers. A successful service instantiation requires that each service provider have a default constructor so that creation from Lookup is possible.

Based on the original specification of the Service Provider Configuration file, the NetBeans Platform provides two add-ons, allowing the removal of existing service providers or changing the order of the registered providers. To make these additions comply with the original Java specification, the add-ons are prefixed with the comment sign (`#`). So, these lines are ignored by JDK implementations.

Removal of a Service Provider

It is possible to remove a service provider registered within another module. This feature can be used to substitute the standard implementation of a service of the NetBeans Platform with another implementation.

A service provider is removed by adding the following entry in your Service Provider Configuration file. At the same time, you can provide your own implementation.

```
# remove the other implementation (by prefixing the line with #-)
#-org.netbeans.core.ServiceImpl
# provide my own
com.galileo.netbeans.module.MyServiceImpl
```

Order of Service Providers

The order in which service providers are returned from Lookup is controlled using a `position` attribute for each provider entry.

For example, this is necessary to control the order of additional entries in the status bar (see the "Status Bar" section in Chapter 5) or to ensure that your own implementation is called before the NetBeans Platform implementation. Optionally, you can specify a negative value for the `position` attribute. The NetBeans Platform orders instances by ascending positions, so that instances with smaller numbers are returned before instances with larger numbers. For that purpose, the following entry is added to the Service Provider Configuration file:

```
com.galileo.netbeans.module.MyServiceImpl
#position=20
com.galileo.netbeans.module.MyImportantServiceImpl
#position=10
```

We recommend assigning position values in larger intervals, as shown in the example. This simplifies adding further implementations later on.

Services Folder

Another way to provide a service implementation is registration using the `Services` folder in the module layer file, as shown in Listing 6-5.

Listing 6-5. *Registration of service providers in a layer file*

```
<folder name="Services">
  <folder name="Mp3Services">
    <file name="com-galileo-netbeans-module-Mp3DatabaseFinder.instance">
      <attr name="instanceOf" stringvalue="com.galileo.netbeans.module.Mp3Finder"/>
    </file>
  </folder>
</folder>
```

If a service is requested using the default Lookup, implementations are discovered by searching the `Services` folder and its subdirectories for instances, which can be assigned to the requested service interface. So, services can be grouped using arbitrary folders, as shown with the folder `Mp3Services` in our example.

In contrast to the registration using the Service Provider Configuration file, the service provider need not provide a default constructor if registered in the layer file. With the layer file, specifying a static method in the `instanceCreate` attribute is possible, creating an instance of the service provider. Let's assume the already created provider `Mp3DatabaseFinder` has a static method `getDefault()` that returns the instance. The declaration can be changed by adding the following attribute:

```
<attr name="instanceCreate"
        methodvalue="com.galileo.netbeans.module.Mp3DatabaseFinder.getDefault"/>
```

With this attribute declaration, the service provider instance is not created using the default constructor, but rather by calling the static method `getDefault()` (more detailed information regarding this attribute and the corresponding `.instance` files are described in Chapter 3).

Also, using the registration via the `Services` folder allows removing existing service providers and controlling the order of the providers. Both mechanisms are achieved using default features of the layer file. A service provider can be removed by adding the suffix `_hidden` to its name, as is done for menu entries (see the "Menu Bar" section in Chapter 5).

```
<file name="com-galileo-netbeans-module-ServImp.instance_hidden">
```

The order in which service providers are returned is controlled using the `position` attribute, which is the same strategy as used for other entries in the layer file (see Chapter 3).

```
<folder name="Services">
   <file name="com-galileo-netbeans-module-ServImp.instance">
      <attr name="position" intvalue="10"/>
   </file>
   <file name="com-galileo-netbeans-module-ServImp2.instance">
      <attr name="position" intvalue="20"/>
   </file>
</folder>
```

In this example, the `position` attributes ensure that the service provider `ServImp` will be returned before `ServImp2`.

Intermodule Communication

Beside the global Lookup, which is provided by the NetBeans Platform and allows access to all registered services, a local Lookup to your own components may be added. The Lookup API offers a factory to create Lookups and an opportunity to listen to changes in Lookups. Using the class `ProxyLookup`, a user can create a proxy combining multiple Lookups into one. Using this feature of the Lookup API and SPI, we create communication between components of different modules without making them interdependent.

A typical use case for the communication of loosely coupled modules is the visualization of detailed information for a selected object. The selection of objects and visualization of information is done in separate modules. As an example, imagine a list displaying the search results for MP3 files. Selecting an entry in the list provides the selected entry via Lookup, so other parts of the application can access the entry and display the required detailed information. This pattern is similar to the observer pattern. The module providing the objects—in this case the search list—is the subject, and the information display module is the observer. This allows multiple modules to display the data or detailed information in various ways. Again, the advantage is loose coupling of the modules: they are completely independent of each other. The only thing they have in common is the provided object (or to be more precise its interface), which is the source of the information to be processed. This loose coupling is achieved by using a proxy object, which acts as a substitute for the subject in the registration process of the observer. So, the observer is registered with the proxy component (in our case the Lookup), not the subject.

Figure 6-3 shows the example implemented in the following paragraphs. Both windows are in a separate module, each independent of the other (i.e., both can be exchanged or new ones can be added arbitrarily).

Figure 6-3. *Typical application example of a data exchange between two modules, without interdependency*

The structure of this concept is shown in Figure 6-4. The class `Mp3SearchList` in module A represents a list of search results. A search result entry is represented by the class `Mp3FileObject`, residing in a separate module, since this class is the most common denominator of all modules. If an entry is selected in the list, the `Mp3FileObject` instance is added to the local Lookup. A moderator (i.e., a proxy component depicted as the interface `ContextGlobal-Provider`) is needed to decouple modules A and B. This proxy component provides the local Lookup of module A to module B, which contains the currently selected instance. To enable the centralized proxy component to access the local Lookup of the class `Mp3SearchList`, the Lookup API provides the interface `Lookup.Provider`. This interface must be implemented from the class `Mp3SearchList`.

Using the method `getLookup()`, the local Lookup can be obtained. The `Lookup.Provider` interface is already implemented by the class `TopComponent`, which is the superclass of all visible NetBeans window system components, as well as the `Mp3SearchList`. The NetBeans window system already provides an instance of the central proxy component, the class `Global-ActionContextImpl`. This class provides a proxy Lookup, which accesses the local Lookup of the focused `TopComponent`. This Lookup can be obtained by calling the static utility method `Utilities. actionsGlobalContext()`. So, there is no need to create our own `ContextGlobal-Provider` instance, but we already have access to the global proxy Lookup. If you are interested

in more details and want to know more about this concept, it may be worthwhile to investigate the sources for the mentioned classes and methods.

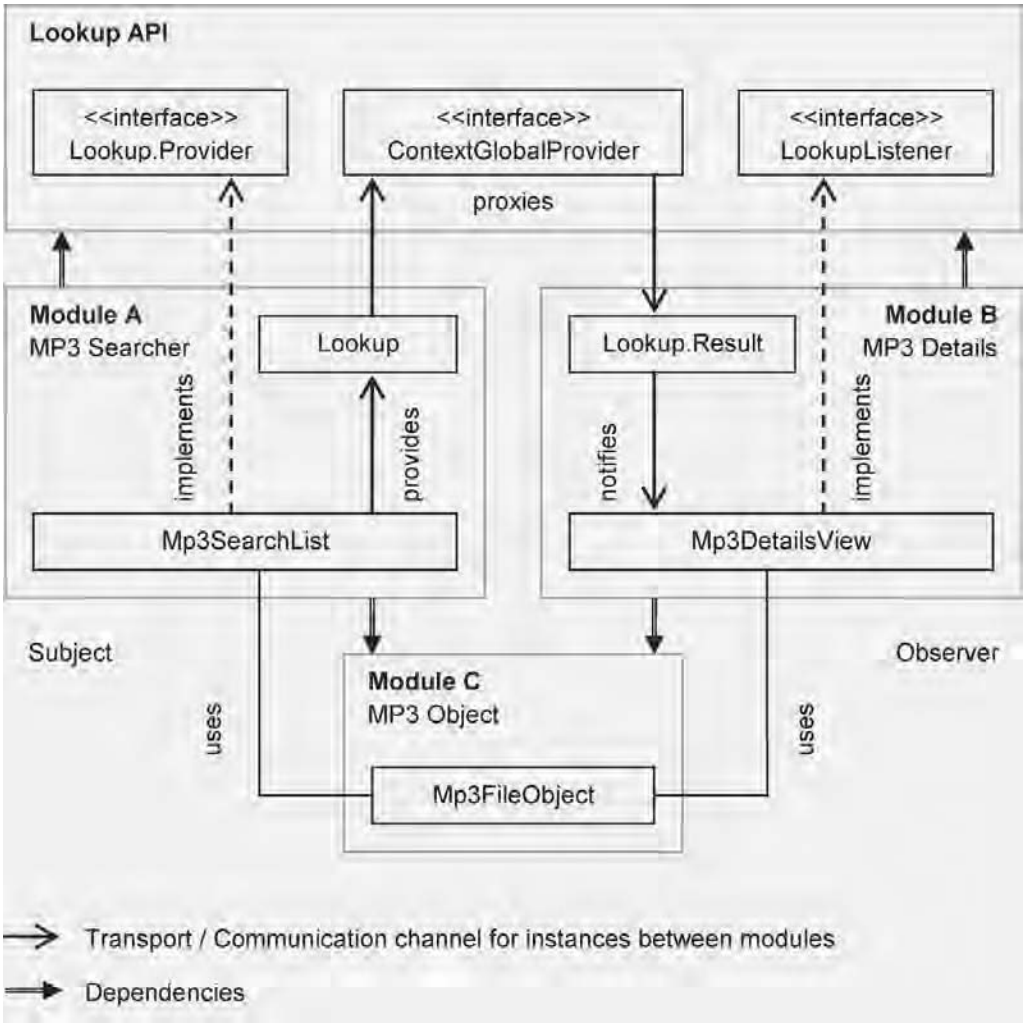


Figure 6-4. *Structure of the intermodule communication concept using a local Lookup via a proxy component to decouple subject and observer*

The class `Mp3DetailsView` gains access to the local Lookup of the `Mp3SearchList` by calling `Utilities.actionsGlobalContext()`. Based on the global Proxy Lookup, we create a `Lookup.Result` for the class `Mp3FileObject`. An instance of the class `Lookup.Result` provides a subset of a Lookup for a special class. The main advantage is that the user can listen for changes in this subset by using a `LookupListener`. So, the component will be notified as soon as another `Mp3FileObject` is selected in the `Mp3SearchList`, or if the window showing the `Mp3SearchList` loses focus. As an example, no detailed MP3 information will be displayed.

Following, you find the classes of this example application. Only the important parts of the classes are shown.

First, we have the class `Mp3SearchList`, which represents a window, and because of this, extends from the base class `TopComponent`. To enable listening to selection changes in the result list, we also implement the `ListSelectionListener` interface. As a private member, we have a data model that manages the data in the table. For demonstration purposes, a simple data model has been chosen, creating three example objects of the class `Mp3FileObject` in the constructor and adding them to the model. This data would normally be provided using the search algorithm. The second private member object is an instance of `InstanceContent`. This enables us to dynamically change the content of the Lookup. In the constructor of the `Mp3SearchList`, we can now create a local Lookup, using the class `AbstractLookup` and passing our `InstanceContent` object into its constructor. Using the method `associateLookup()`, our local Lookup is set as the Lookup of the `TopComponent`, so that it will be returned from the `getLookup()` method.

In the method `valueChanged()`, which gets called if a data set is selected in the table, we get the data set from the data model, wrap it into a collection, and pass it to our `InstanceContent` instance (see Listing 6-6), which is the data storage for the Lookup. So, the selected element is always part of the local Lookup.

Listing 6-6. *Mp3SearchList displays the search results in a table and adds the actual selected data set to the local Lookup.*

```
public class Mp3SearchList extends TopComponent implements ListSelectionListener {
   private Mp3SearchListModel model = new Mp3SearchListModel();
   private InstanceContent content = new InstanceContent();
   private Mp3SearchList() {
      initComponents();
      searchResults.setModel(model);
      searchResults.getSelectionModel().addListSelectionListener(this);
      associateLookup(new AbstractLookup(content));
   }
   public void valueChanged(ListSelectionEvent event) {
      if(!event.getValueIsAdjusting()) {
         Mp3FileObject mp3 = model.getRow(searchResults.getSelectedRow());
         content.set(Collections.singleton(mp3), null);
      }
   }
}
```

Here, the data model `Mp3SearchListModel` of the table with the search results is just an example and is kept quite simple (see Listing 6-7). Three objects of the type `Mp3FileObject` are directly created in the constructor.

Listing 6-7. *Simplified data model managing and providing the data for the result list*

```
public class Mp3SearchListModel extends AbstractTableModel {
   private String[] columns = {"Artist", "Title", "Year"};
   private Vector<Mp3FileObject> data = new Vector<Mp3FileObject>();
   public Mp3SearchListModel() {
      data.add(new Mp3FileObject("Gigi D'Agostino", "The rain", "2006"));
```

```
        data.add(new Mp3FileObject("Marquess", "El temperamento", "2006"));
        data.add(new Mp3FileObject("Floorfilla", "Cyberdream", "2006"));
    }
    public Mp3FileObject getRow(int row) {
        return(data.get(row));
    }
    public Object getValueAt(int row, int col) {
        Mp3FileObject mp3 = data.get(row);
        switch(col) {
            case 0: return mp3.getArtist();
            case 1: return mp3.getTitle();
            case 2: return mp3.getYear();
        }
        return "";
    }
}
```

The class `Mp3DetailsView` is the window showing detailed information of the selected entry of the `Mp3SearchList`. To get notification of changes in the Lookup—e.g., in case of selection changes—the `LookupListener` interface is implemented. A `Lookup.Result`, which enables us to react to changes for a specific type (in our case `Mp3FileObject`), is used as a private member. Opening a window triggers the method `componentOpened()`. We use this callback to obtain the Lookup of the proxy component, using the method `Utilities.actionsGlobalContext()`, which returns a Lookup that always delegates to the local Lookup of the active `TopComponent`. Based on this Proxy Lookup, we now create a `Lookup.Result` for the type `Mp3FileObject` and register a `LookupListener` to listen to changes on this result. If a `TopComponent` now gains the focus, which has one or more instances of this type in a local Lookup, the method `resultChanged()` gets called. With this, we only need to retrieve the instances and display the information accordingly, as shown in Listing 6-8.

Listing 6-8. *The window Mp3DetailsView shows the information of the Mp3FileObject, which is selected in the Mp3SearchList.*

```
public class Mp3DetailsView extends TopComponent implements LookupListener {
    private Lookup.Result<Mp3FileObject> result = null;
    private Mp3DetailsView() {
        initComponents();
    }
    public void componentOpened() {
        result = Utilities.actionsGlobalContext().lookupResult(Mp3FileObject.class);
        result.addLookupListener(this);
        resultChanged(null);
    }
    public void resultChanged(LookupEvent event) {
        Collection<? extends Mp3FileObject> mp3s = result.allInstances();
        if(!mp3s.isEmpty()) {
            Mp3FileObject mp3 = mp3s.iterator().next();
            artist.setText(mp3.getArtist());
            title.setText(mp3.getTitle());
            year.setText(mp3.getYear());
        }
```

```
    }
}
```

The information provided via `Mp3SearchList` and displayed using `Mp3DetailsView` is part of the class `Mp3FileObject` (see Listing 6-9). This class should be implemented in a separate module to achieve the best possible encapsulation and reuse. In this example, it is module C. To grant modules A and B access to this class, they must declare a dependency on module C. If the class `Mp3FileObject` is provided only via module A, it is possible to move the class to module A.

Listing 6-9. *Mp3FileObject provides the data*

```
public class Mp3FileObject {
   private String artist = new String();
   private String title  = new String();
   private String year   = new String();
   public Mp3FileObject(String artist, String title, String year) {
      this.artist = artist;
      this.title  = title;
      this.year   = year;
   }
   public String getArtist() {
      return this.artist;
   }
   public String getTitle() {
      return this.title;
   }
   public String getYear() {
      return this.year;
   }
}
```

As a proxy component, we use the global Proxy Lookup provided by the NetBeans Platform, which delegates to the local Lookup of the active `TopComponent`. In Figure 6-4, this is depicted with the interface `ContextGlobalProvider`. This global proxy Lookup is easily substituted by another implementation. This implementation has only to provide the local Lookup of the component containing the subject to the observer.

Java Service Loader

Since Java 6, an API is available similar to Lookup: `ServiceLoader`. This class loads service providers, which are registered over the `META-INF/services` directory. With this functionality, the API equals the NetBeans standard Lookup that can be obtained using `Lookup.getDefault()`. A `ServiceLoader` is created for a special type using the `Class` object of the service interface or the abstract service class. A static factory method is used for creating a `ServiceLoader` instance. Depending on the classloader used to load the service providers, three methods for creating service loaders are available.

By default, service providers are loaded using the context classloader of the current thread. Inside the NetBeans Platform, this is the system classloader (for more details on the NetBeans

classloader system, see Chapter 2). This allows the user to load service providers from all modules. Such a service loader is created with the following call:

```
ServiecLoader<Mp3Finder> s = ServiceLoader.load(Mp3Finder.class);
```

You may want to use a special classloader to load service providers—e.g., a module classloader to restrict loading of service providers to classes from your own module. To obtain such a `ServiceLoader`, the classloader to be used is passed to the factory method:

```
ServiceLoader<Mp3Finder> s = ServiceLoader.load(
   Mp3Finder.class, this.getClass().getClassLoader());
```

In addition to this, it is possible to create a service loader that only returns installed service providers—e.g., a service provider from JAR archives located in the `lib/ext` directory or in the platform-specific extension directory. Other service providers found on the classpath are ignored. This service loader is created using the `loadInstalled()` method:

```
ServiceLoader<Mp3Finder> s = ServiceLoader.loadInstalled(Mp3Finder.class);
```

The service provider can be obtained using an iterator. The iterator triggers dynamic loading of the provider on first access. The loaded providers are stored in a local cache. The iterator returns the cached providers before loading the remaining previously unloaded providers. If necessary, the internal cache can be cleared using the method `reload()`. This ensures that all providers are reloaded.

```
Iterator<Mp3Finder> i = s.iterator();
if(i.hasNext()) {
   Mp3Finder finder = i.next();
}
```

Summary

In this chapter, you learned one of the most interesting and important concepts of the NetBeans Platform: Lookup. We examined functionality of Lookups and you became familiar with the service interfaces and service providers. You learned to create service interfaces and use them within service providers, as well as how service providers are discovered in a loosely coupled way. To that end, we began to use the various registration mechanisms.

However, Lookups do a lot more than simply discover services. In fact, they also function to enable intermodular communication. We looked at an example, showing how information is shared between windows without them knowing about each other. Finally, we broadened our exploration of this topic by relating it to the JDK 6 `ServiceLoader` class.

CHAPTER 7

File Access and Display

Let's Use the NetBeans Platform to Work with Files!

This chapter illustrates concepts and APIs used to manage, manipulate, and represent data. Upon completion, you will be able to handle and display all forms of data in a professional way. The NetBeans Platform uses the same approach for its internal data, such as the data in the System Filesystem.

Overview

The NetBeans Platform provides comprehensive solutions for creating, managing, manipulating, and presenting data. Solutions are provided by the File Systems, Data Systems, and Nodes APIs.

Each of these APIs can be found on its own abstraction layer. In combination with specific data outside a NetBeans Platform application, this system is divided into four layers, as shown in Figure 7-1.

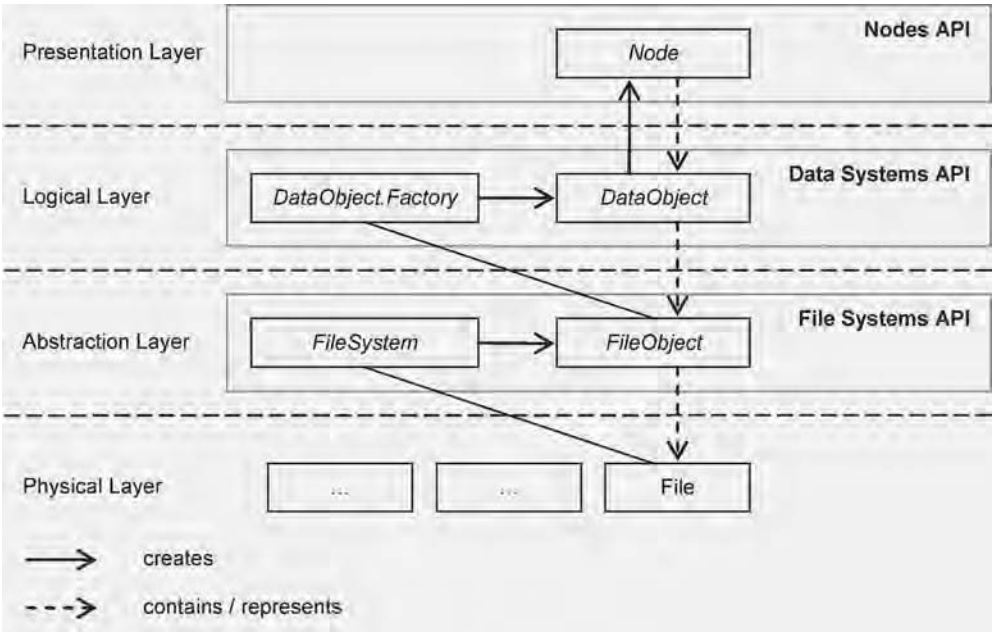


Figure 7-1. *The data representation architecture of the NetBeans Platform*

A data system is initially abstracted using the `FileSystem` class. The `FileSystem` class lets users address physical data from different sources in the same way. Examples of this are the local file system, a file system organized as an XML file (the System Filesystem is built accordingly—see Chapter 3), or even a JAR file.

To use it, simply provide the desired implementation of the abstract `FileSystem` class. This is how the File Systems API abstracts specific data and provides a virtual file system giving shared data access to the whole application.

All access is, in that way, completely independent of its origin. However, abstracted data on the abstraction layer in the form of a `FileObject` class does not contain any information about the kind of data it handles. Moreover, this layer contains no data-specific logic.

Just above this layer is found the Data Systems API, on the logical layer. That locates objects representing data of specific types and is built on top of the `DataObject` class. A `DataObject.Factory` creates objects for every desired data type. The upper layer in this concept is the Nodes API. This API is located on the presentation layer. A node is responsible for type-specific representation of data in the user interface. As a result, a node represents a `DataObject` and is responsible for the creation of the presentation layer.

File Systems API

The NetBeans Platform gives transparent access to files and folders by providing the File Systems API. This API allows abstract access to files that are always handled in the same way,

independently, as if the data is stored in the form of a virtual XML file system, such as that used by the System Filesystem, a JAR file, or in a normal folder.

General interfaces of a file system are specified in the abstract class `FileSystem`. The abstract class `AbstractFileSystem` is used as a superclass for specific file system implementations. The specific implementations `LocalFileSystem`, `JarFileSystem`, and `XMLFileSystem` use this class as their superclass (see Figure 7-2). The class `MultiFileSystem` further provides a proxy for multiple file systems and is also used here as superclass.

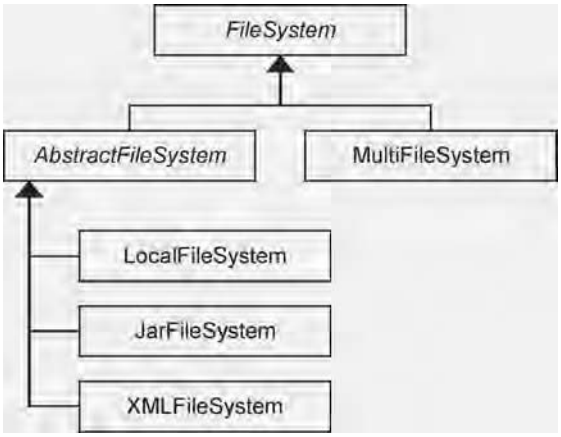


Figure 7-2. *Hierarchy of the FileSystem classes*

Data inside the file system, such as folders and files, is represented by a `FileObject` class. This is an abstract wrapper class for the standard `java.io.File` class. The implementation of a `FileObject` is provided by the specific file system. The `FileObject` class provides, along with these standard operations, the ability to listen to changes to folders and files.

Operations

Following is an outline of operations provided by the `FileObject` class.

Obtaining

To obtain a `FileObject` for an existing file on the local file system, use the helper class `FileUtil`:

```
FileObject obj = FileUtil.toFileObject(new File("C:/file.txt"));
```

To create a `FileObject` from a specific `FileSystem` object, call the `findResource()` method, using the full path:

```
FileSystem fs = Repository.getDefault().getDefaultFileSystem();
FileObject obj = fs.findResource("folder/file");
```

This is the manner in which you create a `FileObject` for data located in the System Filesystem, which is an `XMLFileSystem`.

Creating

The following allows you to create new files or folders using `FileObjects`:

```
File file   = new File("E:/newfolder/newfile.txt");
File folder = new File("E:/newfolder2");
FileObject fo1 = FileUtil.createData(file);
FileObject fo2 = FileUtil.createFolder(folder);
```

If you already have a `FileObject` folder, create a file or folder in the corresponding file system as follows:

```
FileObject folder  = ...
FileObject file = folder.createData("newfile.txt");
FileObject subfolder = folder.createFolder("newfolder");
```

Renaming

To rename a folder or file, make sure someone else is not editing it at the same time, by using a `FileLock` object. This lock is released after renaming, using a `finally` block.

```
FileObject myfile = ...
FileLock lock = null;
try {
   lock = myfile.lock();
} catch (FileAlreadyLockedException e) {
   return;
}
try {
   myfile.rename(lock, "newfilename", myfile.getExt());
} finally {
   lock.releaseLock();
}
```

Deleting

Deleting folders or files is straightforward. The `delete()` method takes care of reserving and releasing a `FileLock`. Consequently, deleting only requires the following line:

```
FileObject myfile = …
myfile.delete();
```

A variant of the `delete()` method is available that enables users to pass their own file `FileLock`, analogous to the renaming of a `FileObject`.

Moving

A `FileObject` cannot be moved in the same way as a `File`, simply by renaming. Instead, this is achieved by using the method `moveFile()`, provided by the class `FileUtil`. It allows moving

FileObjects and handles copying files or folders to the destination folder, deleting the source and automatically allocating and releasing the required FileLock objects.

```
FileObject fileToMove = ...
FileObject destFolder = ...
FileUtil.moveFile(fileToMove, destFolder, fileToMove.getName());
```

Reading and Writing Files

Reading and writing FileObjects is done, as usual in Java, by streams. The FileObject class provides the methods InputStream and OutputStream for this purpose. We wrap these for reading in a BufferedReader and for writing in a PrintWriter, as shown in Listing 7-1.

Listing 7-1. *Reading and writing a FileObject*

```
FileObject myFile = ...
BufferedReader input = new BufferedReader(
   new InputStreamReader(myFile.getInputStream()));
try {
   String line = null;
   while((line = input.readLine()) != null) {
      // process the line
} finally {
   input.close();
}
PrintWriter output = new PrintWriter(
   myFile.getOutputStream());
try {
   output.println("the new content of myfile");
} finally {
   output.close();
}
```

You optionally pass your own FileLock to the method getOutputStream().

Monitoring Changes

To monitor changes, register a FileChangeListener for the FileObject that responds to data changes inside the file system (see Listing 7-2).

Listing 7-2. *Responding to changes to a DataObject*

```
File file = new File("E:/NetBeans/file.txt");
FileObject fo = FileUtil.toFileObject(file);
fo.addFileChangeListener(new FileChangeListener(){
   public void fileFolderCreated(FileEvent fe) {
   }
   public void fileDataCreated(FileEvent fe) {
   }
   public void fileChanged(FileEvent fe) {
   }
```

```
    public void fileDeleted(FileEvent fe) {
    }
    public void fileRenamed(FileRenameEvent fre) {
    }
    public void fileAttributeChanged(FileAttributeEvent fae) {
    }
});
```

The methods `fileFolderCreated()` and `fileDataCreated()`, called when a file or folder is created, only make sense when the monitored `FileObject` is a folder.

While changing files, the event is always processed for the file itself, as well as its parent directory. This means that you will be informed of changes of files, even when only monitoring the parent directory. If you are not interested in all the events of the `FileChangeListener` interface, use the adapter class `FileChangeAdapter` instead.

ONLY INTERNAL EVENTS ARE MONITORED

You are only informed about events processed on the specific `FileObject` inside an application. It is impossible to react to changes made outside your application, such as renaming a file with Windows Explorer.

The classes `FileSystem`, `FileObject`, and `FileUtil` provide several additional and helpful methods. It is worth having a closer look at the documentation of the File Systems API.

Data Systems API

The Data Systems API provides a logical layer on top of the File Systems API. While a `FileObject` manages its data regardless of type, a `DataObject` is a wrapper for a `FileObject` of a quite specific type. A `DataObject` extends a `FileObject` with type-specific features and functionalities. These are specified through interfaces or abstract classes: so-called *cookies*. Their implementations are published by the `DataObject` using the local Lookup.

By this means, the capabilities of a `DataObject` can be flexibly adjusted and accessed from outside. Due to the fact that a `DataObject` knows the type of its managed data, it is able to provide specific data accordingly. That means a `DataObject` is responsible for the creation of a `Node` object, representing data in the user interface. Creating a `DataObject` is done by a `DataObject.Factory`.

The Data Systems API provides a set of superclasses (Figure 7-3), allowing easy implementation of `DataObject`s and data types used in your application.

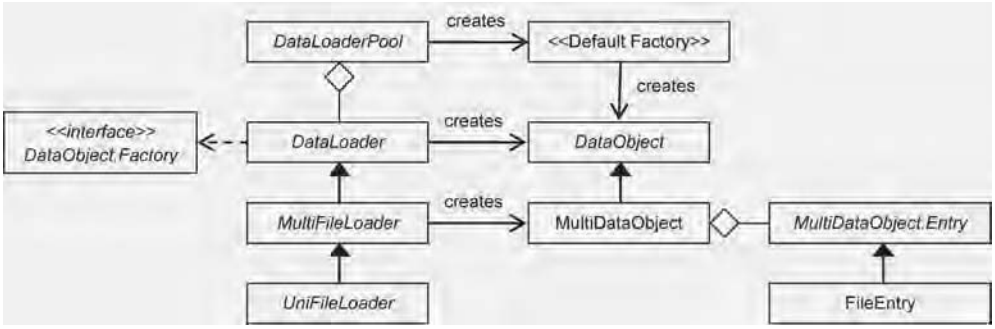


Figure 7-3. *Base classes for the development of DataObjects*

The best way to imagine the combination of the three systems is by means of an example that shows how APIs of the three layers cooperate and build upon each other. The NetBeans IDE provides a wizard that creates parts of the system for a file, all in one go.

Now we will use this wizard to add a `DataObject` for MP3 files to the module created in Chapter 3. To achieve this, go to File ➤ New File, and in the Module Development category, select File Type. First, define the MIME type by entering **audio/mpeg**. The file type is recognized by its file prefix. To use XML files, make the application recognize your file type by entering the root tag. In this case, the files are recognized by the prefix `mp3`, and entered accordingly (see Figure 7-4).

Optionally, provide more than one prefix by entering different types separated by commas. This is meaningful for MPEG video files when you enter, for example, **mpg, mpeg**.

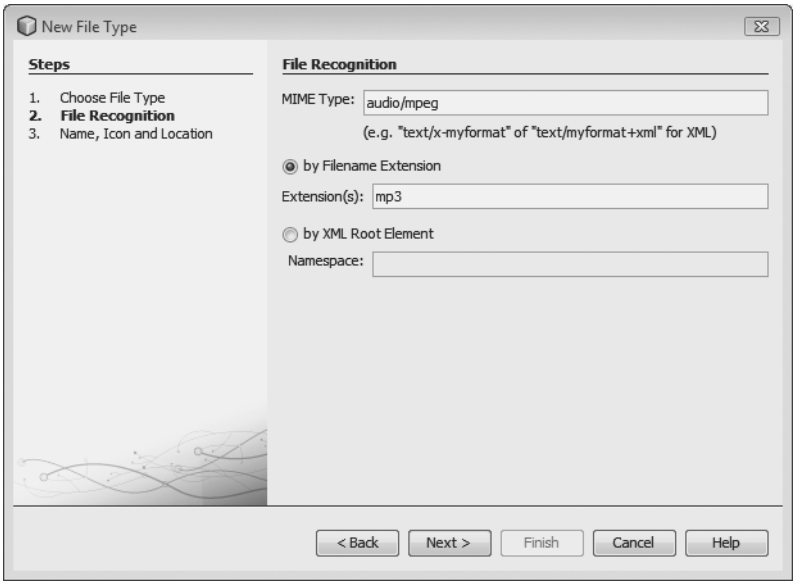


Figure 7-4. *Creating a new file type for MP3 files using the wizard*

Click Next to enter the last page of the wizard. Enter **Mp3** as the class name prefix and choose a supporting icon with a size of 16×16 pixels. When ready, click Finish, which initiates the creation of the file type. This creates the `Mp3DataObject` class containing a `FileObject` of type `.mp3`, and registers a default `DataObject.Factory` in the layer file to create the `DataObject`. We inspect these components in the following sections, using an example. Also presented are features that go beyond the initial examples.

DataObject

In principle, a `DataObject` is specified by the abstract class `DataObject`, but typically `MultiDataObject` is used as a superclass. This class already implements most of the methods in `DataObject`. On one hand, this results in a very small class, but on the other, as the name `MultiDataObject` already suggests, it contains more than one `FileObject`.

A `DataObject` always owns the `FileObject` as its primary file. A `MultiDataObject` contains one or more additional `FileObject`s, which are called *secondary files*. Secondary files are typically used for dependent files, as in the Form Editor, where a single `DataObject` represents the files `myform.java`, `myform.form` and `myform.class`. Here, `myform.java` is the primary file, and the files `myform.form` and `myform.class` are the secondary files.

A `FileObject` inside a `DataObject` is handled using the class `MultiDataObject.Entry`, whereas the subclass `FileEntry` is used in most cases. This class processes such standard file operations as moving or deleting. Have a look at the class `Mp3DataObject` in Listing 7-3, generated by the wizard.

Listing 7-3. *DataObject class for an MP3 FileObject. This class provides the logic for MP3 data.*

```
public class Mp3DataObject extends MultiDataObject {
   public Mp3DataObject(FileObject pf, MultiFileLoader loader)
      throws DataObjectExistsException, IOException {
      super(pf, loader);
   }
   @Override
   protected Node createNodeDelegate() {
      return new DataNode(this, Children.LEAF, getLookup());
   }
   @Override
   public Lookup getLookup() {
      return getCookieSet().getLookup();
   }
}
```

As you already know, `DataObject`s are typically created by a `DataObject.Factory`. The constructor of the class `Mp3DataObject` requires passing both the primary file, which contains the real MP3 file, and the `MultiFileLoader` (`DataLoader` subclass) responsible for this `DataObject`. We simply pass these parameters to the base class constructor, which handles management of the `DataObject`.

Since a `DataObject` knows its data type, it is also responsible for creating the corresponding node used to display the `DataObject` for the user interface. This is done by means of the factory method `createNodeDelegate()`, creating and returning an instance of the `Node` class `DataNode`.

This is the interface to the Nodes API, located in the presentation layer (see Figure 7-1, shown previously). We will revisit this in the "Nodes API" section.

The main difference between a `FileObject` and a `DataObject` is that `DataObject`s know the data they contain. This means a `DataObject` distinguishes itself by the fact that it can provide properties and functionalities for specific types of data; in our case an MP3 file.

The functionality provided by a `DataObject` for its data is specified by its interfaces or abstract classes. These are called cookies. Instances of these interfaces are managed by the `DataObject`, using a `CookieSet`. The fact that the interfaces need not be implemented by the `DataObject` itself, but are managed using the `CookieSet`, allows a `DataObject` to dynamically provide its functionality.

For example, this means it can provide an interface to allow stopping a currently playing MP3 file, only available while the MP3 file is played. In this way, it is possible to flexibly extend a `DataObject` with new functionalities. Type-safe access to these interfaces is obtained via the Lookup provided by the `DataObject` using the `getLookup()` method.

The structure of the `Mp3DataObject` is now finished. The constructor obtains the `FileObject` that is managed from the `MultiFileLoader` in the adjacent abstraction layer. The `DataObject` supplies a representation for the overlying presentation layer and, finally, reveals its functionality to the environment via Lookup. The superclasses `DataObject` and `MultiDataObject` provide several methods for using `DataObject`s.

Looking through the API documentation is very helpful when trying to understand these classes.

Implementing and Adding Cookies

First, specify the functionality of the `Mp3DataObject` by an interface. Call it `PlayCookie` and specify the method `play()`, used to play the corresponding `Mp3DataObject`:

```
public interface PlayCookie {
   public void play();
}
```

Also required is an implementation of functionality specified by our cookie—possibly the direct implementation of the interface by the class `Mp3DataObject`. Better is to use a separate class, a so-called *support class*. This permits flexibly adding and removing functionality to and from the `Mp3DataObject`. Semantic grouping of multiple cookies is possible as well, and the `Mp3DataObject` class remains lean.

```
public class PlaySupport implements PlayCookie {
   private Mp3DataObject mp3 = null;
   public PlaySupport(Mp3DataObject mp3) {
      this.mp3 = mp3;
   }
   public void play() {
      //Mp3Player.getDefault().play(mp3);
      System.out.println("play file");
   }
}
```

Now, adding an instance of this support class to the `CookieSet` of the `Mp3DataObject` class is required. This is accomplished using the method `getCookieSet()` and adding an instance of

the class PlaySupport via assign(). Here, we also set the type to PlayCookie, although you can use the PlaySupport class. But this method allows us independence from the implementation.

```
public class Mp3DataObject extends MultiDataObject {
   public Mp3DataObject(FileObject pf, MultiFileLoader loader)
      throws DataObjectExistsException, IOException {
      super(pf, loader);
      getCookieSet().assign(PlayCookie.class, new PlaySupport(this));
   }
}
```

We have now added functionality to the Mp3DataObject such that it can be used outside the local Lookup available via getLookup().

Using Cookies

A question remains concerning access to the functionalities of a DataObject. This is answered by a CookieAction class. Use the class MyCookieAction created in Chapter 4 or create a new class via the wizard with File ➤ New File ➤ Module Development ➤ Action. The Cookie class is our Mp3DataObject. This allows our actions to be enabled only when an Mp3DataObject is selected—that is, when its node is selected.

The method cookieClasses() is implemented as follows:

```
protected Class[] cookieClasses() {
   return new Class[] { Mp3DataObject.class };
}
```

As shown in Chapter 4, the performAction() method of the CookieAction receives selected nodes. As you have seen in the Mp3DataObject's createNodeDelegate() method, the node representing the Mp3DataObject receives the local Lookup from the Mp3DataObject. This Lookup allows us querying the functionalities of the Mp3DataObject. In other words, the DataNode is a proxy for the Mp3DataObject. It is precisely that Lookup that we obtain from the selected node via the performAction() method. As usual, we now use the Lookup, via the getLookup() method, to obtain an instance of PlayCookie, after which we execute its play() method.

```
protected void performAction(Node[] nodes) {
   PlayCookie pc = nodes[0].getLookup().lookup(PlayCookie.class);
   pc.play();
}
```

Test the code using the Favorites module. Make sure the Favorites module is active in the application by going to Properties ➤ Libraries and then looking in the platform cluster. Then start the application and open the Favorites window using Window ➤ Favorites. Click the window with the right mouse button and select Add to Favorites. Select an MP3 file or a folder with MP3 files and use Add to append the selected entry to the Favorites window. The displayed MP3 files are represented by a DataNode instance, created by an Mp3DataObject, created by a DataObject.Factory the moment the MP3 files are added to you Favorites window.

Providing Cookies Dynamically

By means of the example in Listing 7-4, the provided functionalities of an `Mp3DataObject` can be changed at runtime, which also implicitly control the actions available to the user. Previously, we created a cookie and a support class that plays an MP3 file. Now we create another to stop an MP3 file. Set the current play status of the `Mp3DataObject` using the method `playing()`.

Listing 7-4. *Cookies and support classes required to play an Mp3DataObject*

```
public interface PlayCookie {
   public void play();
}
public class PlaySupport implements PlayCookie {
   private Mp3DataObject mp3 = null;
   public PlaySupport(Mp3DataObject mp3) {
      this.mp3 = mp3;
   }
    public void play() {
       System.out.println("play");
       mp3.playing(true);
    }
}
public interface StopCookie {
   public void stop();
}
public class StopSupport implements StopCookie {
   private Mp3DataObject mp3 = null;
   public StopSupport(Mp3DataObject mp3) {
      this.mp3 = mp3;
   }
   public void stop() {
      System.out.println("stop");
      mp3.playing(false);
  }
}
```

We use the constructor to create both support classes. We can assume that the MP3 file is not being played, and we assign the `PlaySupport` class to the `CookieSet`. Using the `playing()` method, called by the support classes, we change the cookies available in the `CookieSet`. If the file is currently being played, all instances of `PlayCookie` are removed by passing the type without any instances to the `assign()` method, and adding an instance of `StopCookie` (see Listing 7-5). If the file is stopped, everything happens in reverse order.

Listing 7-5. *Adding and removing cookies dynamically*

```
public class Mp3DataObject extends MultiDataObject {
   private PlaySupport playSupport = null;
   private StopSupport stopSupport = null;

   public Mp3DataObject(FileObject pf, MultiFileLoader loader)
      throws DataObjectExistsException, IOException {
```

```
        super(pf, loader);
        playSupport = new PlaySupport(this);
        stopSupport = new StopSupport(this);
        getCookieSet().assign(PlayCookie.class, playSupport);
    }

    public synchronized void playing(boolean value) {
        if(value) {
            getCookieSet().assign(PlayCookie.class);
            getCookieSet().assign(StopCookie.class, stopSupport);
        } else {
            getCookieSet().assign(StopCookie.class);
            getCookieSet().assign(PlayCookie.class, playSupport);
        }
    }
}
```

Making the example complete requires two additional action classes used to start and stop the MP3 file. These are two `CookieAction` classes, using `PlayCookie` and `StopCookie` as cookie classes (see Listing 7-6 for the `PlayAction` class). The menu (or toolbar) entries are activated or deactivated automatically, depending on which cookie or support class is currently provided by the selected MP3 file.

Listing 7-6. *Context-sensitive actions that become active as soon as the selected Mp3DataObject provides an instance of the corresponding cookie*

```
public final class PlayAction extends CookieAction {
    protected void performAction(Node[] n) {
        PlayCookie pc = n[0].getLookup().lookup(PlayCookie.class);
        pc.play();
    }
    protected Class[] cookieClasses() {
        return new Class[] { PlayCookie.class };
    }
}
public final class StopAction extends CookieAction {
    protected void performAction(Node[] n) {
        StopCookie sc = n[0].getLookup().lookup(StopCookie.class);
        sc.stop();
    }
    protected Class[] cookieClasses() {
        return new Class[] { StopCookie.class };
    }
}
```

Creating a DataObject Manually

Normally, a `DataObject` need not be created explicitly, but is created by the `DataLoader` pool on demand. Optionally, you can create a `DataObject` for a given `FileObject` using the static `find()` method of the `DataObject` class:

```
FileObject myFile = ...
try {
   DataObject obj = DataObject.find(myFile);
} catch(DataObjectNotFoundException ex) {
   // no loader available for this file type
}
```

If no `DataLoader` or `DataObject.Factory` is registered for the given file, a `DataObjectNotFoundException` is thrown.

DataObject Factory

New since NetBeans Platform 6.5 is the concept of `DataObject` factories, which can be registered declaratively in the layer file. There is no longer a need for a separate `DataLoader` class. In most cases, you can use the default factory implementation provided by the `DataLoaderPool` class. A `DataObject` factory must implement the new interface `DataObject.Factory`.

In our case, the factory is named `Mp3DataLoader` and is registered in the folder `Loaders/audio/mpeg/Factories` by the File Type wizard (see Listing 7-7). This factory is a default implementation created by the `DataLoaderPool.factory()` method specified via the `instanceCreate` attribute. This method needs the class of the data object to create, the MIME type associated with the object, and an icon to use by default for nodes representing data objects created with this factory.

Listing 7-7. *DataObject factory registration in layer file*

```
<folder name="Loaders">
   <folder name="audio">
      <folder name="mpeg">
         <folder name="Factories">
            <file name="Mp3DataLoader.instance">
               <attr name="SystemFileSystem.icon"
                       urlvalue="nbresloc:/com/galileo/netbeans/module/mp3.png"/>
               <attr name="dataObjectClass"
                       stringvalue="com.galileo.netbeans.module.Mp3DataObject"/>
               <attr name="instanceCreate"
                       methodvalue="org.openide.loaders.DataLoaderPool.factory"/>
               <attr name="mimeType" stringvalue="audio/mpeg"/>
            </file>
         </folder>
      </folder>
   </folder>
</folder>
```

DataLoader

In the previous section, you saw that there is no need for a special `DataLoader` implementation for your file type. Although the usage of the default `DataObject` factory is the recommended approach to take, we will have a look at the `DataLoader` classes and how you can create your own loader.

Implementation

A DataLoader is a factory for a DataObject and is responsible for exactly one data type. The DataLoader recognizes the type of the file, either by file extension or XML root element. DataLoaders are implemented by modules and registered via the layer file. This registration can be used to find the corresponding DataLoader for an appropriate file type. Basically, there exist two types of DataObjects: a DataObject representing only one file, the primary file; and a DataObject that contains additional files, the so-called secondary files. Depending on which data type used, you need different types of DataLoaders.

Using a DataObject with only one primary file, as in our MP3 example, select a DataLoader of the type UniFileLoader. Whenever DataLoaders need to load additional secondary files, choose a MultiFileLoader. The UniFileLoader is only a special variant of the MultiFileLoader, and its created objects are of the type MultiDataObject. You can directly derive from DataLoader and DataObject, but it is considerably easier to use the MultiFileLoader or UniFileLoader class.

The DataLoader Mp3DataLoader responsible for the creation of DataObjects of the type Mp3DataObject was created by the NetBeans wizard. It inherits the abstract superclass UniFileLoader, since our Mp3DataObject solely represents an MP3 file or FileObject. Observe the assembly and functionality of a DataLoader by means of the Mp3DataLoader (see Listing 7-8).

A DataLoader is responsible for a specific MIME type. First, define this type by a private data element. Use the type audio/mpeg. Pass the complete name and DataObject class for which the DataLoader is responsible to the base class constructor. This representation class is labeled for our example as com.galileo.netbeans.module.Mp3DataObject.

Setting a displayable name for the DataLoader is done by overriding the method defaultDisplayName(), in which we read the name from a resource bundle. The initialize() method is used to call the method of the base class and afterward to add the MIME type to the ExtensionList, accessed via the getExtensions() method. The DataLoader infers the file extension from the MIME type, using the file Mp3Resolver.xml, created by the wizard and registered in the layer file in the Services/MIMEResolver folder. If you don't register the MIME type to the ExtensionList using addMimeType(), you can pass the file name directly via addExtension(), making the file Mp3Resolver.xml and its entry in the layer file obsolete.

Registration of file name extensions or MIME types in an ExtensionList allows the DataLoader to recognize its responsibility for a specific file type. Checking if a DataLoader comes into consideration for a specific file type is done by findPrimaryFile(). The UniFileLoader, possessing the ExtensionList, already implements this method and checks if a MIME type or the extension of the parent file matches the type of DataLoader.

If you want to use a MultiDataLoader, you need to implement the method yourself, and should first check if the passed FileObject is the primary file. In this case, directly return it; otherwise (if it is a secondary file), you have to find the appropriate primary file.

For that purpose, the method FileUtil.findBrother() is useful. Otherwise, the findPrimaryFile() method will return null, indicating the DataLoader is not responsible for the file type. The method createMultiObject() is the DataLoader's factory method, responsible for creating an Mp3DataObject instance. This method is called with the primary file as a parameter (in this case an MP3 file), which we pass to the Mp3DataObject constructor. For the creation of MultiDataObject.Entry instances, which manage the FileObjects inside the DataObject, the responsible methods are createPrimaryEntry() and createSecondaryEntry(). For a UniFileLoader, only the createPrimaryEntry() method, which is already implemented, is needed. If you use a MultiFileLoader, you must implement both methods inside the subclass.

Last, a DataLoader defines a folder in the System Filesystem—the layer file in which actions are registered for the particular data type—using the method actionsContext().

These registered actions are shown in the node's context menu—that is, in the context menu of the node responsible for the presentation of the DataObject.

Listing 7-8. *DataLoader for the creation of DataObjects*

```
public class Mp3DataLoader extends UniFileLoader {
   public static final String REQUIRED_MIME = "audio/mpeg";
   public Mp3DataLoader() {
      super("com.galileo.netbeans.module.Mp3DataObject");
   }
   protected String defaultDisplayName() {
      return NbBundle.getMessage(Mp3DataLoader.class, "LBL_Mp3_loader_name");
   }
   protected void initialize() {
      super.initialize();
      getExtensions().addMimeType(REQUIRED_MIME);
   }
   protected MultiDataObject createMultiObject(FileObject pf)
      throws DataObjectExistsException, IOException {
      return new Mp3DataObject(primaryFile, this);
   }
   protected String actionsContext() {
      return "Loaders/" + REQUIRED_MIME + "/Actions";
   }
}
```

Registration

As already mentioned, with the introduction of the DataObject.Factory interface, the DataLoader superclass also implements this interface. This enables you to register your own DataLoader implementation in the layer file instead of the manifest file. The order of the loaders can be determined by the generic position attribute (see also Chapter 3).

The preceding Mp3DataLoader can be registered as Listing 7-9 shows.

Listing 7-9. *DataLoader registration in layer file*

```
<folder name="Loaders">
   <folder name="audio">
      <folder name="mpeg">
         <folder name="Factories">
            <file name="com-galileo-netbeans-module-Mp3DataLoader.instance">
               <attr name="position" stringvalue="100"/>
            </file>
         </folder>
      </folder>
   </folder>
</folder>
```

With this entry, you have registered your own `DataLoader` or `DataObject.Factory` implementation, which creates `Mp3DataObjects`.

Nodes API

The Nodes API is the third and uppermost layer in the NetBeans Resource Management System. The role of the Nodes API is visual representation of data. Closely connected to this API is the Explorer & Property Sheet API, which is the container and manager of nodes. A node exists to present data to the user interface of an application, as well as to give the user actions, functionality, and properties for interacting with underlying data.

However, a node need not merely present data, since it can be used for many other things as well. For example, an action hiding beneath a node could be invoked when the node is double-clicked. Be aware that a node is not typically concerned with business logic, but focuses on providing a presentation layer, delegating user interaction to action classes and, where applicable, to its related `DataObject`.

The general interfaces and behavior are defined by the `Node` superclass. All subclasses can be displayed and managed by an explorer view. Possible subclasses of the `Node` class are shown in Figure 7-5.

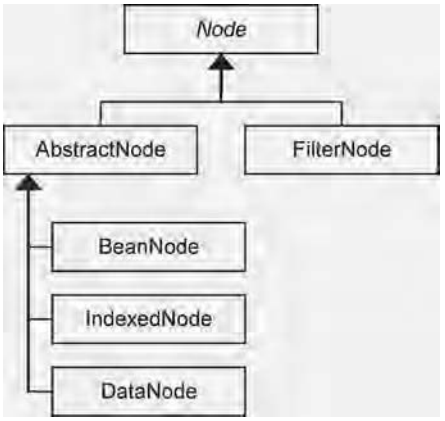


Figure 7-5. *Hierarchy of Node subclasses*

The classes `AbstractNode` and `FilterNode` derive from the `Node` class. The `AbstractNode` provides the simplest form of the `Node` class. Use this class to instantiate a new `Node` directly, without needing to implement or extend any of the `Node` classes. On the other hand, the `FilterNode` creates a proxy `Node` that delegates its method calls to the original `Node`. This kind of `Node` is used when data needs to be displayed in different ways.

The `BeanNode` is used to represent a `JavaBean`. Another kind of `Node`, the `IndexedNode`, lets its children be organized based on a given index. Finally, we have the subclass `DataNode`, which is most commonly used when representing data from files. The `DataNode` is the `Node` type representing `DataObjects` such as those you learned about in the previous sections.

While in previous NetBeans Platform versions, the File Type wizard created a special `DataNode` implementation for your file type, there is in general no more need for such a class.

This class has only set the icon of the file type. This is now done by the registration of the `DataObject` factory. Instead, a `DataNode` instance is used.

Nevertheless, there are use case where you need your own `Node` implementation, as you will see in Chapters 9 and 18 (for example, if the node shall provide properties to be displayed in the Properties window).

Node Container

Each `Node` object has its own `Children` object, providing a container for child nodes, which are the node's subnodes. The `Children` object is responsible for the creation, addition, and structuring of child nodes. Each node within the `Children` object has the `Node` that owns the `Children` object as its parent. For nodes that do not have their own children, such as our `DataNode` for MP3 files, we pass in `Children.LEAF` as an empty container.

Several variations of the `Children` object derive from the `Children` superclass, as shown in Figure 7-6.

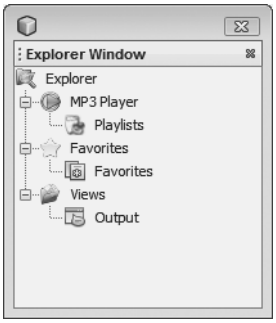


Figure 7-6. *Hierarchy of the different Children container classes*

Table 7-1 shows the different container classes, their characteristics, and their uses.

Table 7-1. *Different Children container class variations and their uses*

Class	Use
`Children.Array`	Superclass for all other `Children` classes. You should not derive from this class directly. This container class manages its nodes in an `Array`. The nodes will be appended at the end of the array and will be delivered in the same order.
`Children.Keys<T>`	Typical superclass for your implementation. Nodes are connected with a key. These keys are also used for ordering.
`Children.Map<T>`	The nodes are stored in a `Map`. The nodes are associated with a key, which is also used for deleting nodes.
`Children.SortedArray`	Extends the `Children.Array` class with a `Comparator`.
`Children.SortedMap<T>`	Extends the `Children.Map<T>` class with a `Comparator`. Therefore, this class is very similar to `Children.SortedArray`.

Actions

A node makes a context menu available to the user, allowing context-sensitive actions. A `DataNode` obtains its context menu's actions from the `DataLoader` of the `DataObject` it represents. For this purpose, a `DataLoader` defines a MIME-specific folder in the layer file via the method `actionsContext()`, where actions are registered. These are read and added automatically to the context menu.

For nodes that do not represent `DataObject`s, the `getActions()` method in the `Node` is used to define the actions in the context menu. Override this method in your class to add more actions to the set provided by default by the NetBeans Platform. Add actions programmatically or use Lookup to retrieve them from the layer file (see the "Context Menu" section in Chapter 5). When overriding the `getActions()` method, make sure to add a call to `super.getActions()`, in addition to the actions you add to the set.

You can also override the `getPreferredAction()` method, which provides the action invoked when the user double-clicks the node. If you return `null` from this method, the first action from the `getActions()` array is invoked.

Event Handling

To react to `Node` events, use a `PropertyChangeListener`, as well as a `NodeListener`. Use the `PropertyChangeListener` to be informed of changes to `Node` properties provided via the `getPropertySet()` method. Via the `NodeListener`, you can listen to internal node changes, such as changes to the name, the parent node, and the child nodes. To that end, the `Node` class makes a range of property keys available, such as `PROP_NAME` and `PROP_LEAF`. The methods listed in Table 7-2 are offered by the `NodeListener`.

Table 7-2. *Methods of the NodeListener interface*

Method	Event
`childrenAdded(NodeMemberEvent evt)`	Called when child nodes are added
`childrenRemoved(NodeMemberEvent evt)`	Called when child nodes are removed
`childrenReordered(NodeMemberEvent evt)`	Called when child nodes are reordered
`nodeDestroyed(NodeEvent evt)`	Called when the parent node is destroyed
`propertyChange(PropertyChangeEvent evt)`	Called when a node property, such as its name, is changed

If you don't want to be informed of the events, or you don't want to implement them, use the `NodeAdapter` class instead of the `NodeListener` interface.

Implementing Nodes and Children

As an example, we introduce the Nodes API, together with the Data Systems API beneath it. You'll learn how to create your own nodes and how to build the `Children` container beneath them. To that end, the nodes are presented in a tree hierarchy, representing actions registered in the layer file. By doing so, we allow the tree structure to be extended, creating an extension

point for other modules. To present the nodes in a tree structure, we use the Explorer & Property Sheet API, about which you will learn more in the next section. The completed example shows an explorer view, as depicted in Figure 7-7.

Figure 7-7. *Example of the usage of the nodes and explorer views*

We define content of the explorer view in the layer file, in a folder called `Explorer`. That folder defines the extension point of our window and, more generally, the extension point of our module. Within the folder, actions are registered at various levels of nesting, displayed in the explorer view represented by nodes. The content of the layer file, as reflected in Figure 7-7, is as shown in Listing 7-10.

Listing 7-10. *Extension point in the layer file. All entries in the Explorer folder are displayed in the Explorer window*

```
<folder name="Explorer">
  <attr name="icon" stringvalue="com/galileo/netbeans/module/explorer.png"/>
  <folder name="MP3 Player">
    <attr name="icon" stringvalue="com/galileo/netbeans/module/player.png"/>
    <file name="PlaylistAction.shadow">
      <attr name="originalFile" stringvalue="
              Actions/Edit/com-galileo-netbeans-module-PlaylistAction.instance"/>
    </file>
  </folder>
  <folder name="Views">
    <attr name="icon" stringvalue="com/galileo/netbeans/module/views.png"/>
    <file name="OutputAction.shadow">
      <attr name="originalFile" stringvalue="Actions/Window/
              org-netbeans-core-output2-OutputWindowAction.instance"/>
    </file>
  </folder>
  <folder name="Favorites">
    <attr name="icon" stringvalue="com/galileo/netbeans/module/favorites.png"/>
    <file name="FavoritesAction.shadow">
      <attr name="originalFile" stringvalue="
              Actions/Window/org-netbeans-modules-favorites-View.instance"/>
    </file>
```

```
  </folder>
</folder>
```

You can see that actions are registered in the same way as is done for menus or toolbars, via shadow files. Additionally, we will assign an icon to a folder with the self-defined attribute `icon`.

To display this structure in a node hierarchy, we require a `Node` class that represents a folder, a `Children` class to manage all the nodes beneath a folder (as well as all the actions and subfolders), and a `Node` class that represents an action.

Start with the `Node` class that represents the content of a folder. We call this `Node` the `ExplorerFolderNode` and create it from the convenience class `AbstractNode`. As a result, we need nothing more than a constructor. Pass a `FileObject` into the constructor, representing a folder within the layer file. The superclass constructor receives an instance of the `Children` class `ExplorerNodeContainer`, representing all the child nodes. Then set the name and icon path of the node from values in the layer file.

```
public class ExplorerFolderNode extends AbstractNode {
   public ExplorerFolderNode(FileObject node) {
      super(new ExplorerNodeContainer(node));
      setDisplayName(node.getName());
      String iconBase = (String) node.getAttribute("icon");
      if(iconBase != null) {
         setIconBaseWithExtension(iconBase);
      }
   }
}
```

The `ExplorerNodeContainer` container is derived from the `Children<Keys>` class normally treated as a superclass. Pass the parent node's `FileObject` into the constructor, since this is where entries found beneath this node are loaded and used.

The `addNotify()` method is called automatically when the parent node is expanded. That means the children are created on demand—that is, only when needed. Within the `addNotify()` method, we set the key with the `FileObject` of the parent node. This object is then received as a parameter within the `createNodes()` method, which is invoked automatically in order to actually create the `Children` object.

Within the `createNodes()` method, use the `getFolders()` method to read all the subfolders, create an instance of `ExplorerFolderNode`, and add it to a list (see Listing 7-11). That class contains an `ExplorerNodeContainer`, whereby we obtain the required recursion for creating any level of hierarchy required.

Next, read the actions from the folder. Use the `FolderLookup` class, which gives the instances. This acts recursively by default, which means it passes back all the actions, not just those wanted from the current folder. Therefore, create a subclass of `FolderLookup` named `ActionLookup`. To prevent the recursion, simply override the `acceptContainer()` and `accept-Folder()` methods and return `null`. Now use the `ActionLookup` to retrieve all instances of the current folder.

Use the `lookupAll()` method to receive all instances that are of type `Action`. For each `Action`, we receive an `ExplorerLeafNode`, which is responsible for representing an `Action` and not a `Children` object, which therefore does not return a lower level of the hierarchy. We add

this `Node` to the list, which we return as an array. Therefore, this method provides the core of the whole system created in this example.

Listing 7-11. *Container class that loads and manages all child nodes dynamically*

```
public class ExplorerNodeContainer extends Children.Keys<FileObject> {
   private FileObject folder = null;
   public ExplorerNodeContainer(FileObject folder) {
      this.folder = folder;
   }
   protected void addNotify() {
      setKeys(new FileObject[] {folder});
   }

   protected Node[] createNodes(FileObject f) {
      ArrayList<Node> nodes = new ArrayList<Node>();
      /* add folder nodes /
      for(FileObject o : Collections.list(f.getFolders(false))) {
         nodes.add(new ExplorerFolderNode(o));
      }
      DataFolder df = DataFolder.findFolder(f);
      FolderLookup lkp = new ActionLookup(df);
      /* add leaf nodes, which represents an action */
      for(Action a : lkp.getLookup().lookupAll(Action.class)) {
         nodes.add(new ExplorerLeafNode(a));
      }
      return(nodes.toArray(new Node[nodes.size()]));
   }

   /* non-recursive folder lookup */
   private static final class ActionLookup extends FolderLookup {
      public ActionLookup(DataFolder df) {
         super(df);
      }
      protected InstanceCookie
         acceptContainer(DataObject.Container con) {
         return(null);
      }
      protected InstanceCookie acceptFolder(DataFolder df) {
         return(null);
      }
   }
}
```

Finally, have a look at the `ExplorerLeafNode` class (see Listing 7-12). It simply contains a single action, invoked on the double-click of a node. The action received via the `ActionLookup` is passed to the constructor. Since this type of node does not contain child nodes, we pass `Children.LEAF` to the superclass constructor. Then we set the name to be used when the node is displayed.

Use the `getPreferredAction()` method to provide an action to be invoked when the node is double-clicked. That action will obviously be the one we received as a parameter. Last but not least, override the `getIcon()` method to set the icon of the action to be the icon of the node.

Listing 7-12. *Leaf node that represent the action, executed by a double-click*

```
public class ExplorerLeafNode extends AbstractNode {
   private Action action = null;
   public ExplorerLeafNode(Action action) {
      super(Children.LEAF);
      this.action = action;
      setDisplayName(Actions.cutAmpersand((String)action.getValue(Action.NAME)));
   }
   public Action getPreferredAction() {
      return action;
   }
   public Image getIcon(int type) {
      ImageIcon img = (ImageIcon) action.getValue(Action.SMALL_ICON);
      if(img != null) {
         return img.getImage();
      } else {
         return null;
      }
   }
}
```

In this section, you've seen how to create your own `Node` classes, and you've seen which responsibilities are handled by the `Children` object. With these classes, we are able to display folders and files within the layer file. To place the node in a tree structure, work with explorer views. That part of the presentation of nodes is handled by the Explorer & Property Sheet API. The next section affords a short introduction into this topic, ending with an example that wraps up the code discussed in this section.

Explorer & Property Sheet API

Using the Explorer & Property Sheet API, you display and manage nodes in a wide variety of ways. To that end, the API makes a range of explorer views available, with which you present your nodes in one of many structures. The class hierarchy of these views is shown in Figure 7-8.

The `ChoiceView` class displays your node in a combo box, while the `MenuView` does so in a menu structure of any depth. The most commonly used view is the `BeanTreeView`, displaying nodes in a tree structure. Apart from the display of nodes, the views provide actions and context-sensitive menus via the `getActions()` method.

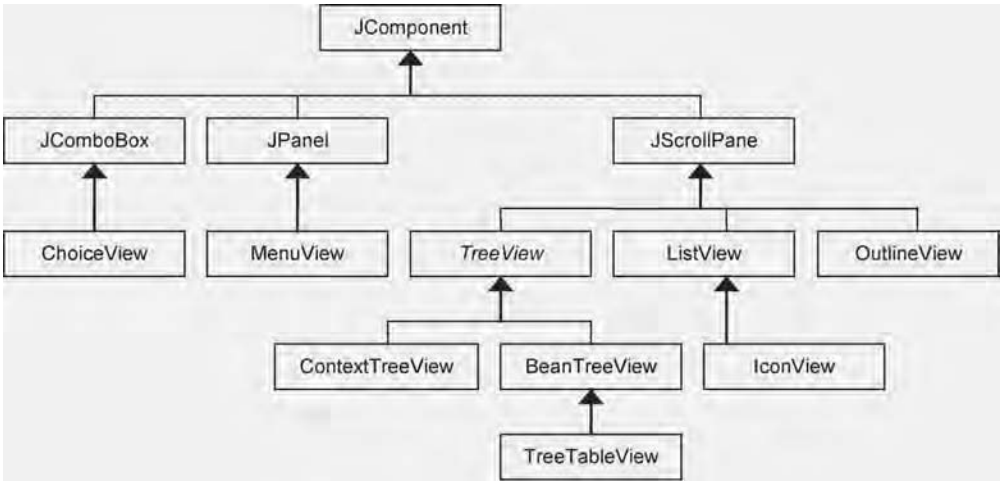


Figure 7-8. *Class hierarchy of different explorer views*

Managing explorer views is done by the `ExplorerManager` class. An instance of this class is provided by the `TopComponent` containing the explorer view. It is important to note that the `ExplorerManager` is not connected to the explorer view via any coding on your part. The explorer view simply examines its component hierarchy until it finds an `ExplorerManager` to display it. To enable the `ExplorerManager` to be found by the explorer view, the `TopComponent` implements the `ExplorerManager.Provider` interface. This interface provides the `getExplorerManager()` method, by which the `ExplorerManager` and the view find each other. As a result, multiple views are displayed by the same `ExplorerManager`.

A main task of the `ExplorerManager` is maintaining the selection of the nodes in the view. It makes available the selected node, together with the selected node's Lookup. To make the selection available to the outside—either to actions, other `TopComponent`s, or completely different modules—we must take additional steps. Use the helper class `ExplorerUtils` to define a Lookup using the `createLookup()` method, representing the selected node along with the selected node's Lookup. Define this Lookup via the `associateLookup()` method as the `TopComponent`'s local Lookup. As a result, the Lookup is available from the outside, thanks to the global Lookup obtained via `Utilities.actionsGlobalContext()`.

In the "Nodes API" section, we created `Node` and `Children` classes. What remains missing is a window containing the explorer view that displays the node. As you complete this missing step, you'll learn how the explorer view and the `ExplorerManager` relate to each other. Start by using the Window Component wizard to create a new `TopComponent` named `ExplorerTopComponent`. Next, add the `ExplorerManager`. Do this by implementing `ExplorerManager.Provider` and create a private instance of the `ExplorerManager`. Then use `getExplorerManager()` to return the `ExplorerManager`.

Next, add an explorer view to the `TopComponent`. In this case, we add a `BeanTreeView`. A simple way of doing this is to drag and drop a `JScrollPane` onto the `TopComponent`, and then, switching to the Properties dialog, set Custom Code Creation in the Code tab to "new BeanTreeView()." Your `initComponents()` method will then look as shown in Listing 7-13. As pointed out earlier, the explorer view finds the `ExplorerManager` automatically, which means you need not take extra steps to connect the view to the `ExplorerManager`.

Each view and each `ExplorerManager` are based on a root element from which all nodes derive. To that end, add `setRootContext()` to the `initTree()` method and pass in an instance of the `ExplorerFolderNode`. From that node, all others are created. We create the node only once the `Explorer` folder is available in the System Filesystem—that is, only when a module registers an `Explorer` folder in its layer file.

Listing 7-13. *Explorer window that displays the nodes with a BeanTreeView. An ExplorerManager manages the nodes.*

```
public final class ExplorerTopComponent extends TopComponent
   implements ExplorerManager.Provider {
   private static final String ROOT_NODE = "Explorer";
   private final ExplorerManager manager = new ExplorerManager();
   private ExplorerTopComponent() {
      initComponents();
      initTree();
      initActions();
      associateLookup(ExplorerUtils.createLookup(manager, getActionMap()));
   }
   private JScrollPane jScrollPane1;
   private void initComponents() {
      jScrollPane1 = new BeanTreeView();
      setLayout(new BorderLayout());
      add(jScrollPane1, BorderLayout.CENTER);
   }
   private void initTree() {
      FileObject folder = Repository.getDefault().
         getDefaultFileSystem().findResource(ROOT_NODE);
      if(folder != null) { /* folder found */
         manager.setRootContext(new ExplorerFolderNode(folder));
      }
   }
   private void initActions() {
      CutAction cut = SystemAction.get(CutAction.class);
      getActionMap().put(cut.getActionMapKey(),
              ExplorerUtils.actionCut(manager));
      CopyAction copy = SystemAction.get(CopyAction.class);
      getActionMap().put(copy.getActionMapKey(),
              ExplorerUtils.actionCopy(manager));
      PasteAction paste = SystemAction.get(PasteAction.class);
      getActionMap().put(paste.getActionMapKey(),
              ExplorerUtils.actionPaste(manager));
      DeleteAction delete = SystemAction.get(DeleteAction.class);
      getActionMap().put(delete.getActionMapKey(),
              ExplorerUtils.actionDelete(manager, true));
   }
   public ExplorerManager getExplorerManager() {
      return manager;
   }
   protected void componentActivated() {
      ExplorerUtils.activateActions(manager, true);
```

```
    }
    protected void componentDeactivated() {
        ExplorerUtils.activateActions(manager, false);
    }
}
```

Our next step connects the standard cut, copy, paste, and delete actions to the `ExplorerManager` actions. We do this with the `initActions()` method. Standard actions are made available to you by the NetBeans Platform. The `ExplorerManager` actions give us the `ExplorerUtils` class, which we register with our `TopComponent` via the `ActionMap` key in the `ActionMap`. To allow the currently selected node to be available to the view via the `TopComponent` Lookup, create a `ProxyLookup` via the call to `ExplorerUtils.createLookup()`. That provides the currently selected node via the Lookup. The `ProxyLookup` is defined as the local Lookup of our `TopComponent` via the `associateLookup()` method.

To ensure that the actions in the `ActionMap` are active, add the `ActionMap` to the Lookup (see Chapter 4). Pass the `ActionMap` directly into the Lookup via the `createLookup()` method, ensuring that the `ActionMap` is available via the global Lookup.

To save resources, we add and remove the listeners of the `ExplorerManager`'s actions in the `componentActivated()` and `componentDeactivated()` methods, which are called when the window is activated and deactivated.

At this point, you are referred to the many tutorials on `http://platform.netbeans.org`, which are easy to understand. Especially in the context of nodes, explorer views, and property sheets, this site contains many code snippets of interest.

Summary

In this chapter, you learned about four of the most important NetBeans APIs, together with their dependencies. You learned about these via an example that displays MP3 files. Of the four, the File Systems API is found on the lowest level, as a generic abstraction layer over any kind of data. On top of that, the Data Systems API handles the logic relating to the data abstracted by the File Systems API. For example, you can use the Data Systems API to connect an MP3 file with the functionality that plays it.

The Nodes API, which is above the Data Systems API, is responsible for providing a data presentation layer, as well as for letting the user set properties and invoke actions on the underlying data. Finally, the Explorer & Property Sheet API offers a wide range of Swing containers that display the nodes and their properties to the user.

CHAPTER 8

Graphical Components Let's Show Some Interesting Views!

To create dialogs and wizards, the NetBeans Platform uses a set of APIs that focus on business layer matters, rather than their infrastructural concerns. These, and additional APIs such as the MultiViews API and the Visual Library API, are discussed in this chapter. The essentials of these APIs are covered in several small code listings.

Dialogs API

The Dialogs API creates and displays dialogs and wizards. The dialogs are based on the Java `Dialog` class. Using the Dialogs API, you can display standard dialogs, as well as custom dialogs tailored to specific business needs. In addition, the API integrates well with the NetBeans window system, as well as the NetBeans help system.

Standard Dialogs

Use the `NotifyDescriptor` class to define the properties of a standard dialog. Provide a message in the form of a string, an icon, or a component, which display together with the dialog. Optionally, use an array to display multiple messages in varying situations. Different types of messages can be specified, giving control over the icon displayed. Define the type via the predefined constants in the `NotifyDescriptor`, as listed in Table 8-1.

Table 8-1. *String literals for displaying message types*

String Literal	Message/Symbol
`PLAIN_MESSAGE`	The message is displayed neutrally, without a symbol.
`INFORMATION_MESSAGE`	The information symbol is displayed with the message.
`QUESTION_MESSAGE`	The question symbol is displayed with the message.

Table 8-1. *String literals for displaying message types (Continued)*

String Literal	Message/Symbol
WARNING_MESSAGE	The warning symbol is displayed with the message.
ERROR_MESSAGE	The error symbol is shown with the message.

An option type defines which buttons are displayed in the dialog. Four string literals are available, as shown in Table 8-2.

Table 8-2. *String literals defining dialog buttons*

String Literal	Buttons Displayed
DEFAULT_OPTION	The standard buttons are displayed. For example, an information dialog only has an OK button, while an entry dialog has an OK button as well as a Cancel button.
OK_CANCEL_OPTION	OK and Cancel buttons are displayed.
YES_NO_OPTION	Yes and No buttons are displayed.
YES_NO_CANCEL_OPTION	Yes, No, and Cancel buttons are displayed.

Finally, you can use the constructor or the `setAdditionalOptions()` method to pass in an `Object` array to add additional buttons to the dialog. Typically, `String` objects are passed here, though you can also use `Component` or `Icon` objects. Rather than the standard buttons, custom buttons can be provided via the `setOptions()` method or by passing them in to the constructor. Here, too, the classes `String`, `Component`, and `Icon` are used:

```
NotifyDescriptor d = new NotifyDescriptor(
   "Text", // Dialog message
   "Title", // Dialog title
   NotifyDescriptor.OK_CANCEL_OPTION, // Buttons
   NotifyDescriptor.INFORMATION_MESSAGE, // Symbol
   null, // Own buttons as Object[]
   null); // Additional buttons as Object[]
```

Dialog description, such as the one defined previously, is passed in to the `notify()` method of the `DialogDisplayer` class, which is responsible for creation and display of dialogs, and also gives a return value when the dialog closes. The `DialogDisplayer` is created as a global service, with a provider obtained via the `getDefault()` method.

```
Object retval = DialogDisplayer.getDefault().notify(d);
```

Buttons the user clicks are identified via the return values, which indicate the following in Table 8-3.

Table 8-3. *String literals as return values*

String Literal	Returned When
`OK_OPTION`	The OK button is clicked
`YES_OPTION`	The Yes button is clicked
`NO_OPTION`	The No button is clicked
`CANCEL_OPTION`	The Cancel button is clicked
`CLOSED_OPTION`	The dialog is closed without any button having been clicked

For various dialog types, the Dialogs API provides four subclasses to the `NotifyDescriptor` class, so you need to define only a few parameters.

Information Dialog

Create an information dialog via the `NotifyDescriptor.Message` class. Pass the text to be displayed to the constructor, as well as an optional message type. By default, the dialog shows the information symbol, as shown in Figure 8-1.

```
NotifyDescriptor nd = new NotifyDescriptor.Message("Information");
```

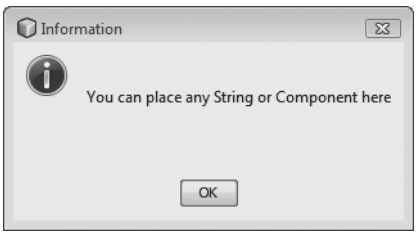


Figure 8-1. *Information dialog*

Question Dialog

Should the user be enabled to answer a question posed in the dialog (see Figure 8-2), use the `NotifyDescriptor.Confirmation` class. To that end, a range of constructors are available for passing in the message, message type, and additional option types.

```
NotifyDescriptor d = new NotifyDescriptor.Confirmation(
   "You can place any String or Component here",
   "That's a question");
```

Figure 8-2. *Question dialog*

Input Dialog

An input dialog is easily created via the `NotifyDescriptor.InputLine` class. Define the text and title to be displayed in the input area of the dialog (see Figure 8-3). Optionally, pass in an option type and message type so that the desired buttons and symbols are shown.

```
NotifyDescriptor d = new NotifyDescriptor.InputLine(
   "First and last name:",
   "Please enter your name");
```

Figure 8-3. *Input dialog*

Access text entered by the user via the `getInputText()` method. Optionally, enter text into the field via the `setInputText()` method.

Error Dialog

To show an exception (see Figure 8-4), use the `NotifyDescriptor.Exception` class. Pass in a `Throwable` instance to the constructor, such as an `Exception` object:

```
Exception ex = new Exception("An exception has occured");
NotifyDescriptor d = new NotifyDescriptor.Exception(ex);
```

Figure 8-4. *Error dialog*

Custom Dialogs

Custom dialogs are created via the `DialogDescriptor` class. The `DialogDescriptor` class is an extension of the `NotifyDescriptor` class. Pass in a `Component` object to be displayed, while also defining dialog modality and a related `ActionListener` that reacts when the buttons are clicked. Optionally, pass in a `HelpCtx` object, providing the ID of a help topic so that the provided topic is automatically opened when the Help button is clicked. For the `DialogDescriptor`, create a `Dialog` object via the `DialogDisplayer`'s `createDialog()` method. Alternatively, create the dialog directly, via the `notify()` or `notifyLater()` methods.

The following example illustrates creation of a Login dialog via the `DialogDescriptor` class. It is important that the dialog be displayed at the appropriate time—when the application starts. The application should only be blocked until login details are correctly entered. Two approaches are supported, as discussed in the following paragraphs.

As mentioned, a `Component` object can be passed into the `DialogDescriptor`, displaying it in the dialog. In the example (see Figure 8-5), this approach is used to integrate two text fields into the dialog so that the user can enter a username and password. The panel makes the username and password available via its `getUsername()` and `getPassword()` methods. To allow dialog display as the application starts, a module installer is used (see Chapter 3), applying the `restored()` method to create a `DialogDescriptor`, resulting in a Login dialog.

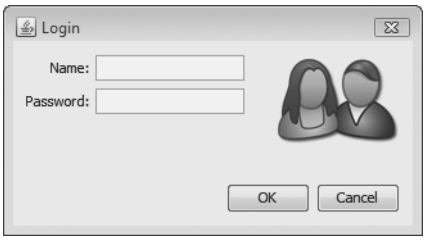


Figure 8-5. *Login dialog created via a DialogDescriptor and a panel*

To allow the dialog to perform asynchronously (required since it must be displayed during the initialization sequence), it is recommended that you register an `ActionListener` to react to user button clicks. Use the `actionPerformed()` method to handle the login logic. If the entered values are incorrect, exit the application via the `LifecycleManager` class (see Chapter 17).

To allow reaction when users click the Close button (in the upper right of the dialog), register a `PropertyChangeListener`, in which the application is closed. To display the dialog immediately after the initialization phase—that is, directly after the splash screen—use the `notifyLater()` method, as shown in Listing 8-1.

Listing 8-1. *Login dialog displayed when application starts, blocking application until username and password are successfully entered*

```
public class Installer extends ModuleInstall implements ActionListener {
   private LoginPanel panel = new LoginPanel();
   private DialogDescriptor d = null;
   @Override
   public void restored() {
      d = new DialogDescriptor(panel, "Login", true, this);
```

```
            d.setClosingOptions(new Object[]{});
            d.addPropertyChangeListener(new PropertyChangeListener() {
               public void propertyChange(PropertyChangeEvent e) {
                  if(e.getPropertyName().equals(DialogDescriptor.PROP_VALUE)
                  && e.getNewValue()==DialogDescriptor.CLOSED_OPTION) {
                     LifecycleManager.getDefault().exit();
                  }
               }
            });
            DialogDisplayer.getDefault().notifyLater(d);
         }
         public void actionPerformed(ActionEvent event) {
            if(event.getSource() == DialogDescriptor.CANCEL_OPTION) {
               LifecycleManager.getDefault().exit();
            } else {
               if(!SecurityManager.login(panel.getUsername(), panel.getPassword())) {
                  panel.setInfo("Wrong username or password");
               } else {
                  d.setClosingOptions(null);
               }
            }
         }
      }
```

Another way to display the dialog uses the `notify()` method in a separate thread, as soon as the application is available. Do this via the `invokeWhenUIReady()` method, provided by the `WindowManager` class. The difference between this approach and `notifyLater()` is that the dialog is only displayed when the application is completely loaded.

```
WindowManager.getDefault().invokeWhenUIReady(new Runnable(){
   public void run() {
      DialogDisplayer.getDefault().notify(d);
   }
});
```

Finally, a complete dialog can be built from scratch, by extending `JDialog`. To that end, use the related NetBeans IDE wizard available via File ➤ New File ➤ Java GUI Forms ➤ JDialog Form. For the application window to be the dialog parent, obtain it like this:

```
Frame f = WindowManager.getDefault().getMainWindow();
```

Wizards

Aside from support for dialogs, the Dialogs API includes a wizard framework to create step-by-step procedures that help users work through a particular process. These processes potentially generate code or other artifacts as the wizard concludes. Wizards of this kind are familiar within the NetBeans IDE itself, such as those used to create new windows or actions.

For each step, provide a panel appropriate to the related data entry required for the step. Coordination between steps is handled by the wizard framework. The NetBeans IDE provides a wizard for creating wizards. To show different ways in which a wizard can be displayed, while focusing on the architecture of wizards, create a wizard for the creation of playlists. The wizard

provides two steps. The first step allows users to describe the playlist, as shown in Figure 8-6, while the second allows music titles to be chosen and added to the playlist.

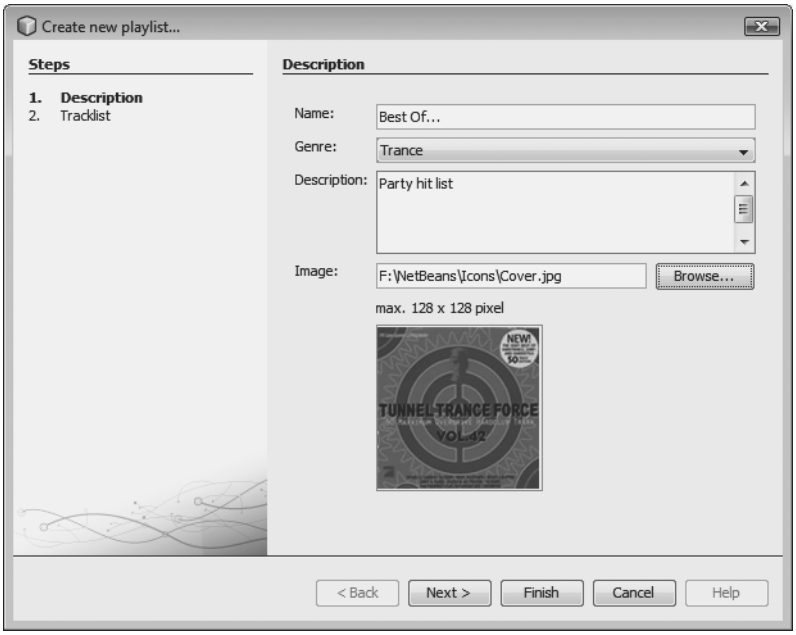


Figure 8-6. *First step in the playlist-creation example*

Wizard Architecture

The `WizardDescriptor` class describes and configures wizards. The class is a subclass of the `DialogDescriptor` class, explained in the previous section. The `DialogDescriptor` class, in turn, is a subclass of `NotifyDescriptor`. The `WizardDescriptor` contains and manages all panels in the wizards and is responsible for tasks such as management of steps and display of user interface components. In other words, the `WizardDescriptor` is the controller of the entire wizard. Typically, the `WizardDescriptor` also provides the `DataModel`, from which data collected over various steps is saved as properties. Rather than using this `DataModel`, your own can be provided.

For each step in the wizard, provide a panel. Typically, a panel is built out of two separate classes. The first class implements the GUI. This class is known as the *visual panel*, and normally extends `JPanel`. The second class, handling the management and validation of the panel, is known as the *wizard panel*. This class extends the NetBeans API `WizardDescriptor.Panel<Data>` class. It creates the visual panel on demand and makes it available to the wizard.

In terms of the MVC paradigm, the visual panel is the view, and the wizard panel is the controller. The visual panel only deals with user interface concerns, providing entry fields for the user and making them available via getters and setters. The visual panel does not contain business logic and, in particular, does not deal with wizard-specific classes or calls. As a result, the panel is completely reusable and can be easily ported to an entirely different wizard. In this way, the panel is reused in a dialog where data is edited in a different context. The relationship between `WizardDescriptor`, `WizardPanel`, and `VisualPanel` is shown in Figure 8-7.

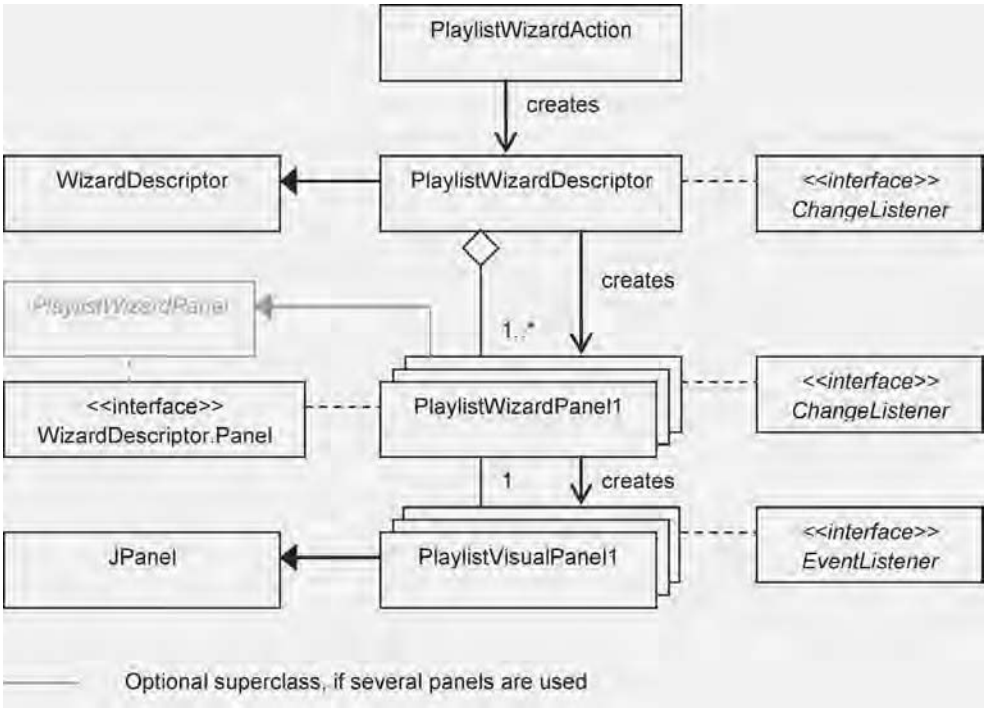


Figure 8-7. *Architecture of a wizard*

Creating Panels

The skeleton of a wizard is created in the NetBeans IDE. Go to File ➤ New File and, in the Module Development category, choose the Wizard file type. In the next step, choose Custom as the Registration Type, and set Static as the Wizard Step Sequence. Enter **2** for the number of panels to be created. In the final step, provide a prefix for the name of the classes created. For this example, use `Playlist` as the prefix. Finally, click Finish. The IDE then creates the skeleton of two panels, both with a visual panel and a wizard panel. Next, customize the content of several of the methods predefined by the wizard.

First, open the user interface of the visual panel (the `PlaylistVisualPanel1` panel) in the NetBeans Form Editor. Add the several fields the user interacts with to describe the playlist. The user must be able to assign a name to the playlist, while choosing a genre and providing a description and image. The completed panel should look like Figure 8-6, where the panel is shown integrated into the wizard.

The panel is a normal Swing component, extending `JPanel`; however, you should bear in mind the following implementation details:

- For each piece of data requested from the user, a public property constant is defined. These are the name of the playlist, the genre, a description, and an image. These constants are needed later to save and load data into the `DataModel`.
- In the constructor, a listener is added to each field expected to be filled with data. In our example, make sure the name has at least three characters, the chosen image has a maximum size of 128×128 pixels, and a warning message is shown when no image has

been selected. To that end, we register a `DocumentListener` for the text fields `playlistName` and `imagePath`.

- It is important to override the `getName()` method. To do this, provide the names of the panels displayed in the headers of the steps in the wizard.
- For each field, add a getter method the wizard panel uses to access data entered by the user.
- In the `DocumentListener` methods `changedUpdate()`, `insertUpdate()`, and `removeUpdate()`, use the `firePropertyChange()` method to notify all registered `PropertyChangeListener`s when the related method is invoked. The interaction between the three layers of a wizard is discussed in the next section.

Listing 8-2 shows a section from the visual panel `PlaylistWizardPanel1`. Notice the panel is completely free from wizard logic, focusing only on the user interface of the wizard step.

Listing 8-2. *Visual panel of the first wizard step*

```
public final class PlaylistVisualPanel1 extends JPanel implements DocumentListener {
   public static final String PROP_PLAYLIST_NAME = "playlist";
   public static final String PROP_GENRE = "genre";
   public static final String PROP_DESCRIPTION = "description";
   public static final String PROP_IMAGE_PATH = "imagePath";
   public PlaylistVisualPanel1() {
      initComponents();
      playlistName.getDocument().addDocumentListener(this);
      imagePath.getDocument().addDocumentListener(this);
   }
   public String getName() {
      return NbBundle.getMessage(PlaylistWizardPanel1.class, "Panel1.Name");
   }
   public String getPlaylistName() {
      return playlistName.getText();
   }
   public String getGenre() {
      return (String)genre.getSelectedItem();
   }
   public String getDescription() {
      return description.getText();
   }
   public String getImagePath() {
      return imagePath.getText();
   }
   public void changedUpdate( DocumentEvent e ) {
      if (playlistName.getDocument() == e.getDocument()) {
         firePropertyChange(PROP_PLAYLIST_NAME, 0, 1);
      } else if(imagePath.getDocument() == e.getDocument()) {
         firePropertyChange(PROP_IMAGE_PATH, 0, 1);
      }
   }
}
```

Look at the content of the related wizard panel, providing the controller for the visual panel. The class implements the `WizardDescriptor.Panel<Data>` interface, defining the interfaces of wizard panels. Pass in a class used as the `DataModel`. As you need no special custom `DataModel`, the class `WizardDescriptor` is used to define the wizard panel. In addition, implement a `PropertyChangeListener`, allowing reaction to changes in the visual panel. A wizard panel has a status: valid or invalid. Validity depends on extensions provided to the wizard panel. In our case, a panel is only valid when the name has at least three characters. The status is saved via the private Boolean `isValid`.

```
public class PlaylistWizardPanel1 implements
   WizardDescriptor.Panel<WizardDescriptor>, PropertyChangeListener {
   private PlaylistVisualPanel1 view = null;
   private WizardDescriptor model = null;
   private boolean isValid = false;
   private ResourceBundle bundle = NbBundle.getBundle(PlaylistWizardPanel1.class);
```

The `getComponent()` method is a factory method creating the visual panel where needed. The method is called from the `WizardDescriptor` when the panel is first created in the wizard. To that end, do not create all panels initially. This will significantly improve the performance of wizards that provide many different steps. Therefore, be very careful using the `getComponent()` method. For example, do not call it in the `getName()` method when the wizard is created.

After creating visual panels, set properties that influence display of components in the wizard. Use `PROP_CONTENT_SELECTED_INDEX` to provide the number of the panel (shown in the table of contents on the left side of the wizard), enabling the user to see the number of the current step, as well as how many steps must still be completed.

Set the property `PROP_AUTO_WIZARD_STYLE` to `true`, which creates wizards with a contents section, as well as a header section. Setting this to `false` makes sense when the wizard has only one step, so that these additional parts become superfluous.

Via the properties `PROP_CONTENT_DISPLAYED` and `PROP_CONTENT_NUMBERED`, specify that names and numbers of wizard steps are shown on the left side of the wizard.

```
   public PlaylistVisualPanel1 getComponent() {
      if (view == null) {
         view = new PlaylistVisualPanel1();
         view.putClientProperty(
                 WizardDescriptor.PROP_CONTENT_SELECTED_INDEX, new Integer(0));
         view.putClientProperty(
                 WizardDescriptor.PROP_AUTO_WIZARD_STYLE, Boolean.TRUE);
         view.putClientProperty(
                 WizardDescriptor.PROP_CONTENT_DISPLAYED, Boolean.TRUE);
         view.putClientProperty(
                 WizardDescriptor.PROP_CONTENT_NUMBERED, Boolean.TRUE);
      }
      return view;
   }
```

Using the `getName()` method, provide the names shown in the header section of the wizard. With `getHelp()`, provide a `HelpCtx.DEFAULT_HELP`, activating the Help button in the wizard. Further information about the `HelpCtx` class and the NetBeans help system are described in Chapter 9.

The status of panels, as discussed earlier (defining whether the wizard step is valid or not) is provided via the `isValid()` method. The `isValid()` method is called from the `WizardDescriptor`, when a panel is closed or via notifications received from the `ChangeListener`. When the method returns the value `true`, the Next or Finish buttons are activated.

The `setMessage()` method is a helper method for which advice is displayed to the user. The advice area of the wizard is provided by default and displayed in the lower part of a panel within the wizard. Text is defined via the property `WizardDescriptor.PROP_ERROR_MESSAGE`. Additionally, if the property is set and the `isValid()` method returns `false`, an error symbol is shown. If the `isValid()` method returns `true`, only a warning symbol is displayed. No symbol is shown if `null` is returned. You can also use the properties `PROP_WARNING_MESSAGE` or `PROP_INFO_ MESSAGE` to display warning or normal (info) messages.

```
public String getName() {
   return bundle.getString("Panel1.Name");
}
public HelpCtx getHelp() {
   return HelpCtx.DEFAULT_HELP;
}
public boolean isValid() {
   return isValid;
}
private void setMessage(String message) {
   model.putProperty(WizardDescriptor.PROP_ERROR_MESSAGE, message);
}
```

The `DataModel` is accessed via the `readSettings()` and `storeSettings()` methods. The type of `DataModel` depends on the template provided, which is defined via the interface specified in the class signature. In this case, the class in question is `WizardDescriptor`.

The `readSettings()` method is called when the panel is opened. Here, values are read into panels in the wizard. Register a `PropertyChangeListener` in the visual panel, informing of user activities in the panel. Register it here to make sure the `WizardDescriptor` is available.

The `storeSettings()` method is called when panels are exited. Save the values defined by the user in the `WizardDescriptor` via the property names provided by the visual panel. In this way, the values are immediately passed from panel to panel until they are read from the `WizardDescriptor` as the wizard closes.

```
public void readSettings(WizardDescriptor model) {
   this.model = model;
   getComponent().addPropertyChangeListener(this);
}
public void storeSettings(WizardDescriptor model) {
   model.putProperty(PlaylistVisualPanel1.PROP_PLAYLIST_NAME,
           getComponent().getPlaylistName());
   model.putProperty(PlaylistVisualPanel1.PROP_GENRE,
           getComponent().getGenre());
   model.putProperty(PlaylistVisualPanel1.PROP_DESCRIPTION,
           getComponent().getDescription());
   model.putProperty(PlaylistVisualPanel1.PROP_IMAGE_PATH,
           getComponent().getImagePath());
}
```

When discussing the visual panel, it was pointed out that values entered by the user must be tested. More specifically, make sure the user has entered a name consisting of at least three characters and that the chosen image is 128×128 pixels in size.

To be informed about changes made to the visual panel—that is, when the user enters a name or chooses an image—register a `PropertyChangeListener` in the `readSettings()` method within the visual panel. Implement the `propertyChange()` method. There, the values entered in the wizard can be tested via the `checkValidity()` method. The `checkValidity()` method verifies criteria, displays a message, and returns relevant values. Via these changes, inform the `WizardDescriptor` so that relevant buttons can be activated or deactivated. The user may only proceed to the next step when the entered data is validated and the `WizardDescriptor` is notified of that fact. Achieve this via the `fireChangeEvent()` method.

```
public void propertyChange(PropertyChangeEvent event) {
   boolean oldState = isValid;
   isValid = checkValidity();
   fireChangeEvent(this, oldState, isValid);
}
private boolean checkValidity() {
   if(getComponent().getPlaylistName().trim().length() < 3) {
      setMessage(bundle.getString("Panel1.Error1"));
      return false;
   } else if(getComponent().getImagePath().length() != 0) {
      ImageIcon img = new ImageIcon(getComponent().getImagePath());
      if(img.getIconHeight()>128 || img.getIconWidth()>128) {
         setMessage(bundle.getString("Panel1.Error2"));
         return false;
      }
   } else if(getComponent().getImagePath().length() == 0) {
      setMessage(bundle.getString("Panel1.Warning1"));
      return true;
   }
   setMessage(null);
   return true;
}
```

To register a `WizardDescriptor` with a wizard panel, the `WizardDescriptor.Panel` interface provides the `addChangeListener()` and `removeChangeListener()` methods. Implement these in the class. Use the `fireChangeEvent()` method to inform all registered listeners. To be efficient, first verify whether the status of panels has changed, so that the `WizardDescriptor` is notified only when changes occur. If the `isValid()` method returns `true`, indicating that the panel has valid status, implement empty methods. The `fireChangeEvent()` method is not called in this case. This scenario applies to the second panel of the example, which always returns `true`.

```
private final Set<ChangeListener> listeners = new HashSet<ChangeListener>(1);
public void addChangeListener(ChangeListener l) {
   synchronized(listeners) {
      listeners.add(l);
   }
}
public void removeChangeListener(ChangeListener l) {
   synchronized(listeners) {
```

```
            listeners.remove(l);
        }
    }
    protected final void fireChangeEvent(
        Object source, boolean oldState, boolean newState) {
        if(oldState != newState) {
            Iterator<ChangeListener> it;
            synchronized (listeners) {
                it = new HashSet<ChangeListener>(listeners).
                        iterator();
            }
            ChangeEvent ev = new ChangeEvent(source);
            while (it.hasNext()) {
                it.next().stateChanged(ev);
            }
        }
    }
}
```

SHARING A BASE PANEL BETWEEN MULTIPLE WIZARD STEPS

If the wizard consists of multiple panels, create a base panel to handle listener logic and helper methods such as the `setMessage()` method. The base panel implements the `WizardDescriptor.Panel<Data>` interface, so specific panels extend the base panel. This approach is outlined in Figure 8-7.

Creating a Wizard from Panels

So far, you've learned about constructing panels that represent steps in a wizard. You saw how a panel consists of a view and a controller, as well as how these work together.

Only one small detail remains to round out your understanding of wizard panels. A wizard is represented by its `WizardDescriptor` class. The `WizardDescriptor` class manages the individual panels. Optionally, one may instantiate the panels and then pass them in to the `WizardDescriptor`. For example, that's how the action class works that is created automatically when using the IDE to create a wizard skeleton. In the interest of encapsulation, composition, and reusability, it is a good idea to create an individual wizard descriptor, extending the `WizardDescriptor` class. Thus, this class itself creates the panels and sets their properties. As done with the action class that starts the wizard, create an instance of the `WizardDescriptor`, which is immediately passed to the `DialogDisplayer`. In this way, a wizard can be called transparently.

For example, create the `PlaylistWizardDescriptor` class, extending the `WizardDescriptor` class (see Listing 8-3). Use the `setPanelsAndSettings()` method to pass in the `Descriptor` for both panels, which are declared as private variables. The panels must be passed via an iterator. One such iterator class is responsible for whole ranges of panels. Use the default `ArrayIterator`. The second parameter for `setPanelsAndSettings()` is a `DataModel`, which the panel receives via the `readSettings()` and `storeSettings()` methods. Here, use data obtained from the wizard for loading and storing purposes. Pass this as a reference to the `PlaylistWizardDescriptor`, which is used as a `DataModel`. Finally, carry out a few configuration tasks.

Listing 8-3. *WizardDescriptor that gathers the panels to form a wizard*

```
public class PlaylistWizardDescriptor extends WizardDescriptor {
   private PlaylistWizardPanel1 p1 = new PlaylistWizardPanel1();
   private PlaylistWizardPanel2 p2 = new PlaylistWizardPanel2();
   public PlaylistWizardDescriptor() {
      List<Panel<WizardDescriptor>> panels =
              new ArrayList<Panel<WizardDescriptor>>();
      panels.add(p1);
      panels.add(p2);
      this.setPanelsAndSettings(new ArrayIterator<WizardDescriptor>(panels), this);
      this.setTitleFormat(new MessageFormat("{0}"));
      this.setTitle(
              NbBundle.getMessage(PlaylistWizardDescriptor.class, "Wizard.Name"));
      putProperty(WizardDescriptor.PROP_AUTO_WIZARD_STYLE, Boolean.TRUE);
      putProperty(WizardDescriptor.PROP_CONTENT_DISPLAYED, Boolean.TRUE);
      putProperty(WizardDescriptor.PROP_CONTENT_NUMBERED, Boolean.TRUE);
      putProperty(WizardDescriptor.PROP_CONTENT_DATA,
              new String[]{p1.getName(), p2.getName()});
   }
}
```

Simpler than the `WizardDescriptor` is the action class that starts the wizard. Create a simple instance of the `PlaylistWizardDescriptor` class and immediately pass it to the `createDialog()` method, as illustrated in the "Custom Dialogs" section earlier in the chapter. This creates a `Dialog` object, which contains a wizard displayed as usual, via the `setVisible()` method (see Listing 8-4).

As the wizard ends, information is gleaned from the button clicked by the user, via the `getValue()` method. The most important point here is how the data is managed. Since the `WizardDescriptor` itself manages data, we can read it directly from the `WizardDescriptor`. The best approach is to use the `getProperties()` method, providing a `Map` with all the properties that have been saved.

Listing 8-4. *Action class that creates and calls a wizard*

```
public final class PlaylistWizardAction implements ActionListener {
   public void actionPerformed(ActionEvent evt) {
      PlaylistWizardDescriptor descriptor = new PlaylistWizardDescriptor();
      Dialog dialog = DialogDisplayer.getDefault().createDialog(descriptor);
      dialog.setVisible(true);
      dialog.toFront();
      if(descriptor.getValue() == WizardDescriptor.FINISH_OPTION) {
         Map<String, Object> props = descriptor.getProperties();
         // Create the playlist with the data stored in props
      }
   }
}
```

Event Handling

In this section, the relationship between the `WizardDescriptor`, wizard panel, and visual panel is revised, focusing on how events and notifications are handled. In the sequence diagram in Figure 8-8, observe the initialization of a wizard and the interaction between the various parts of the wizard as the user enters data.

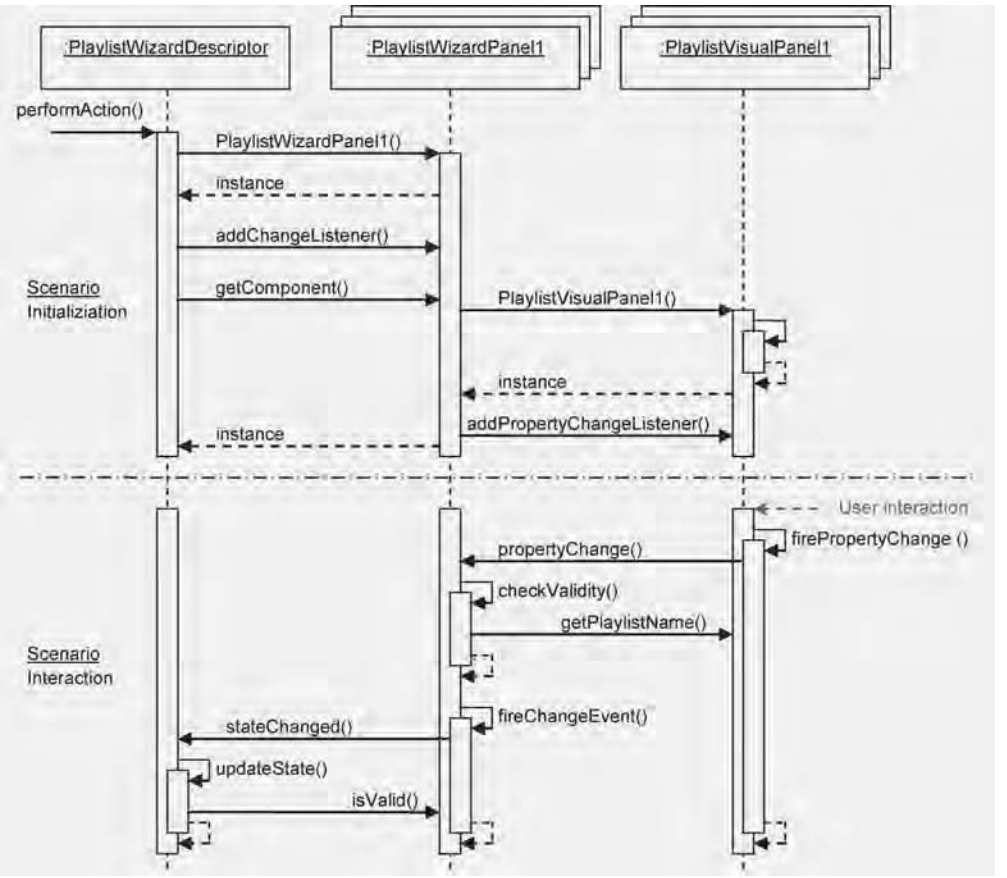


Figure 8-8. *Interaction between the WizardDescriptor, wizard panel, and visual panel*

In the `actionPerformed()` method of the action class that initialized the wizard, an instance of the `PlaylistWizardDescriptor` is created. This `WizardDescriptor` generates its panels and registers a `ChangeListener` for each, so that a notification is fired whenever the status of the panel changes. The visual panel is then obtained via the `getComponent()` method of the wizard panel. This method creates the visual panel on demand and registers a `PropertyChangeListener`, informing of changes made by the user. The `WizardDescriptor` observes the status of its panels via a `ChangeListener`, which in turn observes the status of visual panels via a `PropertyChangeListener`.

When the user types data into a field to which a listener is attached, a `PropertyChangeEvent` is fired, notifying the wizard panel that data has changed. The wizard panel retrieves the data

via the getters and then verifies received data. Depending on the result of the verification, status of the panel is set.

If the status changes, a `ChangeEvent` is fired, notifying the `WizardDescriptor`, which verifies the panel status, calling the `isValid()` method. Depending on the value of the `isValid()` method, the `WizardDescriptor` enables or disables the buttons in the wizard.

Ending a Wizard Prematurely

Depending on the business scenario, it may be useful to allow the user to end the wizard prematurely. Normally, the wizard ends when the Finish button is clicked in the last panel. To allow the user to end the wizard in an earlier panel, implement the interface `WizardDescriptor.FinishablePanel`. The `WizardDescriptor.FinishablePanel` interface provides the method `isFinishPanel()`, which uses the return value `true` when the wizard can be ended. In the example, it is conceivable to implement this interface in the first panel, allowing the user to end the wizard without adding playlist tracks.

Final Verification of Data

Confirming the validity of a panel is provided by the `WizardDescriptor`'s `isValid()` method. The method is called via the `ChangeListener`, as the panel is closed and when notifications are sent. Should additional verifications be required when the user clicks Next or ends the wizard, implement the `WizardDescriptor.ValidatingPanel` interface. The `WizardDescriptor.ValidatingPanel` interface specifies the `validate()` method, in which additional verifications can be performed.

Errors identified in this way are made available as a `WizardValidationException`. The constructor of this exception class receives a `JComponent`, which obtains the focus, so that the user can be shown related error messages. In addition, a failure message can be added, which is then shown in the wizard.

Rather than using the `validate()` method of the `WizardDescriptor.ValdiatingPanel` interface, which is executed asynchronously in the event dispatch thread (EDT) (where no long-running tasks should be performed), use the `WizardDescriptor.AsynchronousValidatingPanel` interface to asynchronously handle verification. Using this interface, the `validate()` method is automatically performed in a separate thread. As a result, the user interface is available to the user, enabling use of the Cancel button to end the process.

Since the asynchronous method is not carried out in the EDT, no access is provided to GUI components in order to read data from them. To that end, the interface provides the `prepareValidation()` method, which is called in the EDT, allowing access to data in the GUI components, while disallowing further change. Using the data retrieved this way, the `validate()` method carries out verification.

Iterators

An iterator within the `WizardDescriptor` creates the whole range of panels. The interface of an iterator of this kind is described by the `WizardDescriptor.Iterator` class. A standard implementation of this interface provides the `WizardDescriptor.ArrayIterator` class, providing panels in a sequential order. The class is also used when passing panels as an array to the `WizardDescriptor` class. However, when giving the user the choice to skip one or more panels

based on entered data, provide your own iterator implementation in the `WizardDescriptor`. The iterator will then handle dynamic creation of panels.

The infrastructure of a dynamic wizard can be created via a wizard in the NetBeans IDE. Returning to the first step of the Wizard wizard, use the Wizard Step Sequence to set the Dynamic option, letting the IDE create an iterator class.

Based on the `WizardDescriptor.Iterator` interface, additional classes are provided. Use the `WizardDescriptor.InstantiatingIterator` interface and its `instantiate()` method to create a set of objects. Alternatively, use the `WizardDescriptor.AsynchronousInstantiatingIterator` with its `instantiate()` method, which is performed asynchronously outside the EDT, when the user clicks the Finish button. Finally, use the `WizardDescriptor.ProgressInstantiatingIterator` interface to show the user a progress bar when the wizard ends, while the `instantiate()` method is processing. In this case, the `instantiate()` method is called in a separate thread, receiving a `ProgressHandle`. Via this class, the status is shown, as is done with the standard progress bar (see Chapter 5).

MultiViews API

Use the MultiViews API and SPI to divide a `TopComponent` across multiple containers. Typically, as the name suggests, this approach is used to provide more than one view for a single `DataObject`. The most common example of this is the NetBeans GUI Builder, in which the user switches from the source view to the design view. The views have, as their basis, a `.java` and a `.form` file. However, a relationship between the views is not mandatory. The container integrates several independent components that are completely independent of each other, displaying different data. The MultiViews SPI can, as a result, be used as a generic framework, rather than one specifically aimed at displaying a single piece of data.

The end result of a multiview includes a drop-down list, allowing the user to switch between the different views. Optionally, one of the views, consisting of any kind of `JComponent`, provides a toolbar displayed next to the drop-down list (see Figure 8-9).

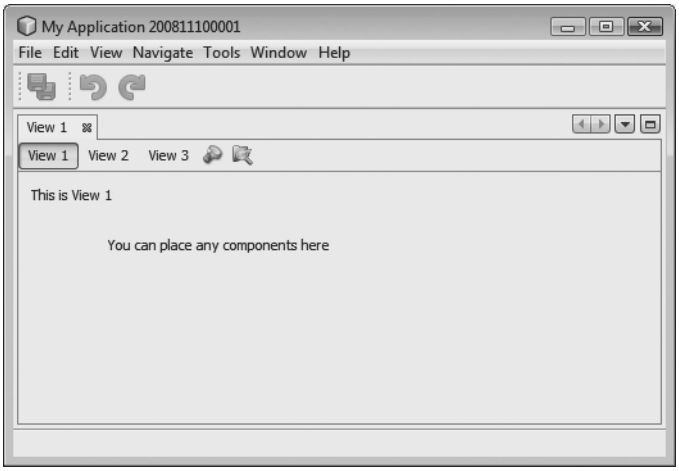


Figure 8-9. *Multiview TopComponent with three views*

Each view consists of independent components, which must be subclasses of `JComponent`. Typically, the base class `JPanel` is used. However, nothing prevents you from using the `TopComponent`, allowing integration of one or more windows into a multiview.

To allow a component to be a view in a multiview `TopComponent`, implement the `MultiViewElement` interface. The methods, specified by this interface, can best be illustrated via a simple example.

```
public class MultiViewPanel1 extends JPanel implements MultiViewElement {
   private JToolBar toolbar = new JToolBar();
   private MultiViewElementCallback callback = null;
   public MultiViewPanel1() {
      initComponents();
      toolbar.add(new Panel1ToolbarAction1());
      toolbar.add(new Panel1ToolbarAction2());
   }
```

To give the view access to the `TopComponent`, use the `setMultiViewCallback()` method to receive a `MultiViewElementCallback`. For example, via this object one obtains the multiview `TopComponent`. To use the callback object in classes, save the data as a private element. An instance of the view is obtained via the `getVisualRepresentation()` method. This method is called whenever the view is activated, meaning that creating the component in this method should be avoided. Normally, use this to access the current components. The toolbar of the current view is obtained via the `getToolbarRepresentation()` method. This method provides a completely created toolbar. Actions in the context menu of the multiview `TopComponent` are obtained from the currently active view, via the `getActions()` method. First, use this method to access the standard actions of a `TopComponent` via the `MultiViewElementCallback` object. Next, add your own actions to the set of standard actions. Use `getLookup()` to obtain the current Lookup—the part of the Lookup that is part of the current multiview `TopComponent` and that is also part of the global context.

```
   public void setMultiViewCallback(MultiViewElementCallback c) {
      callback = c;
   }
   public JComponent getVisualRepresentation() {
      return this;
   }
   public JComponent getToolbarRepresentation() {
      return toolbar;
   }
   public Action[] getActions() {
      if(callback != null) {
         return callback.createDefaultActions();
      } else {
         return new Action[]{};
      }
   }
   public Lookup getLookup() {
      return Lookups.singleton(this);
   }
```

The next methods should be familiar, since they were dealt with in discussions concerning the `TopComponent` class. Via these methods, access is provided to the various states of the view and the multiview `TopComponent`. For example, in the following code, the title of the `TopComponent` is set dynamically, based on the name of the view, whenever the view is opened or activated. The title can be changed via the `MultiViewElementCallback` object, using the `updateTitle()` method.

```
public void componentOpened() {
   callback.updateTitle("View 1");
}
public void componentClosed() {}
public void componentShowing() {}
public void componentHidden() {}
public void componentActivated() {
   callback.updateTitle("View 1");
}
public void componentDeactivated() {}
```

Each view offers its own undo/redo functionality, via the `getUndoRedo()` method. How undo/redo is implemented via the NetBeans API is discussed in Chapter 17. If this support is unwanted, provide `UndoRedo.NONE`, as shown here:

```
public UndoRedo getUndoRedo() {
   return UndoRedo.NONE;
}
```

Finally, implement the `canCloseElement()` method. This method is called on each of the views when the multiview `TopComponent` closes. Only once all the views have provided `CloseOperationState.STATE_OK` can the `TopComponent` be closed. Should a view not be immediately closeable, because (for example) changed data has not yet been saved, provide a `CloseOperationState` object, created via the `MultiViewFactory.createUnsafeCloseState()`. This makes sense only when `CloseOperationHandler` has been implemented, which is passed when the multiview `TopComponent` is created. This handler is available for resolving the `CloseOperationState` objects of all the views. For example, within this handler, a dialog can be shown to the user.

```
   public CloseOperationState canCloseElement() {
      return CloseOperationState.STATE_OK;
   }
}
```

For each view component, create and describe the view via the `MultiViewDescription`. The main point of this class is instantiation of graphic view components, which are created on demand by the `createElement()` method (see Listing 8-5). The method is called once only, when the user opens the view for the first time. The method `getPersistenceType()` is used to specify how the `TopComponent` is saved. Use the constants of the `TopComponent` class, a topic illustrated in Chapter 5.

Listing 8-5. *Description and factory creation of a view*

```
public class MultiViewPanel1Description
        implements MultiViewDescription, Serializable {
   public MultiViewElement createElement() {
      return new MultiViewPanel1());
   }
   public String preferredID() {
      return "PANEL_1";
   }
   public int getPersistenceType() {
      return TopComponent.PERSISTENCE_NEVER;
   }
   public String getDisplayName() {
      return "View 1";
   }
   public Image getIcon() {
      return null;
   }
   public HelpCtx getHelpCtx() {
      return HelpCtx.DEFAULT_HELP;
   }
}
```

Finally, there remains the creation of a multiview `TopComponent` from independently created views. To that end, the MultiViews SPI provides a factory class, which is the `MultiViewFactory` class. It contains methods permitting the creation of `TopComponent`s or `CloneableTopComponent`s, depending on need.

```
MultiViewDescription dsc[] = {
   new MultiViewPanel1Description(),
   new MultiViewPanel2Description(),
   new MultiViewPanel3Description()};
TopComponent tc = MultiViewFactory.createMultiView(dsc, dsc[0]);
tc.open();
```

First, create an array of `MultiViewDescription` classes, representing each of the views. Pass this array to the `createMultiView()` method. The second parameter contains the initially active view. An optional third parameter includes an implementation of the `CloseOperationHandler` discussed earlier, for the creation of a `CloseOperationState` object. The multiview `TopComponent` is then opened via the `open()` method.

To give access from outside to the views, use the static method `MultiViews.findMultiViewHandler()` to create a `MultiViewHandler` for a view `TopComponent`. Via this handler, you can, for example, access the currently selected view or all available views at once.

Visual Library API

The NetBeans Visual Library API is a generic library for displaying different structures. In particular, the library is most suited to graph-oriented representations. The Visual Library API

(version 2.0) is part of the NetBeans Platform and is used by the NetBeans IDE in numerous modules and areas, such as visual modeling of MIDlets in a JME application, as shown in Figure 8-10. To use the Visual Library API, you need only define a dependency on the module (under Libraries within the module's Properties dialog), as previously noted in examples of other modules.

Structure of the Visual Library API

The components of the Visual Library API, like Swing, are structured and managed as a tree. The superclass of all graphic components is the `Widget` class. If you consider Figure 8-10, then the three components (Mobile Device, form, and loginScreen), as well as the edges connecting components, are all widgets. A widget can also be a container for other widgets. Each widget has a position relative to its parent widget. The `Widget` superclass is responsible for presenting the border and background of a widget in addition to managing properties such as color and transparency. Like a Swing container, a widget has a certain layout responsible for the positioning of its child widgets. Widgets depend upon each other in order to be notified about changes. A widget can also be linked to a series of actions that are executed when specific user events occur.

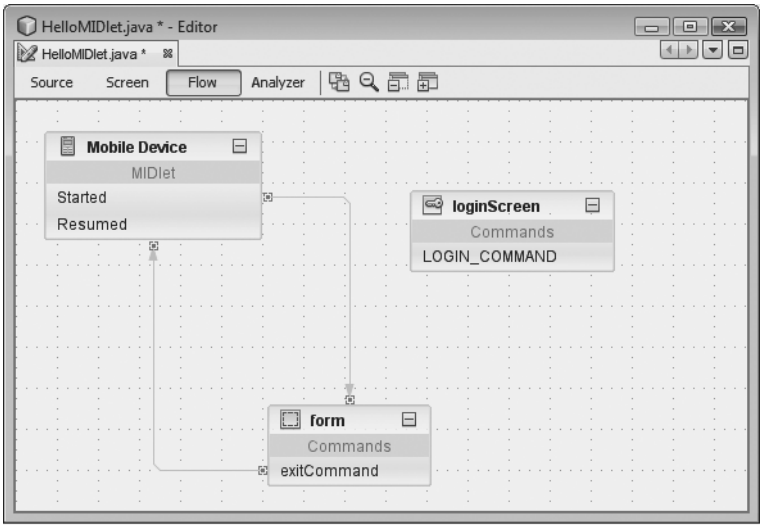


Figure 8-10. *Visual model of a graph-oriented structure using the Visual Library API*

The Widget Classes

All graphic components of the Visual Library API are subclasses of `Widget`, which manages and provides basic features and functionalities such as layout, background, and font. A Widget is a graphic primitive equivalent to the `JComponent` class in Swing. From `Widget`, numerous classes are derived, making relevant `Widget` implementations available for most applications. This inheritance hierarchy is represented in Figure 8-11, and the descriptions of these various `Widget` classes are listed in Table 8-4. The most important are dealt with in more detail in the following sections. For more exhaustive descriptions of these classes, see the Visual Library API documentation, found within the Javadoc of the Visual Library.

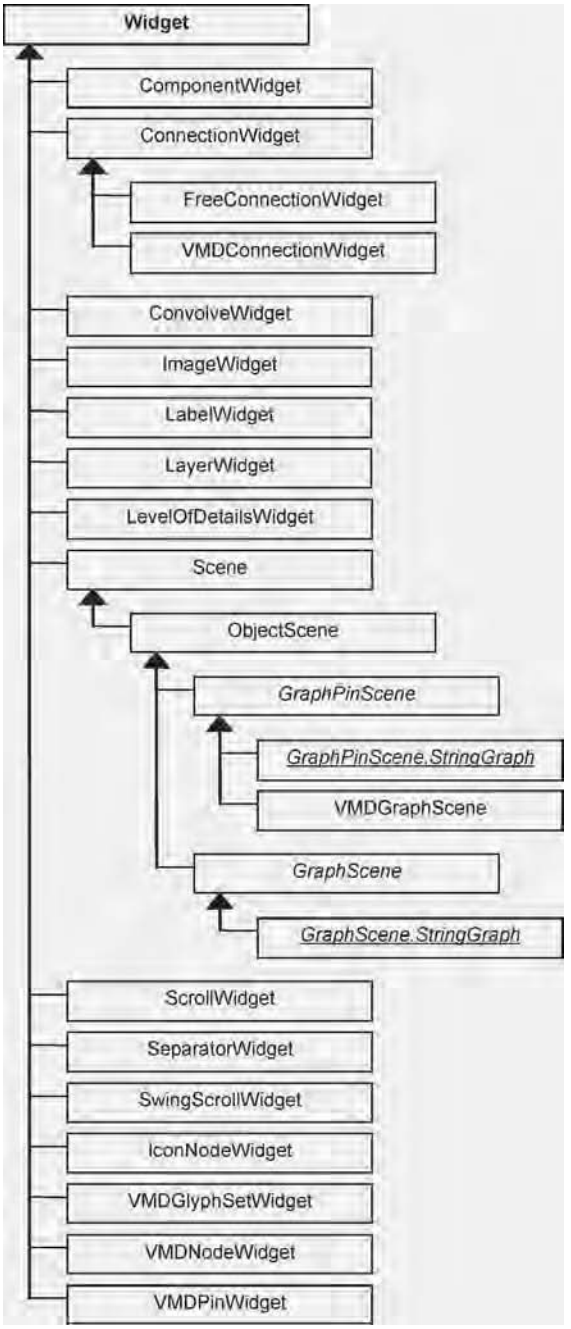


Figure 8-11. *Widget inheritance hierarchy of the Visual Library API*

Table 8-4 provides an overview of the features and functionalities of various `Widget` implementations.

Table 8-4. *The descriptions of the different Widget subclasses*

Class	Description
`ComponentWidget`	Using a `ComponentWidget`, AWT/Swing components are used within a scene. This widget serves as a placeholder and is responsible for displaying and updating the contained component.
`ConnectionWidget`	A `ConnectionWidget` is used to connect two points determined by anchors. It is responsible for the presentation of the connecting line, as well as control points, endpoints, and anchors. Control points, resolved by a router, specify the path of a connecting line.
`ConvolveWidget`	A `ConvolveWidget` applies a convolve filter to a child element.
`ImageWidget`	With an `ImageWidget`, images are represented within a scene.
`LabelWidget`	With this widget, text is displayed. Text can be represented in four different horizontal and vertical alignments.
`LayerWidget`	A `LayerWidget` is a transparent widget, functioning similarly to a `JGlassPane`. A scene uses several such layers to organize different types of widgets.
`LevelOfDetailsWidget`	A `LevelOfDetailsWidget` serves as container for its child widgets and determines their visibility by the zoom factor of the scene.
`Scene`	The `Scene` widget is the root element of the current hierarchy of displayed widgets. It is responsible for control and representation of the whole rendered area. This class makes a view of the scene available in the form of a `JComponent` instance, which is then embedded into any Swing component. We will look at this important class in more detail in the "The Scene: The Root Element" section later in the chapter.
`ScrollWidget`	A `ScrollWidget` is a scrollable container whose functionality corresponds to a `JScrollPane`. The scroll bars are only shown when needed.
`SeparatorWidget`	This widget represents a separator whose thickness and orientation can be set.
`SwingScrollWidget`	This widget, like a `ScrollWidget`, represents a scrollable area, but the `JScrollBar` class is used for the scroll bars.
`IconNodeWidget`	An `IconNodeWidget` represents both an image and label that can alternatively be placed below or next to the image.

Dependencies

Dependencies can be defined between individual widgets. You are thereby able to respond to changes in position or size of other widgets. This dependency is realized by a listener registered on the widget. In addition, the `Widget` class makes available two methods, `addDependency()` and `removeDependency()`, in order that a listener can be added or removed. A

listener is specified by the interface `Widget.Dependency`. The listener must implement the method `revalidateDependency()`, which is called by the respective widget on a change of position or size. By this method, you call the `revalidate()` method of widgets that are dependent on other widgets.

Border

Each widget has border. By default, this is an empty border, represented by the class `EmptyBorder`. Another border can be specified with the `setBorder()` method. A border is specified by the interface `Border`. And this interface is implemented by numerous `Border` classes. In addition to the `EmptyBorder` class, there are also `LineBorder`, `BevelBorder`, `DashedBorder`, `ImageBorder`, `ResizeBorder`, `RoundedBorder`, and `SwingBorder` classes. A `ResizeBorder` adds eight points to the edges of your widget, which are used to change its size. Furthermore, the `SwingBorder` class allows you to use any Swing `javax.swing.border.Border` implementation. Last at your disposal is the `CompositeBorder` class, with which you can combine any number of the `Border` instances mentioned.

The borders, however, are not created directly, but via a factory. This is the `BorderFactory` class that provides you with numerous methods with which you can create various border types. Instances produced by the factory can be simultaneously shared by several widgets. If you wish to use the same border between several widgets, you need only create one instance of it.

Layout

A widget (like a Swing container) has a special layout, managed and specified by a Layout Manager. A layout is defined by the interface `Layout` and is responsible for arrangement of the child widgets. Four different variants of layouts are available, produced by the `LayoutFactory` class and added to a widget with the `setLayout()` method.

AbsoluteLayout

With `AbsoluteLayout`, child widgets are arranged according to the coordinates supplied by `getPreferredLocation()`. The size of child widgets corresponds to the proportions provided by `getPreferredBounds()`. If the two methods supply `null`, the position becomes `(0, 0)` and the size becomes `(0, 0, 0, 0)`. This is the default layout used by a widget. This layout is generated with the following:

```
Layout al = LayoutFactory.createAbsoluteLayout();
```

FlowLayout

The `FlowLayout` arranges widgets in a sequential order in horizontal or vertical directions. Four different alignments can be selected: left top, center, right bottom, and justified. Furthermore, the gap between individual widgets can be specified. The size of widgets corresponds to the value that `getPreferredBounds()` returns. The following methods are available for the creation of this layout; alternatively, supply the alignment as a `LayoutFactory.SerialAlignment` type along with the gap:

```
Layout hfl = LayoutFactory.createHorizontalFlowLayout();
Layout vfl = LayoutFactory.createVerticalFlowLayout();
```

CardLayout

A `CardLayout` shows only the currently active widget, which is specified by the method `setActiveCard()`. The size of the active widget is determined by `getPreferredBounds()`. All other widgets are represented by the size `(0, 0, 0, 0)`, thus making them practically invisible. Determine the currently active widget by the method `getActiveCard()`. Create this layout with the following:

```
Layout cl = LayoutFactory.createCardLayout();
```

and specify the active widget with the following call:

```
LayoutFactory.setActiveCard(Widget parent, Widget activate);
```

To switch the active widget, you can use the `SwitchCardAction` class.

OverlayLayout

The `OverlayLayout` determines the minimum area containing all child widgets. Both the widget that contain this layout and all child widgets are set to the size of this determined area and arranged on top of each other. The last child widget displays at the top. You create this layout as follows:

```
Layout ol = LayoutFactory.createOverlayLayout();
```

Events and Actions

A widget knows its position, size, and, content, but not information about its behavior. The behavior of widgets is influenced by actions added arbitrarily to a widget. These actions are specified by the interface `WidgetAction`, which defines a number of event methods. These methods are called by corresponding events, such as clicking a mouse button on the widget the action is assigned to. Implementation of the action class executes desired behaviors such as moving a widget by drag-and-drop.

Like borders and layouts, actions are created by a factory. This is the `ActionFactory` class. These actions are managed within a widget by the `WidgetAction.Chain` class. This class receives user events and forwards these to the appropriate actions it manages. Each widget has an instance of this class, which is obtained by `getAction()`. With the methods `addAction()` and `removeAction()`, the `WidgetAction.Chain` class adds or removes actions to or from widgets.

Some of the factory methods of the `ActionFactory` class require a provider as a parameter. A provider implements a specific behavior for an action. For some actions (e.g., the `EditAction`), a provider implementation that is executed on double-clicking the appropriate widget must be specified. For other actions, like the `MoveAction`, specify a provider if you wish the behavior to deviate from the default. These providers are specified through special interfaces such as `EditProvider` or `HoverProvider`.

The real advantage or purpose for managing a widget's actions in a `WidgetAction.Chain` class is grouping. For example, in some applications, you permit only certain actions for a scene. Widgets may be moved, but not edited. This functionality is provided by setting the current tool of a scene using the `setActiveTool()` method in the `Scene` class. The `Widget` class manages different actions in separate `WidgetAction.Chain` instances depending on the currently active tool. Previously, access was granted to actions via the `getAction()` method. This supplied the default `WidgetAction.Chain` instance, which is also the case if no tool is set

(`setActiveTool(null)`). Now you can use a variant of the `getAction(String activeTool)` method, to which you supply the name of the tool and obtain the relevant `WidgetAction.Chain` instance.

AcceptAction

This action is for the treatment of drag-and-drop operations. An `AcceptProvider` implementation must be provided to create the action. The `AcceptProvider` interface specifies the method `isAcceptable()`, which allows specifying whether a drop operation on this widget is allowed, as well as the method `accept()`, with which you accomplish the drop operation.

```
ActionFactory.createAcceptAction(AcceptProvider p);
```

ActionMapAction

This action provides a context menu, displayed by right-clicking the widget. You create the action using the default method without parameters, whereby actions for the menu are inferred from the `ActionMap` of the scene view. Additionally, there is an option to supply the method with an `InputMap` and `ActionMap` used for production of the menu.

```
ActionFactory.createActionMapAction();
ActionFactory.createActionMapAction(InputMap i, ActionMap a);
```

AddRemoveControlPointAction

This action can only be used by `FreeConnectionWidget` widgets. With it, you add or remove control points by double-clicking them. You also indicate the sensitivity used.

```
ActionFactory.createAddRemoveControlPointAction();

ActionFactory.createAddRemoveControlPointAction(
   double createSensitivity,
   double deleteSensitivity);
```

MoveAction/AlignWithMoveAction

With the `MoveAction`, a widget can be moved by drag-and-drop. Please note that this action only functions if the parent widget has an `AbsoluteLayout`. Similarly, the `AlignWithMoveAction` behaves like the `MoveAction`. In contrast, however, additional "snapping" with other widgets occurs. All widgets against which alignment is checked are gathered using an `AlignWithWidgetCollector` instance or set through a `LayerWidget`. In the second case, the alignment of all child widgets within each layer is checked.

```
ActionFactory.createMoveAction();

ActionFactory.createMoveAction(
   MoveStrategy             strategy,
   MoveProvider             provider);

ActionFactory.createAlignWithMoveAction(
   AlignWithWidgetCollector collector,
```

```
   LayerWidget              interactionLayer,
   AlignWithMoveDecorator   decorator);

ActionFactory.createAlignWithMoveAction(
   LayerWidget              collectionLayer,
   LayerWidget              interactionLayer,
   AlignWithMoveDecorator   decorator);
```

ResizeAction/AlignWithResizeAction

With the `ResizeAction`, you change the size of widgets, while `AlignWithResizeAction` reviews snapping against other widgets. With widgets needing their alignment checked against others, provide either an `AlignWithWidgetCollector` instance or a `LayerWidget`.

```
ActionFactory.createResizeAction();

ActionFactory.createResizeAction(
   ResizeStrategy             strategy,
   ResizeProvider             provider);

ActionFactory.createResizeAction(
   ResizeStrategy             strategy,
   ResizeControlPointResolver resolver,
   ResizeProvider             provider);

ActionFactory.createAlignWithResizeAction(
   AlignWithWidgetCollector   collector,
   LayerWidget                interactionLayer,
   AlignWithMoveDecorator     decorator);

ActionFactory.createAlignWithResizeAction(
   LayerWidget                collectionLayer,
   LayerWidget                interactionLayer,
   AlignWithMoveDecorator     decorator);
```

ZoomAction/CenteredZoomAction

With these actions, the zoom of the whole scene is changed with the mouse wheel. These actions are not added to a widget, but directly to a scene.

```
ActionFactory.createZoomAction();
ActionFactory.createZoomAction(double zoom, boolean animated);
ActionFactory.createCenteredZoomAction(double zoomMultiplier);
```

ConnectAction/ExtendedConnectAction/ReconnectAction

With the `ConnectAction`, you can connect, with the assistance of `ConnectionWidget`s, two widgets. This action is added to the widget from which the connection is to be made. Only with a `ConnectProvider` instance, following a check of the source and target widgets for the desired connection, can a connection be made. Optionally, supply a specific graphic for the connecting

line using a ConnectDecorator. With the ExtendedConnectAction, a connection can be made only as long as the Ctrl key pressed. This action is meant for cases where conflicts with other actions occur—for example, if you want to use the ConnectAction and the MoveAction at the same time. In these cases, use the ExtendedConnectAction.

```
ActionFactory.createConnectAction(
   LayerWidget interactionLayer,
   ConnectProvider provider);

ActionFactory.createConnectAction(
   ConnectDecorator decorator,
   LayerWidget interactionLayer,
   ConnectProvider provider);

ActionFactory.createExtendedConnectAction(
   LayerWidget interactionLayer,
   ConnectProvider provider);

ActionFactory.createExtendedConnectAction(
   ConnectDecorator decorator,
   LayerWidget interactionLayer,
   ConnectProvider provider);

ActionFactory.createReconnectAction(
   ReconnectProvider provider);

ActionFactory.createReconnectAction(
   ReconnectDecorator decorator,
   ReconnectProvider provider);
```

CycleFocusAction/CycleObjectSceneFocusAction

You shift the focus between the widgets of a scene using the Tab key, either forward or backward. With the CycleFocusAction, you specify the behavior with which the preceding or following widget is focused using a CycleFocusProvider. In the case of the CycleObjectSceneFocusAction, which is applied to an ObjectScene, the order of the focusing is determined by the return value of getIdentityCode().

```
ActionFactory.createCycleFocusAction(CycleFocusProvider p);
ActionFactory.createCycleObjectSceneFocusAction();
```

EditAction/InplaceEditorAction

In order to edit a widget by double-clicking, add an EditAction. The triggered behavior is implemented by an EditProvider. A further option supplies an in-place editor that is displayed upon double-clicking. For this, use the InplaceEditorAction, whereby the editor can be any JComponent subclass. For example, with an IconNodeWidget or a LabelWidget, this would typically be a JTextField.

```
ActionFactory.createEditAction(
   EditProvider provider);

ActionFactory.createInplaceEditorAction(
   InplaceEditorProvider provider);

ActionFactory.createInplaceEditorAction(
   TextFieldInplaceEditor editor);

ActionFactory.createInplaceEditorAction(
   TextFieldInplaceEditor editor,
   EnumSet expansionDirections);
```

ForwardKeyEventsAction

With this action, you can forward keyboard events to other widgets.

```
ActionFactory.createForwardKeyEventsAction(
   Widget forwardToWidget,
   String forwardToTool);
```

HoverAction

With the `HoverAction`, you can react to a mouse pointer that is hovering over a widget. How the widget behaves is specified by a `HoverProvider` or a `TwoStateHoverProvider`.

```
ActionFactory.createHoverAction(HoverProvider p);
ActionFactory.createHoverAction(TwoStateHoverProvider p);
```

MoveControlPointAction/FreeMoveControlPointAction/OrthogonalMoveControlPointAction

These actions move the control points of the connecting line of a `ConnectionWidget`. The `OrthogonalMoveControlPointAction` is used when a `ConnectionWidget` has an `OrthogonalSearchRouter`. The `FreeMoveControlPointAction` has no restrictions on positioning the points.

```
ActionFactory.createMoveControlPointAction(MoveControlPointProvider provider);
ActionFactory.createFreeMoveControlPointAction();
ActionFactory.createOrthogonalMoveControlPointAction();
```

PanAction

If the view of a scene is contained within a `JScrollPane`, the `PanAction` allows scrolling the view by moving the mouse while the middle button is pressed. This action is usually added to a scene.

```
ActionFactory.createPanAction();
```

PopupMenuAction

Use the `PopupMenuAction` to provide a widget with a context menu. This requires implementing a `PopupMenuProvider` within which you provide a `JPopupMenu` instance.

```
ActionFactory.createPopupMenuAction(
   PopupMenuProvider more provider);
```

SelectAction/RectangularSelectAction

The `SelectAction` is similar to the `EditAction`; however, this event is the result of a single click. The logic is implemented within a `SelectProvider`, inside which you also specify whether a widget is selected. The `RectangularSelectAction` is usually added to an `ObjectScene` or a `LayerWidget` with which you select widgets by drawing rectangles around them.

```
ActionFactory.createSelectAction(SelectProvider provider);

ActionFactory.createRectangularSelectAction(
   ObjectScene scene,
   LayerWidget interactionLayer);

ActionFactory.createRectangularSelectAction(
   RectangularSelectDecorator decorator,
   LayerWidget interactionLayer,
   RectangularSelectProvider provider);
```

SwitchCardAction

This action is required for switching between widgets that are in a `CardLayout`.

```
ActionFactory.createSwitchCardAction(Widget cardLayoutWidget);
```

The Scene: The Root Element

The components of the Visual Library API—i.e., widgets—are arranged and managed in a hierarchical tree structure. This means widgets, in turn, can contain other widgets. The `Scene` class, which itself is a widget, represents the container for all subsequent elements and therefore is the root element of the hierarchy (see Figure 8-11). Graphically, a scene is represented by a view, which is a simple `JComponent` instance. This is then typically added to a `JScrollPane`. One always starts with a scene, to which are added further widgets in hierarchical arrangement, depending upon the application's needs. Listing 8-6 illustrates this.

Listing 8-6. *Creating a scene and adding widgets*

```
final class SceneTopComponent extends TopComponent {
   private JScrollPane scenePane = new JScrollPane();
   private Scene sc = new Scene();

   private SceneTopComponent() {
      scenePane.setViewportView(sc.createView());
      LayerWidget layer1 = new LayerWidget(sc);
      sc.addChild(layer1);
      ImageWidget w1 = new ImageWidget(sc,
         ImageUtilities.loadImage("com/galileo/netbeans/module/node.gif"));
      layer1.addChild(w1);
      ImageWidget w2 = new ImageWidget(sc,
         ImageUtilities.loadImage("com/galileo/netbeans/module/node.gif"));
      layer1.addChild(w2);
      LayerWidget layer2 = new LayerWidget(sc);
      sc.addChild(layer2);
      ImageWidget w3 = new ImageWidget(sc,
         ImageUtilities.loadImage("com/galileo/netbeans/module/node2.gif"));
      layer2.addChild(w3);
      WidgetAction ma = ActionFactory.createMoveAction();
      w1.getActions().addAction(ma);
      w2.getActions().addAction(ma);
      w3.getActions().addAction(ma);
   }
}
```

We create the scene as a private variable. The `createView()` method provides a view for this scene of the type `JComponent`, and can be embedded into any Swing container. So that the scene or view is not limited to a certain size, we add this to a `JScrollPane`. We now hierarchically attach our widgets to the scene after first creating a `LayerWidget` that acts like a `JGlassPane` and to which we add two `ImageWidget`s. To illustrate the grouping and alignment of widgets, we create a further `LayerWidget` instance and add an additional `ImageWidget`. The added `LayerWidget` and contained `ImageWidget`s are then assigned to the scene. So that the widgets can be moved within the scene, we add a `MoveAction` instance created with the `ActionFactory`. An example can be seen in Figure 8-12.

Overview

In order to establish fast navigation within larger scenes, a scene offers us an overview in the form of an interactive `JComponent`. This is a satellite view created using the `createSatelliteView()` method. If the view of your scene is embedded in a `JScrollPane` and the scene is larger than that of the displayed area, navigate the scene by moving the gray frame present in the overview to update the view (see Figure 8-12).

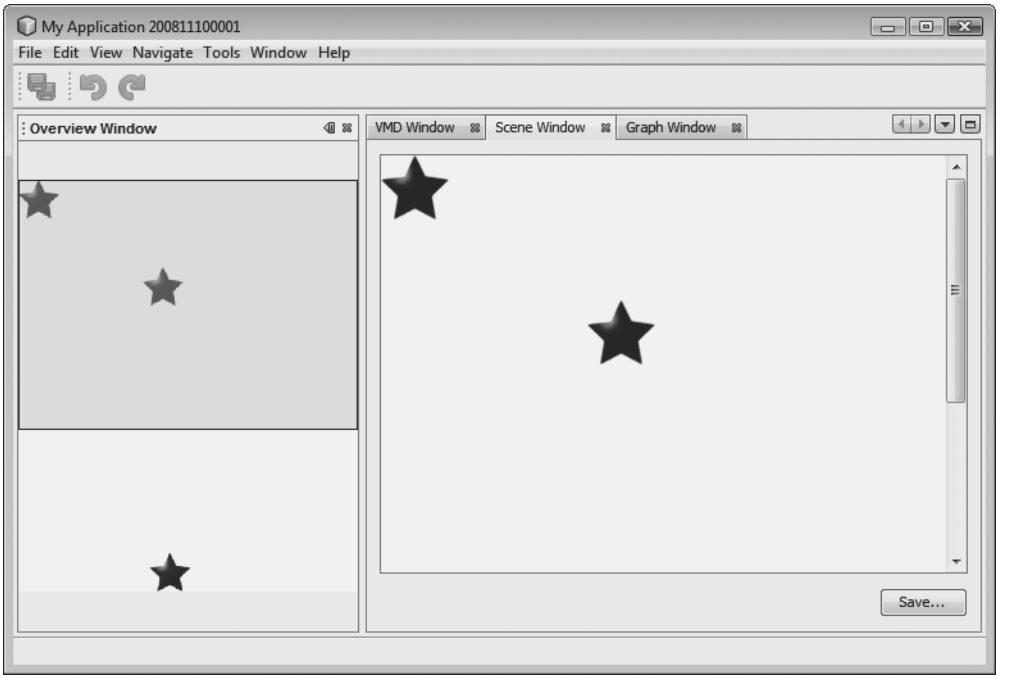


Figure 8-12. *An overview for a scene can be created and also used for navigation.*

Exporting a Scene

In few steps and with the assistance of some Java utilities, you can export a scene generated with the Visual Library API into an image contained within a PNG file. To this end, first produce a `BufferedImage` object, into which is written the graphic data. Specify the size of this `Image` instance using the current size of the view of the respective scene, to assure the complete contents are stored. From this object, we avail ourselves of the `Graphics2D` context with which we feed data into the buffer of the `BufferedImage` object. This context is then supplied to the `paint()` method of the `Scene` object so that content is written to the buffer of the `BufferedImage` instance rather than the screen (see Listing 8-7). Following this, dispose of the context so that the resources can be released. With a `JFileChooser`, request a file name and, if necessary, append the appropriate suffix. Once this is done, utilize the `ImageIO` class that looks for an `ImageWriter` for PNG files, using it to write the data of the `BufferedImage` object into the selected file.

Listing 8-7. *Exporting a scene into a PNG file*

```
private Scene sc = new Scene();

public void exportScene() {
   BufferedImage img = new BufferedImage(
      sc.getView().getWidth(),
```

```
            sc.getView().getHeight(),
            BufferedImage.TYPE_4BYTE_ABGR);

        Graphics2D graphics = img.createGraphics();
        sc.paint(graphics);
        graphics.dispose();

        JFileChooser chooser = new JFileChooser();
        chooser.setFileFilter(new FileNameExtensionFilter(
            "Portable Network Graphics (.png)", "png"));

        if(chooser.showSaveDialog(sc.getView()) == JFileChooser.APPROVE_OPTION) {
            File f = chooser.getSelectedFile();
            if (!f.getName().toLowerCase().endsWith(".png")) {
                f = new File(f.getParentFile(), f.getName() + ".png");
            }

            try {
                ImageIO.write(img, "png", file);
            } catch (IOException e) {
                Logger.getLogger(getName()).warning(e.toString());
            }
        }
    }
```

ObjectScene: Model-View Relationship

The Visual Library API provides only the constituent components making up a view. That is, a widget only possesses information about presentation or flow of data. What a widget does not possess is a data model. This is where the `ObjectScene` class comes into play, representing an extension of the `Scene` class. The function of this class is to manage widget mapping to an associated data model, which can be any object displayed in a view. The class `ObjectScene` makes available methods allowing widget assignment to a data model. It's also possible to determine the data model registered to a widget and vice versa. Besides mapping the data model to widgets, the `ObjectScene` class also provides information about the current state of a widget or data model, represented by the `ObjectState` class.

Data models are stored internally in a `Map`. For identification and comparison purposes, data models use the `equals()` method. Ensure that your data model contains a meaningful implementation of this method and note that each unique data model can only be added once. Thus, if the data model `d1` is in an `ObjectScene`, and a second, `d2`, is added, whereby `d1.equals(d2) == true` applies, an exception is raised.

Table 8-5 summarizes the most important methods of the `ObjectScene` class and their functions.

Table 8-5. *The most important methods of the ObjectScene class*

Method	Description

`void addObject(`
`  Object model,`
`  Widget...widgets)`
	With the `addObject()` method, several widgets and their associated data models are added to a scene.

`void removeObject(` `  Object model)`	You can remove a known data model with the `removeObject()` method. Note that the associated widget is not removed, but eliminated separately with the `removeChild()` method.
`Object findObject(` `  Widget widget)`	Use the `findObject()` method in order to find the data model belonging to a certain widget.
`Widget findWidget(` `  Object model)`	This method is the counterpart to `findObject()`, and finds the widget for a given data model.
`List<Widget> findWidgets(` `  Object model)`	If several widgets are assigned to a model, these are returned with `findWidgets()`.
`ObjectState getObjectState(` `  Object model)`	To receive the current state of a model, use the `getObjectState()` method. If the state of a data model changes, the state of the widget is changed accordingly. The opposite does not occur. The status of a widget is determined with the `getState()` method.

Graph

In order to facilitate the creation of graphs (see Figure 8-10 for an example of flow designers)—i.e., the production of nodes and associated edges—the API based on the `ObjectScene` class introduced in the previous section makes available the classes `GraphScene` and `GraphPinScene`. For these classes, an example is shown hereafter, the basis of which clarifies the practical significance of the `ObjectScene` class.

Both `GraphScene` and `GraphPinScene` are abstract classes, whose only task is the management of data models and widgets. The creation of widgets is dependent on subclasses and is reliant upon data models. This is achieved by overriding the relevant abstract methods within the subclasses. The types of data models are defined by templates and can vary in each case for nodes, edges, and pins. In the simplest case, as in this example, use the type `String`. For the nodes and edges, we provide a separate `LayerWidget` and add these to the scene.

```
public class MyGraphPinScene extends GraphPinScene<String, String, String> {
   private LayerWidget mainLayer;
   private LayerWidget connectionLayer;

   public MyGraphPinScene() {
      mainLayer = new LayerWidget(this);
```

```
        addChild(mainLayer);
        connectionLayer = new LayerWidget(this);
        addChild(connectionLayer);
    }
```

The `attachNodeWidget()` method is responsible for creating nodes. Therefore, we use the `IconNodeWidget` class here in the example. The `ImageWidget` class could also be used. However, we also manage the accompanying pins with the `IconNodeWidget` class. This is done by using `LabelWidget`, which is accessed by the `getLabelWidget()` method. So that the pins can be arranged and presented correctly, a `FlowLayout` is defined for this widget. In order to move the node, a `MoveAction` instance is added. Last, we add the node to the `MainLayer` and return it.

```
    protected Widget attachNodeWidget(String node) {
        IconNodeWidget widget = new IconNodeWidget(this);

        widget.setImage(
                ImageUtilities.loadImage("com/galileo/netbeans/module/node.gif"));
        widget.getLabelWidget().setLayout(
                LayoutFactory.createHorizontalFlowLayout(
                LayoutFactory.SerialAlignment.JUSTIFY, 5));
        widget.getActions().addAction(ActionFactory.createMoveAction());

        mainLayer.addChild(widget);
        return widget;
    }
```

The `attachEdgeWidget()` method is responsible for creating edges. But we use the `ConnectionWidget` class to enable using a router, so edges are not simply drawn as straight lines between nodes possibly intersecting with other nodes or edges. A router permits us to have a series of `LayerWidget`s, whose widgets are not to be crossed. Accordingly, the router determines paths for edges, so that no intersections occur (see Figure 8-13). Such routers are created with `RouterFactory`. The configured edges are then added to the `ConnectionLayer` and returned.

```
protected Widget attachEdgeWidget(String edge) {
    ConnectionWidget widget = new ConnectionWidget(this);
    widget.setTargetAnchorShape(AnchorShape.TRIANGLE_FILLED);
    widget.setRouter(RouterFactory.createOrthogonalSearchRouter(
        mainLayer, connectionLayer));
    connectionLayer.addChild(widget);
    return widget;
}
```

Pins are created with the `attachPinWidget()` method. A *pin* is an input or output of a node, to which an edge can be connected (the red points in Figure 8-13 represent pins). A pin is assigned to a node, which may possess multiple pins. The data model for the pin and the node to which it will be added is received within those parameters. The `findWidget()` method assists in determining the widget associated with the node to which the created pin is added.

```
protected Widget attachPinWidget(String node, String pin) {
   ImageWidget widget = new ImageWidget(this,
      ImageUtilities.loadImage("com/galileo/netbeans/module/pin.gif"));
   IconNodeWidget n = (IconNodeWidget) findWidget(node);
   n.getLabelWidget().addChild(widget);
   return widget;
}
```

Last, override the `attachEdgeSourceAnchor()` and `attachEdgeTargetAnchor()` methods (see Listing 8-8). With these, the start and endpoints of an edge are specified. Here, first determine the edge to which the pin is to be connected with the `findWidget()` method. Then provide the `AnchorFactory` with an anchor point to the pin, which has likewise been determined by the `findWidget()` method, and add this to the edge.

Listing 8-8. *Implementation of a GraphPinScene class*

```
protected void attachEdgeSourceAnchor(
   String edge, String oldPin, String pin) {
   ConnectionWidget c = (ConnectionWidget) findWidget(edge);
   Widget widget = findWidget(pin);
   Anchor a = AnchorFactory.createRectangularAnchor(widget);
   c.setSourceAnchor(a);
}
protected void attachEdgeTargetAnchor(
   String edge, String oldPin, String pin) {
   ConnectionWidget c = (ConnectionWidget) findWidget(edge);
   Widget widget = findWidget(pin);
   Anchor a = AnchorFactory.createRectangularAnchor(widget);
   c.setTargetAnchor(a);
}
```

Similarly, you could create an implementation of the `GraphScene` class, which has no pins. Here, edges are connected directly to the node rather than a pin. The advantage of the implementation just fashioned now becomes apparent. As with a normal scene, create an instance and add its view to a `JScrollPane`. What remains is the creation of individual widgets (see Listing 8-9). You need only supply the data model (a `String` object) to the methods `addNode()`, `addPin()`, or `addEdge()`. These internally call the methods implemented by us to create the widgets and produce a visual representation of the data model.

Listing 8-9. *Use of a GraphPinScene*

```
final class GraphTopComponent extends TopComponent {
   private GraphTopComponent() {
      MyGraphPinScene scene = new MyGraphPinScene();
      scenePane.setViewportView(scene.createView());
      scene.addNode("Node 1");
```

```
        scene.addNode("Node 2");
        scene.addNode("Node 3");
        scene.addPin("Node 1", "p1");
        scene.addPin("Node 2", "p2");
        scene.addPin("Node 2", "p3");
        scene.addPin("Node 3", "p4");
        scene.addEdge("Edge 1");
        scene.addEdge("Edge 2");
        scene.setEdgeSource("Edge 1", "p1");
        scene.setEdgeTarget("Edge 1", "p2");
        scene.setEdgeSource("Edge 2", "p3");
        scene.setEdgeTarget("Edge 2", "p4");
        GridGraphLayout<String,String> layout = new GridGraphLayout<String,String>();
        SceneLayout sceneLayout = LayoutFactory.createSceneGraphLayout(scene, layout);
        sceneLayout.invokeLayout();
    }
}
```

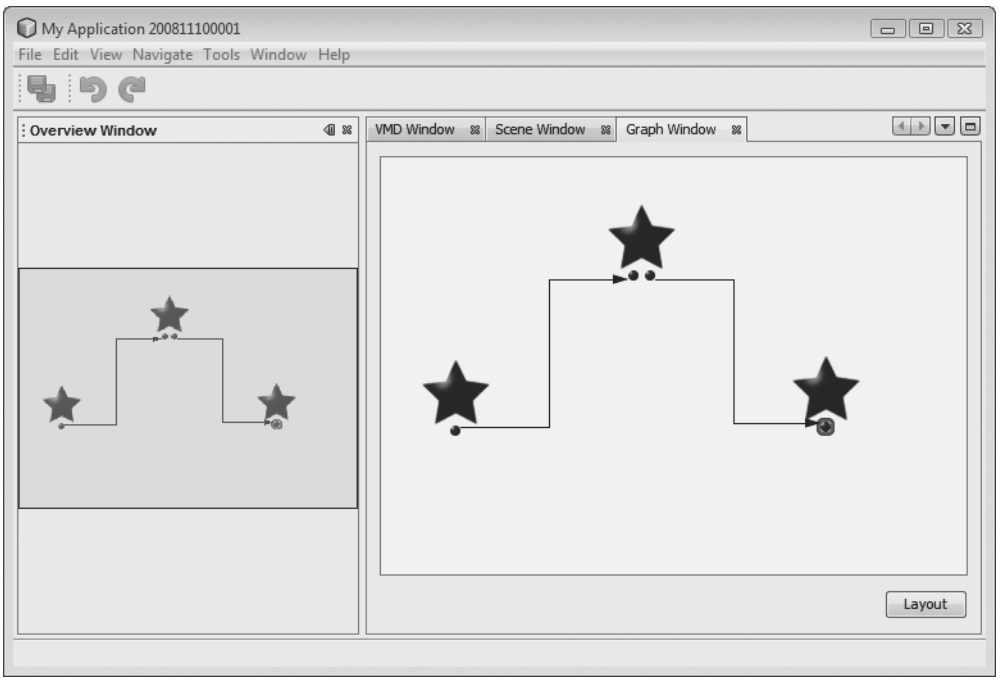


Figure 8-13. *Example of creating a graph using a GraphPinScene implementation*

VMD: Visual Mobile Designer

Even easier is implementation of graphs using the VMD classes. VMD stands for Visual Mobile Designer and marks its classes with that acronym. These classes make a `GraphPinScene` implementation available, as shown in the previous section. In addition to the scene, there are also special classes for nodes, edges, and pins that offer uniform design (see Figure 8-14). Due to this, there are no implementation details, such as creation of widgets, or setting of layout or router; just add the necessary elements. A `VMDGraphScene` already has four layers, and actions such as zoom, pan, and select. The simple example with three nodes shown in Listing 8-10 is created using a `VMDGraphScene`.

Listing 8-10. *Creating a graph using the VMD classes*

```
private VMDTopComponent() {
   VMDGraphScene scene = new VMDGraphScene();
   scenePane.setViewportView(scene.createView());
   VMDNodeWidget node1 = (VMDNodeWidget)scene.addNode("Node 1");
   node1.setNodeName("Node 1");
   VMDNodeWidget node2 = (VMDNodeWidget)scene.addNode("Node 2");
   node2.setNodeName("Node 2");
   VMDNodeWidget node3 = (VMDNodeWidget)scene.addNode("Node 3");
   node3.setNodeName("Node 3");
   VMDPinWidget p1 =(VMDPinWidget)scene.addPin("Node 1", "Pin 1");
   p1.setPinName("Pin 1");
   VMDPinWidget p2 =(VMDPinWidget)scene.addPin("Node 2", "Pin 2");
   p2.setPinName("Pin 2");
   VMDPinWidget p3 =(VMDPinWidget)scene.addPin("Node 2", "Pin 3");
   p3.setPinName("Pin 3");
   VMDPinWidget p4 =(VMDPinWidget)scene.addPin("Node 3", "Pin 4");
   pin4.setPinName("Pin 4");
   scene.addEdge("Edge 1");
   scene.setEdgeSource("Edge 1", "Pin 1");
   scene.setEdgeTarget("Edge 1", "Pin 2");
   scene.addEdge("Edge 2");
   scene.setEdgeSource("Edge 2", "Pin 3");
   scene.setEdgeTarget("Edge 2", "Pin 4");
}
```

The `VMDGraphScene` uses the type `String` for the data models of nodes, edges, and pins. As you know from the previous section, elements are added with the methods `addNode()`, `addPin()`, and `addEdge()`. Here in the example, we give the nodes and pins a name. With `setProperties()` and other methods, you can set additional properties, such as icons, for nodes or pins.

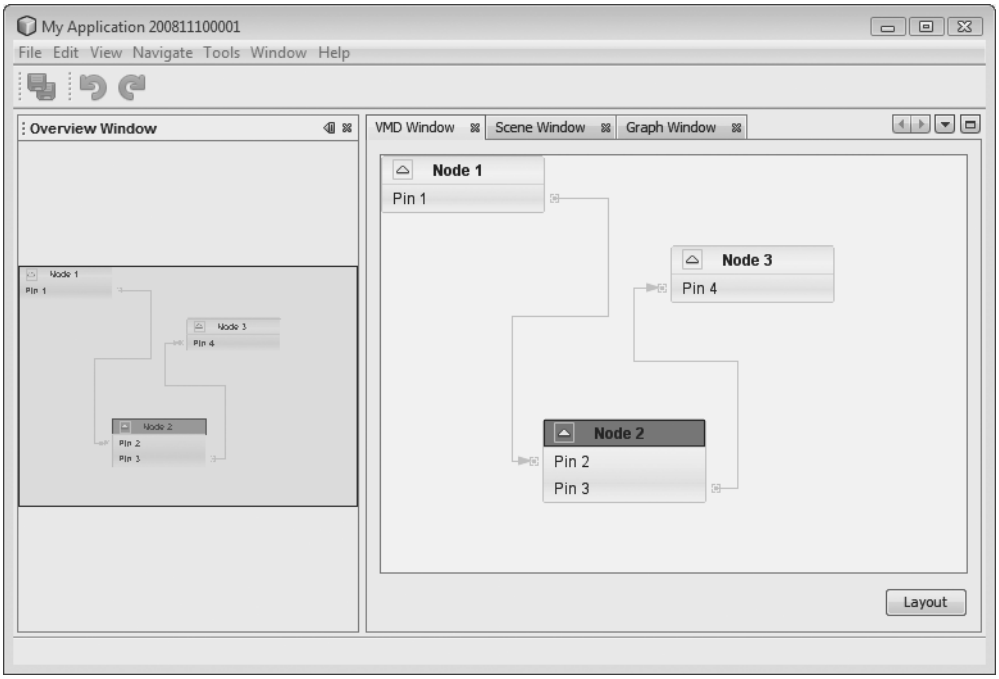


Figure 8-14. *The VMD graph classes offer additional features, suchs as hiding pins or adding icons.*

Summary

In this chapter, we looked at NetBeans Platform APIs for the creation of graphical components. Firstly, we explored the Dialogs API, with which you can display system dialogs, as well as those you create yourself.

Next, we looked at the Wizards API. You can use this comprehensive framework to create your own graphic sequences that the user steps through to create artifacts or set properties in the application.

Also, we looked at how to create multiple views within one `TopComponent`, using the Multi-Views API. Finally, we dealt with the NetBeans Platform's powerful widget library, known as the Visual Library API. We looked at the API classes, focusing on the different widgets, layouts, and actions supported out of the box, as well as how to extend them.

CHAPTER 9

Reusable NetBeans Platform Components

Let's See What We Get for Free!

In this chapter, we introduce out-of-the-box NetBeans Platform components. These can be integrated directly into your application, as in the case of the Output window and the Navigator. You'll learn the purpose of these components, as well as how they can best be customized and extended.

Help System

The NetBeans help system is based on the standard JavaHelp API. The NetBeans Platform provides a module containing the JavaHelp library and exposes a class allowing access to it. To use the help system, set a dependency in your module on the `JavaHelp Integration` module. The dependency is defined automatically when using the wizard to create a starting point for the JavaHelp system, as described in the next section.

Then, when running the module, choose Help ➤ Help Contents, which will open the Help window. There, you'll see help topics from all modules in the application, integrated and displayed as one single helpset.

Creating and Integrating a Helpset

The IDE provides a wizard to set up new helpsets. It makes an otherwise tricky process child's play. Choose File ➤ New File, select the Module Development category, and then select JavaHelp Help Set. Click Next. On the last page of the wizard, you'll see a list of files that will be created. Click Finish to create those files, which constitute your new helpset.

The basic helpset is added to the module, together with entries that register them in the layer file. That is done through the `Services/JavaHelp` extension point, in the following way:

```
<folder name="Services">
   <folder name="JavaHelp">
      <file name="module-helpset.xml" url="module-helpset.xml"/>
   </folder>
</folder>
```

In the layer file, the wizard registered the `module-helpset.xml` file, referencing all other files in the helpset. The `module-helpset.xml` file is located in the same package as the layer file. Help topics are contained in a folder separate from the Java source files. The `nbdocs` protocol is used to access the `module-hs.xml` file, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE helpsetref PUBLIC
   "-//NetBeans//DTD JavaHelp Help Set Reference 1.0//EN"
   "http://www.netbeans.org/dtds/helpsetref-1_0.dtd">
<helpsetref url="nbdocs:/com/galileo/netbeans/module/docs/module-hs.xml"/>
```

The helpset consists of the following configuration files, all of which are created by the wizard.

module-hs.xml

Other configuration files making up the helpset are registered in this file. Use the title element to assign the helpset a unique name. The `maps` element refers to map files that register help topics, defining their unique map IDs, used to reference help topics in the files defining tables of contents and indexes. The `view` element defines the helpset search engine, tables of contents, and indexes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE helpset PUBLIC
   "-//Sun Microsys Inc.//DTD JavaHelp HelpSet Version 2.0//EN"
   "http://java.sun.com/products/javahelp/helpset_2_0.dtd">
<helpset version="2.0">
   <title>My Module Help</title>
   <maps>
      <homeID>com.galileo.netbeans.module.about</homeID>
      <mapref location="module-map.xml"/>
   </maps>
   <view mergetype="javax.help.AppendMerge">
      <name>TOC</name>
      <label>Table of Contents</label>
      <type>javax.help.TOCView</type>
      <data>module-toc.xml</data>
   </view>
   <view mergetype="javax.help.AppendMerge">
      <name>Index</name>
      <label>Index</label>
      <type>javax.help.IndexView</type>
      <data>module-idx.xml</data>
   </view>
   <view>
      <name>Search</name>
      <label>Search</label>
      <type>javax.help.SearchView</type>
      <data engine="com.sun.java.help.search.DefaultSearchEngine">
              JavaHelpSearch</data>
   </view>
</helpset>
```

module-map.xml

The map file centralizes registration of help topics within the module. Using its `target` attribute, you can register HMTL files as help topics and assign them to a unique map ID. Later, refer to the help topics via their map ID when defining files that create tables of contents and indexes. Map IDs are used by the `HelpCtx` object to call up context-sensitive help, as described later in this section.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC
   "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 2.0//EN"
   "http://java.sun.com/products/javahelp/map_2_0.dtd">
<map version="2.0">
   <mapID target="com.galileo.netbeans.module.about" url="module-about.html"/>
</map>
```

module-toc.xml

Table of contents files connect map IDs to help topics displayed as helpset tables of contents. Help topics are grouped into folders containing related topics by nesting elements within other elements, as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE toc PUBLIC
   "-//Sun Microsystems Inc.//DTD JavaHelp TOC Version 2.0//EN"
   "http://java.sun.com/products/javahelp/toc_2_0.dtd">
<toc version="2.0">
   <tocitem text="My Module">
      <tocitem text="About My Module" target="com.galileo.netbeans.module.about"/>
   </tocitem>
</toc>
```

module-idx.xml

In the index file, use the element `indexitem` to register the map IDs of the help topics displayed on the Index tab of the Help window:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE index PUBLIC
   "-//Sun Microsystems Inc.//DTD JavaHelp Index Version 2.0//EN"
   "http://java.sun.com/products/javahelp/index_2_0.dtd">
<index version="2.0">
   <indexitem text="About My Module" target="com.galileo.netbeans.module.about"/>
</index>
```

Help Topics

Together, the previously described XML files configure help topics in the application. In the New File dialog, the JavaHelp Help Set wizard combines them in a very simple procedure. After completing the wizard, simply create HTML files and register them as help topics (as described in the previous sections) so they open in the Help window when accessed by the user.

Adding Links to Help Topics

Within a help topic, you can link to external sites or other help topics, even though they are found in other modules.

Links to External Sites

Typically, you will display external sites in an external browser, since the Help window is inadequate for that purpose. The NetBeans class that helps you in this regard is `BrowserDisplayer`. Use the `object` tag in a help topic to reference it:

```
<object classid="java:org.netbeans.modules.javahelp.BrowserDisplayer">
   <param name="content" value="http://www.netbeans.org">
   <param name="text" value="http://www.netbeans.org">
   <param name="textFontSize" value="medium">
   <param name="textColor" value="blue">
</object >
```

The `BrowserDisplayer` class passes the link to the `URLDisplayer` service. The default implementation of this service is made available, opening the link in the internal Swing HTML browser. Alternatively, you might want the external browser to open instead. Enabling this, we create a class named `ExternalURLDisplayer`, extending `HtmlBrowser.URLDisplayer`. This interface is located in the `UI Utilities` module, on which you set a dependency. This interface includes the `showURL()` method, providing the link to open a URL. Via this link, pass in a `URI` to the `browse()` method of the Java `Desktop` class, opening the appropriate page in the external browser.

```
public class ExternalURLDisplayer extends HtmlBrowser.URLDisplayer{
   public void showURL(URL link) {
      try {
         Desktop.getDesktop().browse(link.toURI());
      } catch(Exception ex) {
         Logger.getLogger("global").log(Level.SEVERE, null, ex);
         // show the user a message dialog
      }
   }
}
```

Next, register this service provider in the module `META-INF/services` folder, in a file called `org.openide.awt.HtmlBrowser$URLDisplayer`:

```
com.galileo.netbeans.module.ExternalURLDisplayer
#position=0
```

Links to Other Help Topics

Links to other help topics are created by simply inserting `href` tags or the `nbdocs` protocol into your help topics:

```
<a href="nbdocs://org.netbeans.modules.usersguide/org/netbeans/
   modules/usersguide/configure/configure_options.html">
   Using the options dialog</a>
```

When using the `nbdocs` protocol, it is important that you use the code name base of the module to which you will link. In the preceding example, the code name base is `org.netbeans.modules.usersguide`. After the code name base, specify the path to the applicable help topic. The help system provides an appropriate message if the specified module is not available.

Context-Sensitive Help

Context-sensitive help enables the user to directly access the help topic relating to the current context of the application. Rather than having users search for a particular topic, it is immediately available.

Create context-sensitive help topics by connecting a specific component in your application to a specific help ID in a map file. For a component to support context-sensitive help, implement the `HelpCtx.Provider` interface and use its `getHelpCtx()` method to provide an ID.

Many commonly used classes in the NetBeans APIs implement the `HelpCtx.Provider` interface, which makes the `getHelpCtx()` method available. Examples of these classes include `Node`, `DataObject`, `TopComponent`, `SystemAction`, `WizardDescriptor.Panel`, and `DialogDescriptor`. In its subclasses, you need only override the `getHelpCtx()` method, providing the map ID of the topic to be displayed.

Typically, context-sensitive help topics are made available by means of the F1 key. However, in a dialog or a wizard, overriding the `getHelpCtx()` method provides a button the user clicks to show related help topics.

Pressing the F1 key provides the help topic, thanks to this shortcut registration in the layer file:

```
<folder name="Shortcuts">
   <file name="F1.shadow">
      <attr name="originalFile" stringvalue="
              Actions/Help/org-netbeans-modules-javahelp-HelpAction.instance"/>
   </file>
</folder>
```

Now the user presses the F1 key, which runs the `HelpAction` that automatically searches the activated component. The ID of the help topic is identified via the `getHelpCtx()` method. In addition, a `JComponent` subclass can be used, and then the `setHelpIDString()` method can be used to define the map ID:

```
JComponent c = ...
HelpCtx.setHelpIDString(c, "com.galileo.netbeans.module.about");
```

Note that your component must be in focus; otherwise, the help topic will not be found. By default, the `TopComponent` is not focusable, to which end you use the `isFocusable()` method. Make the window focusable simply by calling `setFocusable()`:

```
final class MyTopComponent extends TopComponent {
   private MyTopComponent() {
      setFocusable(true);
   }
```

```
    public HelpCtx getHelpCtx() {
        return new HelpCtx("com.galileo.netbeans.module.about");
    }
}
```

Now `MyTopComponent` is activated. The user can press the F1 key and the help topic matching the map ID `com.galileo.netbeans.module.about` will be called. You define a map ID for each help topic in the map file as shown in the "Creating and Integrating a Helpset" section earlier in the chapter. To show the Help window without a specific topic being displayed within it, return `HelpCtx.DEFAULT_HELP`. The `HelpCtx` determines the ID of the help topic by using the fully qualified name of the class. In the preceding example, if we were to use `new HelpCtx(getClass())`, the help ID would be unique, returning `com.galileo.netbeans.module.MyTopComponent`.

Opening the Help System

To call the help system programmatically, access it with Lookup (see Listing 9-1). There is a registered implementation of the `Help` class.

Listing 9-1. *Calling a specific help topic*

```
Help h = Lookup.getDefault().lookup(Help.Class);
if(h != null)
    h.showHelp(new HelpCtx("com.galileo.netbeans.Module.about"));
    // h.showHelp(HelpCtx.DEFAULT_HELP);
```

We pass a `HelpCtx` instance representing a help topic to the method `showHelp()`. The constructor receives the ID of the requested help topic, which was registered in the map file. Instead, to show the default help topic, pass `HelpCtx.DEFAULT_HELP` to the constructor.

Output Window

The NetBeans Platform provides the Output window as a display area for showing messages to the user (see Figure 9-1). Typically, messages come from tasks processed by your application. Messages from multiple tasks display in different tabs simultaneously.

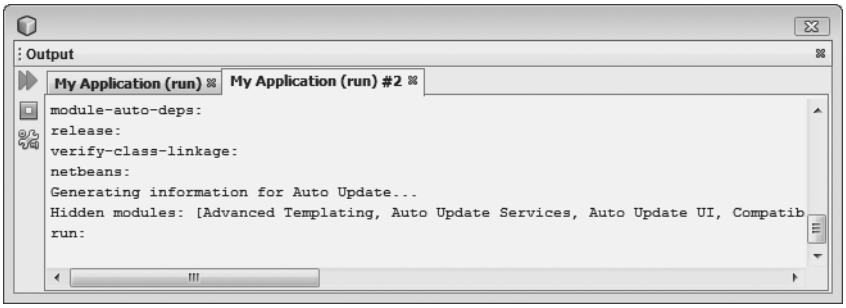


Figure 9-1. *Output window*

To use this module in applications, go to the Project Properties dialog of the application and activate the `Output Window` module and the `I/O APIs` module, both within the `platform` cluster in the Libraries panel. Once modules are activated within the application, dependencies on them can be set in the module.

Next, specify a dependency on I/O APIs in the module. In the Project Properties dialog of the module, set I/O APIs as a dependency.

As a result of the preceding settings, the module writes to the Output window as shown in Listing 9-2.

Listing 9-2. *Using the Output window*

```
InputOutput io = IOProvider.getDefault().getIO("Task", true);
io.getOut().println("Info message");
io.getErr().println("error message");
io.getOut().close();
io.getErr().close();
```

In the preceding snippet, we use `IOProvider.getDefault()`, using Lookup to access the related service provider located in the `Output Window` module. If the `Output Window` module (and thus the service provider) is not available, the standard output is used instead. The Output window is implemented as a global service, about which further information is found in Chapter 6.

The `getIO()` method provides the `InputOutput` through which the window is accessed. Define the name appearing in the tab or title bar. The Boolean parameter determines whether a new tab is created or an already existing one is used. Using the methods `getOut()` and `getErr()`, obtain an `OutputWriter`, which is a subclass of the Java `PrintWriter` class. Use `println()` for printing messages, as is normally done.

The text of the `getErr()` output is displayed in red in the Output window. It is important to end the stream with the `close()` method again. Doing so also sets the bold text of tabs back to normal font and signals the user that the task is complete.

If multiple tabs are displayed, use the `InputOutput.select()` method to ensure the appropriate tab is active. Open the Output window by means of the Window ➤ Output menu item. This menu item is added by the `Output Window` module.

The Output window has its own toolbar, within which you integrate actions. In Figure 9-1, you can see two actions for stopping and resuming the current process. To that end, there is a variant on the `getIO()` method, anticipating an array of actions as its second parameter. You can pass in very simple action implementations. However, it is important that your action uses the `SMALL_ICON` property to provide an icon to be added to the Output window toolbar.

In the following snippet, an example of such an action is illustrated. It derives from the `AbstractAction` class, which implements the `Action` interface. In the constructor, create an `ImageIcon` and assign it to the `SMALL_ICON` property:

```
public class StopTask extends AbstractAction {
   public StopTask() {
      putValue(SMALL_ICON,
             new ImageIcon(ImageUtilities.loadImage("icon.gif", true)));
   }
   public void actionPerformed(ActionEvent evt) {
      // stop the task
```

```
    }
}
```

Finally, pass a second parameter to the `getIO()` method. The second parameter is an array of actions, in this case consisting of an instance of the `StopTask` class:

```
InputOutput io =
        IOProvider.getDefault().getIO("Task", new Action[]{new StopTask()});
```

Navigator

Context-sensitive panels for navigating documents shown in the editor are defined by using the Navigator and its API. Constructors, methods, and other elements of an opened Java source file are shown by default in the Navigator (see Figure 9-2), allowing the user to double-click these items so that the cursor jumps to the relevant point in the document. However, this is only one of the Navigator's many uses. In fact, the Navigator can be used to direct the user through any kind of document.

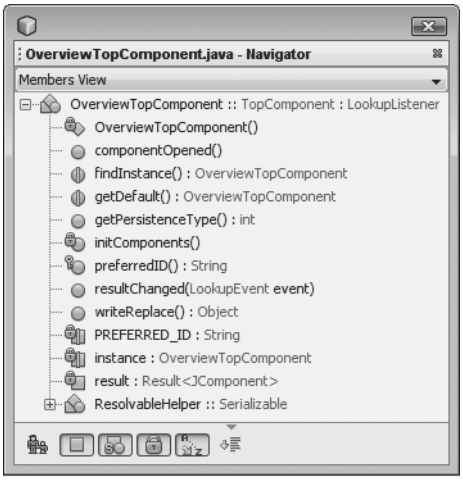


Figure 9-2. *Navigator panel for a single Java source file*

The Navigator API provides the `NavigatorPanel` interface, defining new panels in the Navigator. Panels are added declaratively, using the folder of a specific MIME type in the module layer file. When a file of a particular MIME type is opened, Navigator panels registered under that MIME type are displayed.

However, providing a MIME type is not required. In some cases, a `FileObject`, `DataObject`, or `Node` corresponding to a Navigator panel may not be available. In those cases, use the `NavigatorLookupHint` interface, adding it to the related component Lookup. The `NavigatorLookupHint` interface specifies a single method, used to return a MIME type. In this way, a Navigator panel can be associated with a component, even if the component has no corresponding MIME type.

As an example, take the code from the "Intermodule Communication" section in Chapter 6 and extend it with another module. The example in Chapter 6 searches a list and shows `Mp3FileObject`s as its entries. However, the selected element is made available via the local Lookup. This element, making music titles available, is used in a new module, where we make all albums available in a Navigator panel, as shown in Figure 9-3 and in the sections that follow.

The example also shows how easy it is to extend an application on the NetBeans Platform by adding new modules and components that provide additional features.

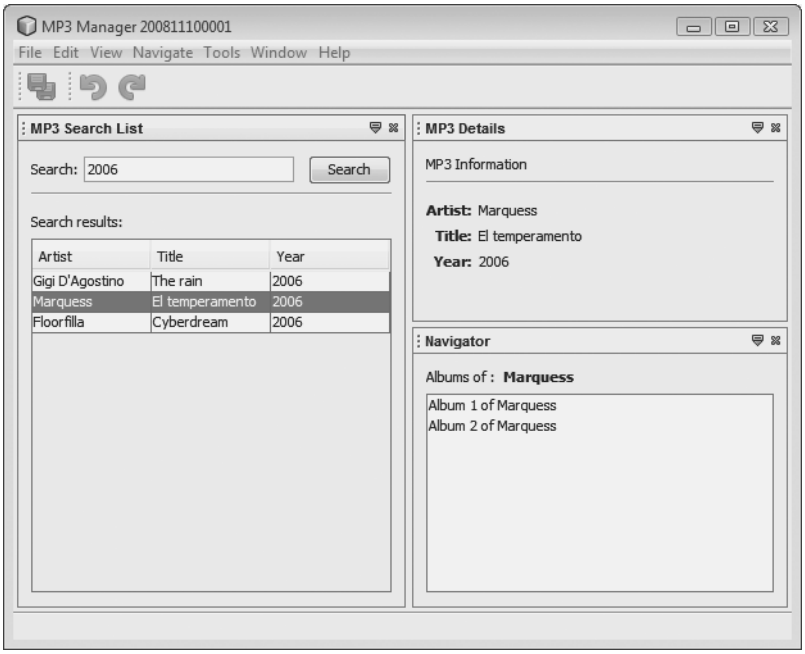


Figure 9-3. *Context-sensitive Navigator panel*

The Navigator API belongs not to the standard set of modules in the NetBeans Platform, but to those belonging to the NetBeans IDE. Therefore, specify using that API by going to the Project Properties dialog and choosing the API from the `ide` cluster. First, activate the cluster, and then select the specific module where the API is found.

Next, add a new module to the suite, with the name `MP3 Navigator`. Dependencies required by this module are the Navigator API and the Utilities API. Before beginning to code, use the New File dialog to create a new JPanel Form. Change the class created by the wizard so that it extends `JComponent` instead of `JPanel`. Next, implement the interfaces `NavigatorPanel` and `LookupListener`.

The contents of the `JPanel` are defined with the Matisse GUI Builder. For example, to understand the code shown in Listing 9-3, you need know that two `JLabel`s and a `JList` have been added.

Listing 9-3. *Navigator panel implementation*

```
public class Mp3AlbumNavigatorPanel extends JComponent
   implements NavigatorPanel, LookupListener {
   private Lookup.Result<Mp3FileObject> result = null;
   public Mp3AlbumNavigatorPanel() {
      initComponents();
   }
   public JComponent getComponent() {
      return this;
   }
   public void panelActivated(Lookup context) {
      result = Utilities.ActionsGlobalContext().lookupResult(Mp3FileObject.class);
      result.addLookupListener(this);
   }
   public void panelDeactivated() {
      result.removeLookupListener(this);
      result = null;
   }
   public void resultChanged(LookupEvent event) {
      Collection<? extends Mp3FileObject> mp3s = result.allInstances();
      if(!mp3s.isEmpty()) {
         Mp3FileObject  mp3 = mp3s.iterator().next();
         // search for albums of selected artist and display it
         albumsOf.setText(mp3.getArtist());
         DefaultListModel model = new DefaultListModel();
         model.addElement(new String("Album 1 of " + mp3.getArtist()));
         model.addElement(new String("Album 2 of " + mp3.getArtist()));
         albums.setModel(model);
      }
   }
}
```

Use the `getComponent()` method specified by the `NavigatorPanel` interface to return the Navigator panel. The `panelActivated()` and `panelDeactivated()` methods are called if the panel is selected or deselected. With activation of the panel, we receive a `Lookup.Result` for the `Mp3FileObject` via the global Lookup. Next, register a `LookupListener` to be able to react when new entries need be added to the list.

When the Lookup changes, the `resultChanged()` method is called, which adds new content to the panel. To simplify things, our code will simply add two new entries. In real life, however, you'd typically search a database and display the content you find there.

To enable the Navigator to find and integrate the panel, register the panel in your layer file. This is done in the `Navigator/Panels` folder. Within this folder, assign the panel to a MIME type relevant to the entries displayed in the panel (see Listing 9-4). In this case, use the `audio/mpeg` MIME type, although any MIME type could be used.

Listing 9-4. *Registration of the Navigator panel*

```
<folder name="Navigator">
   <folder name="Panels">
      <folder name="audio">
```

```
        <folder name="mpeg">
           <file name="com-galileo-netbeans-module-mp3navigator-
                       Mp3AlbumNavigatorPanel.instance"/>
        </folder>
      </folder>
   </folder>
</folder>
```

You might ask how the Navigator knows when to show our panel. The Navigator normally shows entries that correspond to the content of the currently selected node. In cases such as ours, in which we are not dealing with nodes, use the `NavigatorLookupHint` instead. The `NavigatorLookupHint` interface provides the `getContentType()` method, with which the component (in our case the `Mp3SearchList`) provides the MIME type for which a panel should be shown. Implement this interface in the `Mp3SearchList` class (see Listing 9-5) and return the `audio/mpeg` MIME type, which is how our panel was registered in the layer file.

Listing 9-5. *Implementation of the NavigatorLookupHint interface*

```
final class Mp3SearchList extends TopComponent implements ListSelectionListener {
   private Mp3SearchList() {
      ...
      associateLookup(new ProxyLookup(
         new AbstractLookup(content),
         Lookups.singleton(new Mp3AlbumNavigatorLookupHint())));
   }
   private static final Class Mp3AlbumNavigatorLookupHint
           implements NavigatorLookupHint {
      public String getContentType() {
         return "audio/mpeg";
      }
   }
}
```

Provide the inner class `Mp3AlbumNavigatorLookupHint`, implementing the `NavigatorLookupHint` interface. Add an instance of this class to your local Lookup. Because we defined an `AbstractLookup` as a local Lookup, which contains the selected entry from the search list, we cannot display this instance directly. Provide a `ProxyLookup` to which we pass the `AbstractLookup` and a Lookup providing the `Lookups` factory. Define this `ProxyLookup` with the `associateLookup()` method as a local Lookup. As soon as the `Mp3SearchList` receives focus, the Navigator is informed about the available `NavigatorLookupHint` in the global Lookup. As a result, the Navigator calls the method `getContentType()` and, with the help of the return value, shows the required panel.

The Navigator becomes even more interesting when multiple components are available within it. Many panels can be created as described previously, and the related MIME types can be registered in the layer file. The Navigator switches between panels automatically, depending on which component is currently active.

Finally, be aware of the `NavigatorHandler` class. The `NavigatorHandler` class contains the `activatePanel()` method, to which you can pass a panel that you want opened programmatically.

Properties Window

The Properties window is a component for displaying and editing *node properties*. Node properties represent data with which the user interacts (see the "Nodes API" section in Chapter 7), while the Properties window allows the user to change those properties.

A group of related properties can be managed by the NetBeans `Sheet` class, belonging to the Nodes API. The `AbstractNode` class, which is typically the superclass of your node, provides a sheet of properties via its `getSheet()` method. Simply override the `createSheet()` method, adding the node's specific properties to the sheet.

Implementing a Properties window is shown by example. In the example, we show properties of the currently selected MP3 file in the Properties window. First, create a new file type for MP3 files, together with a related `Node` class, as described in Chapter 7. The end result should be as shown in Figure 9-4.

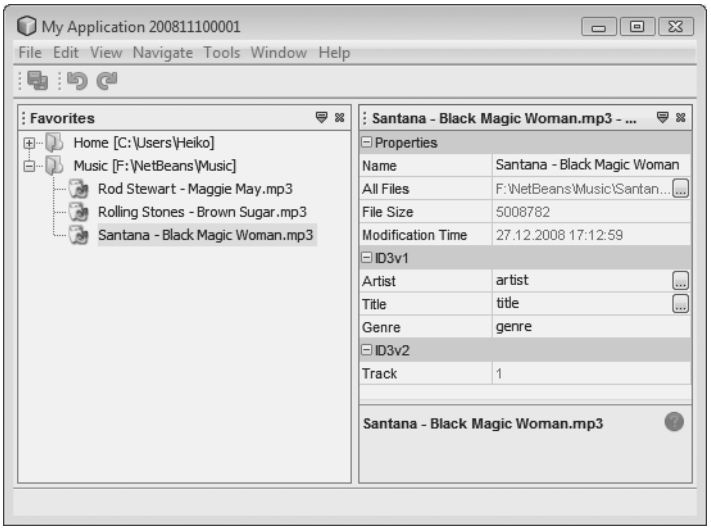


Figure 9-4. *Properties of the selcted node shown in the Properties window*

Providing Properties

Override the `createSheet()` method in the `Node` class (in this example the `Mp3DataNode` class) representing files of the MP3 type. First, create a `Sheet` instance via the call to the superclass, providing a set of default properties to the instance. You can see these default properties in the first section of the Properties window in Figure 9-4.

If you do not want this default set of properties, provide your own `Sheet` instance to the constructor. From Figure 9-4, you can see that the properties can be divided across different areas, and expanded or collapsed by the user. Properties of each area are managed by the `Sheet.Set` class (see Listing 9-6).

To create the areas for `ID3v1` and `ID3v2`, use the `createPropertiesSet()` factory method to create two `Sheet.Sets`. Provide a unique name for the `Sheet.Set`, using the method `setName()` for use within the internals of the Properties window module. If you fail to name your

`Sheet.Set`, only the most recently created `Sheet.Set` is shown. With the `setDisplayName()` method, specify the name of the heading for the set you create.

Listing 9-6. *Use createSheet() to provide a sheet of properties shown in the Properties window.*

```
public class Mp3DataNode extends DataNode {
   protected Sheet createSheet() {
      Sheet s = super.createSheet();
      Sheet.Set id3v1 = Sheet.createPropertiesSet();
      Sheet.Set id3v2 = Sheet.createPropertiesSet();
      id3v1.setName("ID3v1");
      id3v1.setDisplayName("ID3v1");
      id3v2.setName("ID3v2");
      id3v2.setDisplayName("ID3v2");
      Mp3DataObject mp3 = getLookup().lookup(Mp3DataObject.class);
      try {
         PropertySupport.Reflection<String> artistProp =
            new PropertySupport.Reflection<String>(mp3, String.class, "artist");
         PropertySupport.Reflection<String> titleProp  =
            new PropertySupport.Reflection<String>(mp3, String.class, "title");
         PropertySupport.Reflection<String> genreProp  =
            new PropertySupport.Reflection<String>(mp3, String.class, "genre");
         Property trackProp = new PropertySupport.Reflection<Integer> (
                mp3, Integer.class, "getTrack", null);
         artistProp.setName("Artist");
         titleProp.setName("Title");
         genreProp.setName("Genre");
         trackProp.setName("Track");
         id3v1.put(artistProp);
         id3v1.put(titleProp);
         id3v1.put(genreProp);
         id3v2.put(trackProp);
      } catch (NoSuchMethodException ex) {
         ex.printStackTrace();
      }
      s.put(id3v1);
      s.put(id3v2);
      return s;
   }
}
```

We use Lookup to access a `DataObject` representing the MP3 file that made its properties available. For each property, create an object. In addition, make a distinction between properties that can be changed and those that can only be displayed. For properties the user is able to change, we provide a `PropertySupport.Reflection` instance with the corresponding type—in this case a `String`.

As parameters, pass the `DataObject`, the property data type, and the name of the getter/setter combination. For example, for the first property, we can pass in the artist, which means that we need to create an `Mp3DataObject`, as well as the `setArtist()` and `getArtist()` methods. Otherwise, a `NoSuchMethodException` will be thrown.

Then create a different constructor for properties the user should not be able to change. In this case, pass in the getter and setter separately. Since we do not want the user to be able to change the property, pass in `null` for the setter. To illustrate these points, part of the `Mp3DataObject` class is shown in Listing 9-7.

Listing 9-7. *A DataObject supporting properties must include getters and setters. If the property should not be changed, a setter is not provided.*

```
public class Mp3DataObject extends MultiDataObject  {
   public String getArtist() {
      return this.artist;
   }
   public void setArtist(String artist) {
      this.artist = artist;
   }
   ...
   public int getTrack() {
      return this.track;
   }
}
```

Give created instances representing individual properties a name via the `setName()` method, and add them to the `Sheet.Set` via the `put()` method. Finally, use another `put()` to add the `Sheet.Set` to the `Sheet`, which is returned at the end of the overridden `createSheet()` method.

User-Defined Properties Editor

A Swing component can be provided as an editor for a property in the Properties window. Doing so, you can support the user by (for example) restricting the available list of values defined for a particular property. In Figure 9-4, you saw the value for the `genre` property set via a `JComboBox`. To provide an editor of this kind, provide the following statement for each property with the `GenrePropertyEditor`, providing a Swing component such as a `JComboBox`:

```
genreProp.setPropertyEditorClass(GenrePropertyEditor.class);
```

Now observe an editor of this kind being created. Focus only on the most important classes and methods.

Start with the `GenrePropertyEditor` class, which is extended using the standard JDK class `PropertyEditorSupport`, a base implementation that must be implemented by all user-defined editors. In addition, implement `ExPropertyEditor` and `InplaceEditor.Factory` (see Listing 9-8) Obtain a `PropertyEnv` object via the `attachEnv()` method, which belongs to `ExPropertyEditor`, providing access to the Properties window.

Use the `attachEnv()` method to register an `InplaceEditor.Factory` instance, which is our class, responsible for the creation of the editor. The `getInplaceEditor()` method retrieves the editor. Next, provide implementation of the graphic editor's component as a private inner class, derived from `InplaceEditor`.

To use a `JComboBox` as the editor, create it as a private member of the class and initialize it with desired values. Then use `getComponent()` to return the `JComboBox` from the editor. Also important in the `InplaceEditor` are the `setValue()` and `getValue()` methods, which define and

provide the values of the `JComboBox`, together with the `reset()` method, which returns a changed entry to its original value, typically via the Esc key.

Listing 9-8. *The user-defined editor for selecting the genre*

```
public class GenrePropertyEditor extends PropertyEditorSupport
        implements ExPropertyEditor, InplaceEditor.Factory {
   private InplaceEditor ed = null;
   public void attachEnv(PropertyEnv propertyEnv) {
      propertyEnv.registerInplaceEditorFactory(this);
   }
   public InplaceEditor getInplaceEditor() {
      if(ed == null)
         ed = new Inplace();
      return ed;
   }
   private static Class Inplace implements InplaceEditor {
      private PropertyEditor editor = null;
      private PropertyModel  model  = null;
      private JComboBox genres = new JComboBox(
         new String[] {"Techno", "Trance", "Rock", "Pop"});

      public JComponent getComponent() {
         return this.genres;
      }
      public Object getValue() {
         return this.genres.getSelectedItem();
      }
      public void setValue(Object object ) {
         this.genres.setSelectedItem(object);
      }
      public void reset() {
         String genre = (String) editor.getValue();
         if(genre != null)
            this.genres.setSelectedItem(genre);
      }
   }
}
```

Options Window

With the Options Dialog API and SPI, you easily provide extensions to the NetBeans Platform Options window. Using the Options window, the user can easily and comfortably customize application settings. In addition, the module providing this functionality also provides basic scaffolding of the dialog into which your panels can be integrated declaratively via the layer file.

Two types of integration panels are supported: *primary panels* and *secondary panels*. A primary panel is a (main) category within the Options dialog, as, for example, the General or Keymap categories (see Figure 9-5). A tab (subcategory) within a primary panel is a secondary panel.

Storing and loading of settings is done via the Preferences API. A specific implementation of this API is provided by the NetBeans Platform, as will be discussed in the "Settings Administration" section.

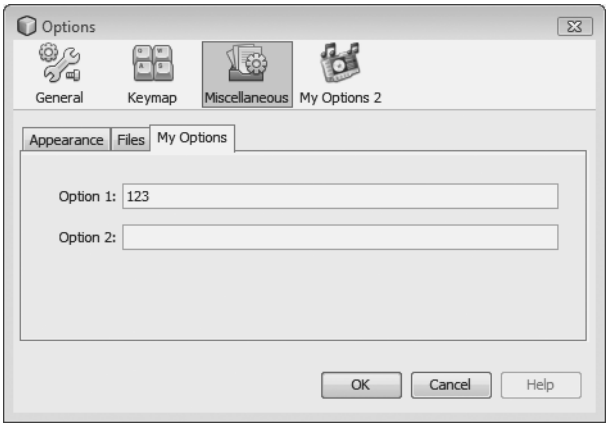


Figure 9-5. *Standard Options window, within which you integrate your panels.*

Providing an Options Panel

To create either kind of Options panel, the IDE provides a wizard. You find the wizard under File ➤ New File ➤ Module Development ➤ Options Panel. Depending on the type of panel being integrated, choose either the secondary panel or primary panel option.

In case of a secondary panel, you need to choose a primary panel, specify a title and a tooltip for this panel, and define the keywords for the Quick Search. If you choose the primary panel option, you need to specify the title, a category label, an icon, and the keywords for the Quick Search. Click Next to specify a class name prefix for the classes to be created by the wizard, and then click Finish.

View and Controller

An Options panel consists of a *view* and a *controller*. The view provides the GUI, as well as the loading and storing of data. The controller generates the view and mediates between the Options window and the view. The panel that defines the view is derived from the `JPanel` class. On this panel, you place arbitrary Swing components, which are used to show options in the panel and allow the user to select them.

For example, in Listing 9-9, two text fields are used. The values of these fields are stored and loaded via the Preferences API (see the "Settings Administration" section) using the methods `store()` and `load()`. These are called by the controller while opening and closing the panel.

The Options window can be closed by the user when appropriate settings have been selected. At that point, the panel must inform the Options window about its state, either `valid` or `invalid`. Implement validation via the `valid()` method. In this example, we want to make sure the user enters a value in the first field. Accordingly, we add a `DocumentListener` to the text field. Whenever the user makes an entry in the first text field, the controller is informed via the

`Controller.changed()`, which then calls the `valid()` method, only returning `true` when at least one character is entered into the text field.

Listing 9-9. *View of the Options panel, deriving from JPanel*

```
public final class ModuleOptions1Panel extends JPanel
        implements DocumentListener {
   private JTextField option1;
   private JTextField option2;
   private final ModuleOptions1PanelController controller;
   public ModuleOptions1Panel(ModuleOptions1PanelController  ctrl) {
      this.controller = ctrl;
      initComponents();
      option1.getDocument().addDocumentListener(this);
   }
   public void insertUpdate(DocumentEvent event) {
      controller.changed();
   }
   public void removeUpdate(DocumentEvent event) {
      controller.changed();
   }
   public void changedUpdate(DocumentEvent event) {
      controller.changed();
   }
   public void load() {
      option1.setText(NbPreferences.forModule(ModuleOptions1Panel.class).
              get("option1", "default"));
   }
   public void store() {
      NbPreferences.forModule(ModuleOptions1Panel.class).
              put("option1", option1.getText());
   }
   public boolean valid() {
      if(option1.getText().length() == 0) {
         return false;
      } else {
         return true;
      }
   }
}
```

Now look at the responsibilities of the controller. Since the controller needs to interact with the Options window, its interfaces are defined by the abstract class `OptionsPanelController`.

The controller's most important task is creation of the view, using `getPanel()`, which we provide via the method `getComponent()`. As you can see in Listing 9-10, `getComponent()` receives a Lookup. This is a proxy Lookup, containing the Lookups of all controllers available in the Options window. The controller uses the `getLookup()` method to make a Lookup available, which is already implemented by the abstract class `OptionsPanelController`. This default implementation provides an empty Lookup. To put objects into the Lookup, override the `getLookup()` method. This Lookup is received via the `getComponent()` method, which is used to communicate with other Options panels. See Chapter 6 for a discussion on Lookups.

The method `update()` is called the first time a panel is accessed. Here, we call the `load()` method on the panel, which loads data to initialize the fields. When the user clicks the OK button, the `applyChanges()` method in the Options window is called. Here, the data is saved via the `save()` method. If the user closes the Options window, we use the `cancel()` method to handle this scenario, but we obviously do not save the data. Users can also roll back the changes that have been made.

Using the `isValid()` method, we inform the Options window whether data in the panels is in order. If this is not the case, the OK button is automatically deactivated. Moreover, the Options window must also be informed whether data has been changed. This is done with the `isChanged()` method.

With the `getHelpCtx()` method, a `HelpCtx` object is provided that contains a reference to a help topic displayed if the user clicks the Options window's Help button.

To inform the Options window of changes in data, provide it with code that will register changes. This is achieved via the standard JDK methods `addPropertyChangeListener()` and `removePropertyChangeListener()`.

You already know the `changed()` method from the view class `ModuleOptions1Panel`. This is called when data changes in the view, informing the Options window, which has registered itself as a listener. As a result, the Options window checks again whether data is valid.

Listing 9-10. *Options panel controller*

```
final class ModuleOptions1PanelController extends OptionsPanelController  {
   private ModuleOptions1Panel panel;
   private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
   private boolean changed;
   public JComponent getComponent(Lookup masterLookup) {
      return getPanel();
   }
   private ModuleOptions1Panel getPanel() {
      if (panel == null) {
         panel = new ModuleOptions1Panel(this);
      }
      return panel;
   }
   public void update() {
      getPanel().load();
      changed = false;
   }
   public void applyChanges() {
      getPanel().store();
      changed = false;
   }
   public void cancel() {
   }
   public boolean isValid() {
      return getPanel().valid();
   }
   public boolean isChanged() {
      return changed;
   }
```

```
    public HelpCtx getHelpCtx() {
        return null;
    }
    public void addPropertyChangeListener(PropertyChangeListener l) {
        pcs.addPropertyChangeListener(l);
    }
    public void removePropertyChangeListener(PropertyChangeListener l) {
        pcs.removePropertyChangeListener(l);
    }
    public void changed() {
        if (!changed) {
            changed = true;
            pcs.firePropertyChange(OptionsPanelController.PROP_CHANGED, false, true);
        }
        pcs.firePropertyChange(OptionsPanelController.PROP_VALID, null, null);
    }
}
```

Registration

Option panels can be integrated declaratively into the application via the layer file. The layer file entries are processed by two factory methods, responsible for Options panel controller creation, dependent on the type of the panel concerned. The wizard creates the necessary entries in the layer file. These are shown in Listing 9-11.

Listing 9-11. *Registration of a secondary Options panel*

```
<folder name="OptionsDialog">
  <folder name="Advanced">
    <file name="com-galileo-netbeans-module-ModuleOptions1AdvancedOption.instance">
      <attr name="controller" newvalue="
              com.galileo.netbeans.module.ModuleOptions1OptionsPanelController"/>
      <attr name="displayName" bundlevalue="com.galileo.netbeans.module.Bundle
              #AdvancedOption_DisplayName_ModuleOptions1"/>
      <attr name="instanceCreate" methodvalue="
              org.netbeans.spi.options.AdvancedOption.createSubCategory"/>
      <attr name="keywords" bundlevalue="com.galileo.netbeans.module.Bundle
              #AdvancedOption_Keywords_ModuleOptions1"/>
      <attr name="keywordsCategory" stringvalue="Advanced/null"/>
      <attr name="toolTip" bundlevalue="com.galileo.netbeans.module.Bundle
              #AdvancedOption_Tooltip_ModuleOptions1"/>
    </file>
  </folder>
</folder>
```

The `controller` attribute specifies the panel controller, which we looked at earlier. The `displayName` attribute is a pointer to a bundle where the name of the created panel is stored. The same applies to the `toolTip` and `keywords` attributes. The `keywordsCategory` attribute specifies the relative path to your panel inside the Options window.

Important is the `instanceCreate` attribute. It is used to specify the factory method, which is new to version 6.5 and is responsible for the controller creation. When dealing with a secondary panel, this is the `AdvancedOption.createSubCategory()` method.

The registration of a primary panel differs slightly, depending on whether the panel allows secondary panels or not (see Listing 9-12).

Listing 9-12. *Registration of a primary Options panel*

```
<folder name="OptionsDialog">
   <file name="ModuleOptions2OptionsCategory.instance">
      <attr name="categoryName" bundlevalue="com.galileo.netbeans.module.Bundle
             #OptionsCategory_Name_ModuleOptions2"/>
      <attr name="controller" newvalue="
             com.galileo.netbeans.module.ModuleOptions2OptionsPanelController"/>
      <attr name="iconBase" stringvalue="com/galileo/netbeans/module/icon.png"/>
      <attr name="instanceCreate" methodvalue="
             org.netbeans.spi.options.OptionsCategory.createCategory"/>
      <attr name="keywords" bundlevalue="com.galileo.netbeans.module.Bundle
             #OptionsCategory_Keywords_ModuleOptions2"/>
      <attr name="keywordsCategory" stringvalue="ModuleOptions2"/>
      <attr name="title" bundlevalue="com.galileo.netbeans.module.Bundle
             #OptionsCategory_Title_ModuleOptions2"/>
   </file>
</folder>
```

Additional to the secondary panel, the preceding entries include the primary panel registration and an icon definition via the `iconBase` attribute. The factory method for a primary panel is `OptionsCategory.createCategory()`. This type of registration allows no secondary panels, because of the `controller` attribute, which specifies that the controller provides an Options panel.

If you've checked the option allowing the primary panel to have secondary panels, the preceding layer entry will contain the `advancedOptionsFolder` attribute instead of the controller attribute:

```
<attr name="advancedOptionsFolder"
        stringvalue="OptionsDialog/ModuleOptions2OptionsCategory"/>
```

In this case, you can place your secondary panels under the folder `OptionsDialog/ModuleOptions2OptionsCategory`.

Open Option Panels

Using the `OptionsDisplayer` class, you can access the Options window. You can access this window directly with a particular tab opened as follows:

```
OptionsDisplayer.getDefault().open("ModuleOptions2OptionsCategory");
```

Settings Administration

Settings and configuration data within the NetBeans Platform is stored and loaded via the JDK Preferences API. With the `Preferences` class, Java saves and loads settings without users

needing to know their physical location. Various implementations are available, so that the settings can be stored in a file, a system registry, or a database. Settings are stored in hierarchical structure in the form of key/value pairs. An instance of the `Preferences` class represents one node within the hierarchy. Imagine a node as a file in a database where data can be saved.

With the `NbPreferences` class, the Utilities API provides an implementation of the Preferences API tailored to the NetBeans Platform. The `NbPreferences` class allows storage of settings in a central configuration location within the application user directory. It does this using properties files. Therefore, the NetBeans Platform implementation lets you handle preferences on a per user basis.

The `NbPreferences` class provides two methods. The `forModule()` method provides a `Preferences` node for each module in a properties file, stored in the `config/Preferences` folder within the user directory (see Figure 9-6). The `root()` method provides an application-level node that allows storing global preferences in the file `config/Preferences.properties`.

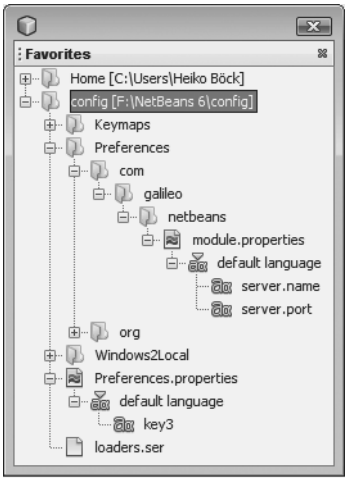


Figure 9-6. *Settings can be stored via the NetBeans Platform Preferences implementation, either for a specific module or globally.*

Using the NetBeans Platform implementation of the `Preferences` class, loading and storing user settings is easily accomplished. For example, to store the name and port of a server, you simply make the call shown in Listing 9-13.

Listing 9-13. *Loading and storing via the Preferences API*

```
Preferences node = NbPreferences.forModule(this.getClass());
String name = node.get("server.name", "localhost");
int port = node.getInt("server.port", 8080);
node.put("server.name", name);
node.putInt("server.port", port);
```

Apart from the methods shown here for data access, several others are available. For example, you can store arrays or Boolean values. Moreover, you can use the Preferences API with a `Preferences` instance (which is a node) to register a `NodeChangeListener` as well as a

`PropertyChangeListener`, notifying you when a child node is added or removed and when changes to them are made.

Palette

The `Palette` module, which is part of the NetBeans IDE, is concerned with the graphic display of components that can be dragged and dropped onto a surface in your application. Typical implementations of the module aim to give the user easy access to a range of snippets, created when a component is dropped into an editor. A good example of this is the Matisse GUI Builder in the NetBeans IDE, which places its AWT and Swing components into the palette (see Figure 9-7), from where the user moves them to the NetBeans editor. New components can also be added to the palette at runtime.

The content of the palette is handled by the `PaletteController`. The `PaletteController` is available to the `TopComponent` via its Lookup. Whenever a particular `TopComponent` is active while a `PaletteController` is found in its Lookup, the palette opens automatically, displaying its content. The `PaletteController` is created by a `PaletteFactory` class.

There are two ways to create the components displayed in the palette. You can define the components within XML files you register in the layer file; alternatively, you can create a node hierarchy and then display the hierarchy as components in your palette. We examine both approaches in this chapter.

Note that you are not limited to providing palettes to your own `TopComponent`s. Optionally, you may register a palette in the layer file under the folder of a MIME type used by a `TopComponent` in a different module. When a document of that MIME type is opened in the editor, the palette opens. This approach is of particular use to those who want to extend the NetBeans IDE. See the "Palette" section in Chapter 15 for further information on this particular topic.

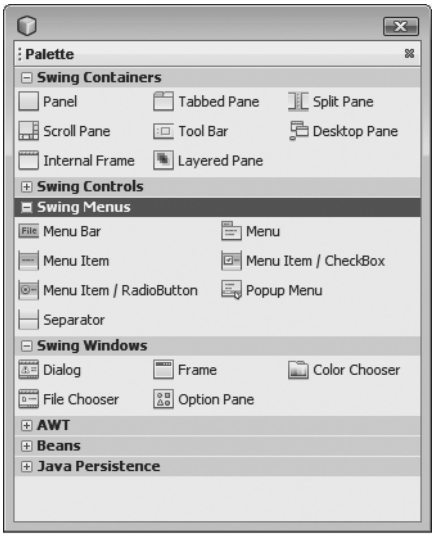


Figure 9-7. *The NetBeans GUI Builder's palette*

Defining and Adding Palette Components via the Layer File

A component placed into a palette is defined by the XML file shown in Listing 9-14.

Listing 9-14. *An XML file defining a palette component*

```
<!DOCTYPE editor_palette_item PUBLIC
   "-//NetBeans//Editor Palette Item 1.1//EN"
   "http://www.netbeans.org/dtds/editor-palette-item-1_1.dtd">
<editor_palette_item version="1.1">
   <body></body>
   <icon16 urlvalue="file:/E:/icon16.jpg"/>
   <icon32 urlvalue="file:/E:/icon32.jpg"/>
   <inline-description>
      <display-name>My Palette Item</display-name>
      <tooltip>My Palette Item</tooltip>
   </inline-description>
</editor_palette_item>
```

Looking at the preceding XML file, you can see that the elements `icon16` and `icon32` define an icon to be shown for the component in question, assuming that the user has not chosen to hide the component. Typically, the icons are 16 and 32 pixels in size. However, they can have other dimensions as well. Despite that flexibility, it makes sense to use a 16-pixel icon for the `icon16` attribute because the same icon is used to represent the component in the Palette Manager, which you can display by choosing it from the palette's context menu. Icons shown in the Palette Manager must be of that size. It is also interesting that, as you've just seen, absolute paths can be used. That implies the icons need not necessarily be found within the module. They are also defined in user-specific lists, which is useful when letting users define their own components in the palette.

With the `inline-description` element and its subelements `display-name` and `tooltip`, you set the text used for the component in the palette. As an alternative to `inline-description`, you can use the `description` element. Then, with the attribute `localising-bundle`, you provide a resource bundle that supplies the values of `display name` and `tooltip`. The attributes `display name key` and `tooltip key` provide keys for these values. Thus, you may internationalize each of your palette entries. For doing this, it is also useful to have a look at the DTD of the XML file included in the Appendix.

Using the approach just described, many components for display in the palette can be defined. When you're ready to add them to the palette, do so by registering them in the module layer file. Start by defining a new folder in the layer file, with any name you like. Within the folder, create a subfolder for each category of component you want displayed in the palette. Finally, within the category subfolder, register each of the XML files created to define the components. A structure similar to the following results from this procedure:

```
<folder name="MyPaletteItems">
   <folder name="My Category">
      <file name="myitem1.xml" url="myitem1.xml"/>
      <file name="myitem2.xml" url="myitem2.xml"/>
   </folder>
</folder>
```

Finally, we need the `PaletteController` added to the `TopComponent` for which we are creating the palette. As stated in the introduction of this section, a `PaletteFactory` class is required for this purpose. The `PaletteFactory` class makes a `createPalette()` method available, which is a factory method to which we pass the name of the palette root folder defined in the layer file, in this case `MyPaletteItems` (see Listing 9-15). All remaining work is handled for us by the `PaletteFactory` and `PaletteController` classes.

The second parameter requires implementing the `PaletteAction` class, offering actions for certain events. In the simplest case, you provide an empty implementation of this abstract class by defining each method as `null` or returning an empty array.

Listing 9-15. *Creating a palette and connecting it to a TopComponent*

```
private MyTopComponent() {
   ...
   try {
      associateLookup(Lookups.fixed(
         PaletteFactory.createPalette("MyPaletteItems", new MyActions())));
   } catch(IOException e) {
      // MyPaletteItems cannot be found
   }
}
```

Creating a Palette from a Node Hierarchy

Components in a palette are represented by NetBeans `Node` classes. However, in the previous section we defined each component via an XML file. The NetBeans Platform then provided a normal node to represent each XML file. In this section, we look at how a node implementation is used to create palette components, rather than doing so via XML files.

It is important to realize that a node hierarchy used in this way must consist of three levels. The uppermost level is a single root node that you pass to the `createPalette()` method to generate the components. The middle level, which consists of the child nodes of the root node, defines the categories in the palette. Finally, the second level's children define the palette components.

We'll present an example to show how this fits together. We'll create a palette that allows the user to manage music albums via drag-and-drop onto a playlist, as shown in Figure 9-8.

Figure 9-8. *Using the palette for managing music albums*

Node Classes for Creating and Displaying Data

As you saw in Chapter 7, the `Children` class creates and manages child nodes, while also acting as their container. We use this class to manage the categories displayed as genres in the palette. As in most such cases, we start by extending `Children.Keys<String>`, which has the `createNodes()` method for generating nodes. In the example that follows (see Listing 9-16), this approach is used to create three genre nodes.

It's possible you'd create the genre nodes from records retrieved from a database. To do this, see Chapter 13, where the connection and usage of databases in the context of NetBeans Platform Applications is discussed, and extend the example by adding a palette.

Start with the `GenreNode`, which is quite a simple class. Its constructor receives a `Genre` as a parameter, which is immediately passed to the `AlbumNodeContainer` class, at which point we find ourselves within the second and third levels of the node hierarchy.

Listing 9-16. *All genres shown on the palette are managed by the GenreNodeContainer. A genre is represented by the GenreNode.*

```
public class GenreNodeContainer extends Children.Keys<String>{
   protected void addNotify() {
      setKeys(new String[] {"root"});
   }
   protected Node[] createNodes(String root) {
      return(new Node[]{
         new GenreNode("Techno, Trance and Dance"),
         new GenreNode("Rock and Pop"),
         new GenreNode("Country and Classic")});
   }
}
public Class GenreNode extends AbstractNode{
   public GenreNode(String genre) {
      super(new AlbumNodeContainer(genre));
      this.setDisplayName(genre);
   }
}
```

The `AlbumNodeContainer` class, responsible for the creation of albums for a particular genre, is defined in the same way as the `GenreNodeContainer` class. In the example in Listing 9-17, we again create three nodes, this time of the `AlbumNode` type and using the `createNodes()` method, which receives the genre when the `addNotify()` method is invoked. That happens when users expand the category node. Using the `genre` parameter, a database for suitable albums that match the genre might be queried. Album data is managed by the `Album` class.

Listing 9-17. *The AlbumNodeContainer class manages nodes of a certain genre for albums.*

```
public class AlbumNodeContainer extends Children.Keys<String>{
   private String genre = new String();
   public AlbumNodeContainer(String genre) {
      this.genre = genre;
   }
   protected void addNotify() {
      setKeys(new String[] {genre});
   }
   protected Node[] createNodes(String genre) {
      return(new Node[] {
         new AlbumNode(
            new Album("Tunnel Trance Force 39", "42", "2","2007",
                  "com/galileo/netbeans/Module/cover_small.jpg",
                  "com/galileo/netbeans/Module/cover_big.jpg")),
         new AlbumNode(
            new Album("Dream Dance 43", "39", "3", "2007",
                  "com/galileo/netbeans/Module/cover2_small.jpg",
                  "com/galileo/netbeans/Module/cover2_big.jpg")),
         new AlbumNode(
            new Album("DJ Networx 31", "45", "2", "2006",
                  "com/galileo/netbeans/Module/cover3_small.jpg",
```

```
                "com/galileo/netbeans/Module/cover3_big.jpg"))
        });
    }
}
```

Finally, the `AlbumNode` class is responsible for display of the albums in the palette (as you can see in Figure 9-8). Since an `AlbumNode` does not own its own nodes, we pass the empty container `Children.LEAF` to the superclass via the constructor.

Use `setDisplayName()` to define the name shown on the palette. To make the display name more attractive, you may use HTML tags as well. In Listing 9-18, you can see the `getLabel()` method, which constructs an HTML string that shows the album data in a table and is used by `setDisplayName()` in the constructor defining the component display name. The value of `getHtmlDisplayName()` is used by the Palette Manager, which can be opened after right-clicking inside the palette.

Next, `getIcon()` shows the component icon, which in this case is an album cover. The user selects whether small or large icons are displayed, assuming that they have provided both types of icons.

Listing 9-18. *AlbumNode is responsible for displaying palette components.*

```
public class AlbumNode extends AbstractNode {
    private Album album = null;
    public AlbumNode(Album album) {
        super(Children.LEAF);
        this.album = album;
        this.setDisplayName(getLabel());
    }
    public String getHtmlDisplayName() {
        return "<b>" + album.getTitle() + "</b> (" + album.getTracks() + " Tracks)";
    }
    public Image getIcon(int type) {
        return album.getIcon(type);
    }
    private String getLabel() {
        String label = new String("<html>" +
            "<table cellspacing=\"0\" cellpadding=\"1\">" +
                "<tr>" +
                    "<td><b>Title </b></td>" +
                    "<td>" + album.getTitle() + "</td>" +
                ...
    }
}
```

Creating the Palette

Once we create the node hierarchy, providing data for the palette, we initialize the palette and add it to the Lookup of the `PlaylistTopComponent`. That ensures the palette is displayed whenever the playlist is active.

Use the constructor of the PlaylistTopComponent to provide a root node, from which all child nodes are created. The root node receives a new instance of GenreNodeContainer, handling creation and management of child nodes. Finally, we require a PaletteAction instance, which can simply be empty for now. Later, use createPalette() to create the PaletteController, which is then added to the TopComponent's local Lookup.

```
private PlaylistTopComponent() {
   ...
   Node root = new AbstractNode(new GenreNodeContainer());
   PaletteActions a = new MyPaletteActions();
   PaletteController p = PaletteFactory.createPalette(root, a);
   associateLookup(Lookups.fixed(p));
}
```

Implementing Drag-and-Drop Functionality

What's still missing is drag-and-drop functionality, which lets us drag albums from the palette and drop them onto the TopComponent. Two additional pieces of code are necessary to implement this functionality. First, make some changes to the Album and AlbumNode classes. Next, add code to the TopComponent, which must react appropriately when albums are dropped onto its surface.

The data we transfer onto the TopComponent is defined by the Album class. To allow data to be draggable, we must define the Transferable interface in this class (see Listing 9-19). We create a new DataFlavor, so data can be identified. We create a static instance of the DataFlavor class in the Album. Next, with the method getTransferDataFlavors(), we return our DataFlavor.

Call the method getTransferData() from the TopComponent, using this to retrieve the album instance, assuming the returned DataFlavor is of the type DATA_FLAVOR. If a different DataFlavor is returned, throw an exception.

Listing 9-19. *The album contains the data and implements the Transferable interface to provide it via drag-and-drop.*

```
public class Album implements Transferable {
   public static final DataFlavor DATA_FLAVOR =
            new DataFlavor(Album.class, "album");
   ...
   public DataFlavor[] getTransferDataFlavors() {
      return new DataFlavor[] {DATA_FLAVOR};
   }
   public boolean isDataFlavorSupported(DataFlavor flavor) {
      return flavor == DATA_FLAVOR;
   }
   public Object getTransferData(DataFlavor flavor)
           throws UnsupportedFlavorException {
      if(flavor == DATA_FLAVOR) {
         return this;
      } else {
         throw new UnsupportedFlavorException(flavor);
      }
```

```
   }
}
```

Be aware that the user will not be dragging an `Album` object, but the `Node` that represents it. We add the drag functionality to the `Node`, rather than to the `Album` object itself. This is done by overriding the `drag()` method in the `Node` class, which provides a `Transferable` instance, in our case the `Album` instance hiding behind the `AlbumNode`.

```
public class AlbumNode extends AbstractNode {
   private Album album = null;
   ...
   public Transferable drag() throws IOException {
      return album;
   }
}
```

Finally, we extend the `PlaylistTopComponent` such that it takes an action when the palette component is dropped. We also require a `TransferHandler` registered on the object that displays the result of the dropped component. In our case, we use the table `albums` with its scroll pane `scrollPane`.

To define the data that we want to accept and process that data, we begin by implementing our own `TransferHandler`, called `AlbumTransferHandler`. In doing so, we simply override two methods. First, we override the `canImport()` method, which is called when a component is on or over the `TopComponent`. Here, we determine whether the component is accepted. The test is based on the `DataFlavor` defined in the `Album` (see Listing 9-20). If this method returns `true`, a suitable mouse pointer is shown to the user, indicating that the `TopComponent` is able to handle the drop event.

Second, we override the `importData()` method, which is called when the drop event is invoked. Via the `TransferSupport` object, which is received as a parameter, we obtain the `Transferable` instance provided by the `drag()` method from the `AlbumNode`. We then use `getTransferData()` and our `DataFlavor` to obtain the related `Album`, and then add the data to the `JTable`.

Listing 9-20. *Accepting the drop of an album*

```
final class PlaylistTopComponent extends TopComponent {
   private TransferHandler th = new AlbumTransferHandler();
   private PlaylistTopComponent() {
      ...
      albums.setTransferHandler(th);
      scrollPane.setTransferHandler(th);
   }
   private final Class AlbumTransferHandler extends TransferHandler {
      public boolean canImport(TransferSupport support) {
        return support.isDataFlavorSupported(Album.DATA_FLAVOR);
      }
      public boolean importData(TransferSupport support) {
         try {
            Album a = (Album) support.getTransferable().
                    getTransferData(Album.DATA_FLAVOR);
```

```
            DefaultTableModel model = (DefaultTableModel)albums.getModel();
            model.addRow(new Object [] {
                    a.getTitle(), a.getTracks(), a.getCDs(), a.getYear()} );
            return(true);
         } catch(Exception e) {
            e.printStackTrace();
            return false;
         }
      }
   }
}
```

Summary

In this chapter, you learned that the NetBeans Platform provides more than APIs. It also provides a range of self-contained components. These can be used by services and extended by service provider interfaces.

We started by looking at the help system, which you can very easily use in your own applications. Among other things, you learned how to create a new helpset and how to set up context-sensitive help. We also looked at the Output window, displaying processing messages, as well as the Navigator and the Properties window. Both are used to display data about the structures in your application. Finally, we looked at the Options window and the palette. You can use these or extend them quite easily via their APIs and SPIs.

CHAPTER 10

Internationalization and Localization

Let's Translate Our Code!

This chapter shows how to internationalize an application, adapting it to several languages. You will learn the specifics on internationalizing source code and manifest files, as well as adapting help topics and other resources, such as graphics.

Professional, flexible applications must be designed to adapt as easily as possible to their specific target countries and languages. For this, the Java API, the NetBeans Platform, and the NetBeans IDE support internationalization at minimum effort.

INTERNATIONALIZATION VS. LOCALIZATION

Internationalization is the process of designing/implementing an application so that it can easily be localized. This is the first step, and is done by the software developer. For example, if you put something like `NbBundle.getMessage(...)` in your code, you internationalize your application.

Localization is the process of creating/providing of resources (text, icons, etc.) for a specific country/language. So, if you provide, for example, a `Bundle_es_ES.properties` file, you localize your application.

An application must be internationalized before it can be localized.

String Literals in Source Code

String literals in source code are outsourced in properties files. The language-dependent literals can be separated and changed into other languages. This is possible even after the release of an application. The constants are saved as key/value pairs in a properties file:

```
CTL_MyTopComponent = My Window
HINT_MyTopComponent = This is My Window
```

Any such resource file is handled by the Java class `ResourceBundle`. A `ResourceBundle` is responsible for resources of a particular locale setting that specifies both country and language. For easy handling of properties files and access to a `ResourceBundle` instance, the NetBeans

Platform provides the class `NbBundle`. The resource file must be named `Bundle.properties`, and typically such a file is created for each package. The easiest way to create `ResourceBundle` objects is by the following call:

```
ResourceBundle bundle = NbBundle.getBundle(MyTopComponent.class);
```

The class `NbBundle` creates a `ResourceBundle` object for the `Bundle.properties` file, provided in the package of the class `MyTopComponent`. The required string literal is easily read with the `ResourceBundle` method `getString()`:

```
String msg = bundle.getString("CTL_MyTopComponent");
```

If only a few literals are required inside your class, use the method `getMessage()` to read a literal directly without creating a `ResourceBundle` instance:

```
String msg = NbBundle.getMessage(MyTopComponent.class, "CTL_MyTopComponent");
```

Also, it is possible to add a placeholder to your string literals. This is most often required for data or file names/paths. A pair of braces is used as a placeholder, which includes the number of the parameter:

```
Result = {0} MP3-Files found for {1}
```

Pass these parameters to the `getMessage()` method, which replaces the placeholder with the parameter. Alternatively, you can pass up to three parameters or an array with an unlimited number of parameters:

```
String label = NbBundle.getMessage(MyTopComponent.class,
                                   "Result",
                                   new Integer(results.size()),
                                   search.getText());
```

For each properties file, only the literals of a single language are saved. To add another language to your application, save the literals—with the same keys—in a properties file named `Bundle_<language>_<country>.properties` in the same folder. The class `NbBundle` returns the `ResourceBundle` using the method `getBundle()`. This equals the locale setting, which `Locale.getDefault()` returns. The `Bundle.properties` file that does not contain language and country identification is the default package. This package is used if there is no bundle for the locale setting available. A specific bundle can also be requested by passing a `Locale` object to the `getBundle()` method. To know in which order the bundles are searched, the method `NbBundle.getLocalizingSuffixes()` lists all suffixes in the order used.

The method `Locale.getDefault()` returns the locale setting of the virtual machine by default. To run the whole application with a specific locale setting, set the command-line parameter `locale`. This parameter passes a language and country identification to the application. You can find more information on this in Chapter 11.

The NetBeans IDE provides a wizard for the internationalization of string literals for your source files. Let the wizard scan your files for strings, which can be moved to a properties file (see Figure 10-1). You can thereby edit the key, the value, and the code to be pasted, rather than the literal. You can find the wizard by going to Tools ➤ Internationalization ➤ Internationalization Wizard.

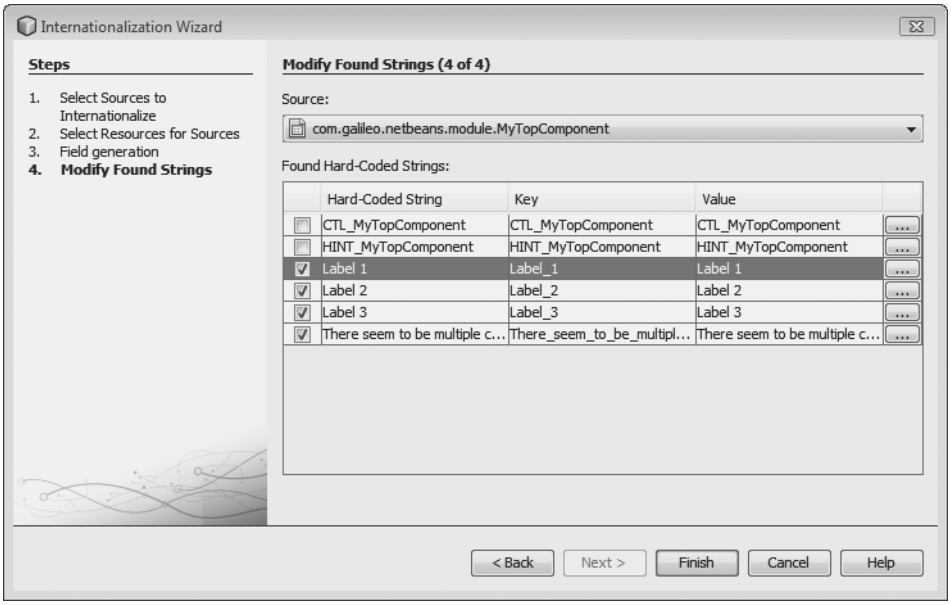


Figure 10-1. *Move string literals automatically to a bundle and paste the needed source code with the Internationalization wizard.*

String Literals in the Manifest File

Among the string literals of the source files, you can also internationalize the textual information of the manifest file. There are two options for doing so. The first option appends a language identifier to the manifest attributes for using the same attribute several times:

```
Manifest-Version: 1.0
OpenIDE-Module: com.galileo.netbeans.module
OpenIDE-Module-Name: My Module
OpenIDE-Module-Name_de: Mein Modul
```

The second option (preferred by the author) is to move the attributes you intend to internationalize to a properties file. The attribute names are used as keys and are provided in the according bundle for each language. For the attributes to be read off the bundle, notify the manifest file by using the `OpenIDE-Module-Localizing-Bundle` attribute, as shown in Listings 10-1 through 10-3 (see also Chapter 3).

Listing 10-1. *Manifest.mf*

```
Manifest-Version: 1.0
OpenIDE-Module: com.galileo.netbeans.module
OpenIDE-Module-Localizing-Bundle: com/galileo/netbeans/module/Bundle.properties
```

Listing 10-2. *Bundle.properties*

```
OpenIDE-Module-Name = My Module
```

Listing 10-3. *Bundle_de.properties*

```
OpenIDE-Module-Name = Mein Modul
```

Internationalization of Help Pages

Generally, the help pages, including the helpset configuration files, are internationalized like properties files (see the "String Literals in Source Code" section) by appending country and/or language identifiers. Since a helpset typically consists of a large number of files, this results in a confusing structure. Therefore, it is possible to store the files intended for internationalization in a subfolder (see Figure 10-2). Language and country identification is no longer necessary, as this is already represented by the subfolder.

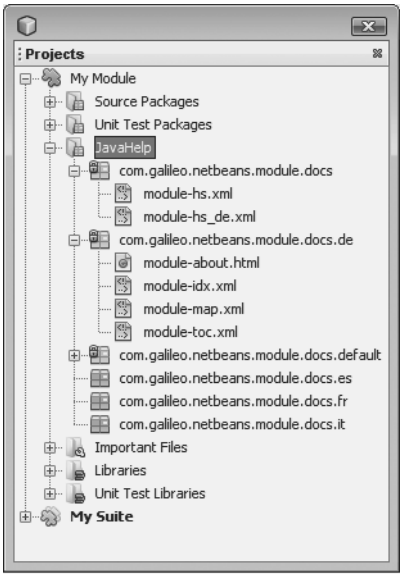


Figure 10-2. *Helpsets for specific languages are stored in separate folders.*

Only the helpset file `module-hs.xml` remains in the default folder, and has no identifier. In this file, you delegate to the corresponding folders (see Listing 10-4). The helpset file without an identifier is always used when the active locale setting does not match with the existing files. Normally, the default package contains the English version.

Listing 10-4. *Helpset file that refers to a language-specific package*

```
<maps>
   <homeID>com.galileo.netbeans.module.about</homeID>
   <mapref location="default/module-map.xml"/>
</maps>
<view mergetype="javax.help.AppendMerge">
   <name>TOC</name>
   <label>Contents</label>
   <type>javax.help.TOCView</type>
   <data>default/module-toc.xml</data>
</view>
<view mergetype="javax.help.AppendMerge">
   <name>Index</name>
   <label>Index</label>
   <type>javax.help.IndexView</type>
   <data>default/module-idx.xml</data>
</view>
```

Internationalizing Other Resources

Consider previously mentioned areas of internationalization of applications. Among those of greatest importance, there are additional possibilities for internationalizing other application components. The NetBeans Platform provides such possibilities.

Graphics

Not only can text be adapted as language and country specific, but also graphics such as icons. For this purpose, the `ImageUtilities` class provides a version of the `loadImage()` method that is usually used for loading graphics. Set a Boolean parameter for whether an available language/country-specific version of the graphic should be loaded against the current locale setting. The method `NbBundle.getLocalizingSuffixes()` lists possible identifications used as the search order.

```
Image img = ImageUtilities.loadImage("resources/icon.gif", true);
```

If the active locale setting is, for example, `de_DE` on this call, first `icon_de_DE.gif` and `icon_de.gif` are searched.

Any File

The NetBeans Platform defines a special protocol for loading other internationalized resources. This is the `nbresloc` protocol, an extension of the `nbres` protocol that loads resources of all available modules. You can easily create a `URL` object for a resource addressed with this protocol:

```
URL u =new URL("nbresloc:/com/galileo/netbeans/module/icon.png");
ImageIcon icon = new ImageIcon(u);
```

If the locale setting is `de_DE` and a file named `icon_de_DE.png` or `icon_de.png` exists, then this icon is loaded instead of `icon.png`.

Folders and Files

The System Filesystem provides two special attributes to internationalize names and icons of folders and files. This makes sense for, e.g., menus whose names are only in the layer file declared and cannot be read from the `NbBundle` class. These attributes are `SystemFileSystem.localizingBundle` and `SystemFileSystem.icon`. With the first, link to your resource bundle, leaving the `.properties` extension out. In this bundle, a key is searched that matches the full path of the folder or file that the `SystemFileSystem.localizingBundle` attribute contains. In the following example (see Listings 10-5 through 10-7), it concerns `Menu/MyMenu` and `Menu/MyMenu/MySubMenu`. With the `SystemFileSystem.icon` attribute, you may additionally set an icon for the folder or the file. Use the `nbresloc` protocol to load the internationalized version.

Listing 10-5. *Layer.xml*

```
<folder name="Menu">
   <folder name="MyMenu">
      <attr name="SystemFileSystem.localizingBundle"
         stringvalue="com.galileo.netbeans.module.Bundle"/>
      <folder name="MySubMenu">
         <attr name="SystemFileSystem.localizingBundle"
            stringvalue="com.galileo.netbeans.module.Bundle"/>
         <attr name="SystemFileSystem.icon"
            urlvalue="nbresloc:/com/galileo/netbeans/module/icon.png"/>
      </folder>
   </folder>
</folder>
```

Listing 10-6. *Bundle.properties*

```
Menu/MyMenu=Extras
Menu/MyMenu/MySubMenu=My Tools
```

Listing 10-7. *Bundle_de.properties*

```
Menu/MyMenu=Extras
Menu/MyMenu/MySubMenu=Meine Tools
```

In addition to these two special attributes, NetBeans Platform 6.5 introduces a generic approach to localizing attributes. Instead of `stringvalue` or `urlvalue`, you can use `bundlevalue` to point to a string literal in a properties bundle. You can use `bundlevalue` for any attributes in

an `XMLFileSystem` like the System Filesystem. We already used this approach while registering actions in Chapter 4, registering `DataLoader`s in Chapter 7, and adding Option panels in Chapter 9.

Let's now look at an action registration entry in the layer file. We want to place the action's name in a properties bundle instead of writing it directly into the layer file. To load the value of the `displayName` attribute from a properties bundle, the entry is defined as shown in Listing 10-8.

Listing 10-8. *Usage of bundlevalue attributes*

```
<file name="com-galileo-netbeans-module-MyFirstAction.instance">
   <attr name="displayName"
      bundlevalue="com.galileo.netbeans.module.Bundle#CTL_MyFirstAction"/>
</file>
```

With this entry, we are able to put the name of the action in a properties file, as shown in the Listing 10-9.

Listing 10-9. *Values of the layer attributes in properties files*

```
com/galileo/netbeans/module/Bundle.properties
   CTL_MyFirstAction=My First Action
```

Administration and Preparation of Localized Resources

Up until this point, localized resources have been stored in the same folder as the module, whether `Bundle.properties` files or graphics are found. But how do you keep resources for each language separated and later extend an already provided module with an additional translation? In this scenario, the NetBeans Platform shines. It easily provides an opportunity to separate localized resources from the rest (basically the classes) of the module. It provides a `locale` folder, stored under the module storage folder (see Figure 10-3). The resources for a language in a JAR archive in this folder are extended with a language/country identifier. The archive must have the same name as the module JAR archive. In this *locale extension archive*, all language/country-specific resources are managed. They have the same package structure as the module. The resources are separated and can be updated individually in this way. The translation of the NetBeans Platform modules is done in the same manner.

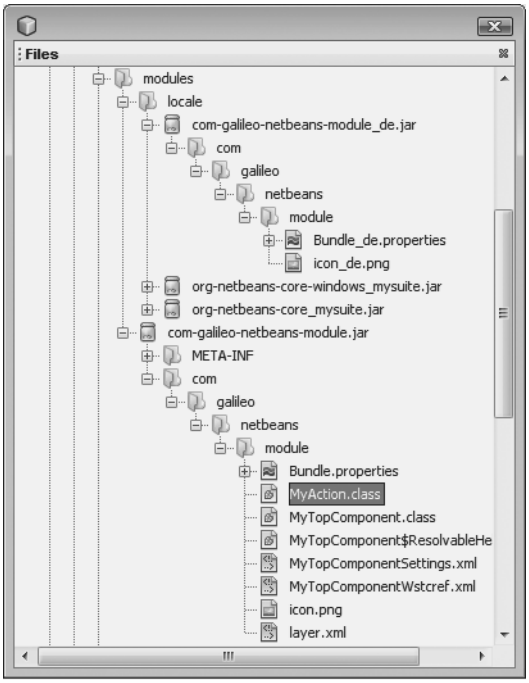


Figure 10-3. *Allocation of language-specific resources in a separate JAR archive in the locale folder*

Note that the individual localized resources still need the language/country identifier. The locale extension archive needs no manifest file, because the archive is exclusively identified by the name of the localized packet and is added to the classpath of the module classloader (see Chapter 2).

Here in this example, the resources for German are in the `locale` folder as a locale extension archive. The English resources that have no identifier are the default resources and are provided in the JAR archive of the module. It is interesting that you can put the default resources into a locale extension archive, because it has no identifier, has the same name as the module, and is in the `locale` folder. Therefore, the resources you plan to localize are completely separated from the module itself. Later, adding another language is done by a third party because it is obvious which resources must be localized.

Summary

In this chapter, you learned how to prepare your application for different countries and languages (internationalization) and how to provide different resources for different countries and languages (localization). We started by looking at string literals in source code. You saw that with the help of the `NbBundle` class, together with a NetBeans IDE wizard, it is very easy to put your language-specific content into a separate properties file.

We also looked at the localization of help pages. Not only strings can be localized, since there is also support for loading locale-specific graphics. You also learned how to localize layer file values, and you learned how to distribute your localized content as a separate module.

CHAPTER 11

Real-World Application Development

Let's Understand the Development Cycle!

This chapter shows how to create and configure a real application based on application modules, including customizing the NetBeans Platform to fit particular application needs. Finally, the chapter shows how to create an application distribution that can be shipped to end users.

Creation

To deliver modules as complete and standalone applications, create a NetBeans Platform Application. This project type, similar to a Module Suite, is a container for application modules, providing configuration environments for the product. Normally, an application is created before the modules that define it, which are added to the application afterward. Application modules can be tested during the development cycle as a complete application. Furthermore, an application is necessary to define module dependencies. The creation of a NetBeans Platform application is supported inside the NetBeans IDE via a special wizard. The wizard is started with File ➤ New Project. Select the category NetBeans Modules and the project type NetBeans Platform Application. The application modules are added using the project properties under Sources. If you are creating a new module, define the assigned application in the wizard.

You define the NetBeans Platform modules of your application in the Libraries category of the application's Project Properties dialog. There, you can activate or deactivate modules for inclusion in the application. Note that the modules you create can define dependencies only on modules that are activated here. If you want to define a dependency on a module but you cannot find it, you probably simply need to activate it in the Project Properties dialog.

The application is configured using project properties beneath the category Build. If choosing to create a standalone application, specify a branding name. Its purpose is described in the "Customization of Platform Modules" section, which follows. Additionally, you can

define the name of your application and select an icon, shown as the window icon in the About dialog. Selecting the Build ➤ Splash Screen category, using the tree on the left-hand side of the dialog (see Figure 11-1), the image displayed as the splash screen during startup of the application is specified. A progress bar is used on this screen to display status of application initializations, and to display performed actions. Position and size of these components are defined using drag-and-drop. Color is specified using a combo box.

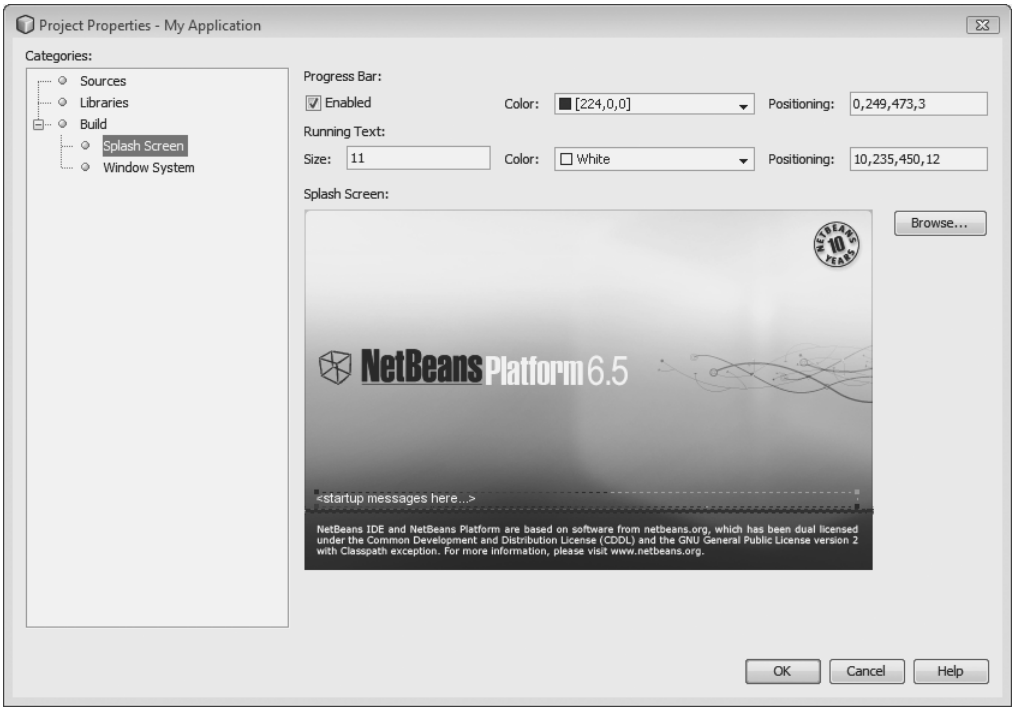


Figure 11-1. *Configuration of the NetBeans Platform application*

Customization of Platform Modules

Similar to the localization concept—as described in Chapter 8—the generic NetBeans Platform modules can be adapted to fit specific application needs. In most cases, this is helpful or necessary for labels and icons. This is accomplished using the *branding ID* defined for applications. This ID is used similarly to language or country suffixes, which are appended to the expected resources. Those suffixes can be combined. If you want to, e.g., change the name of the `Favorites` window of the NetBeans Platform in an application using the branding ID `my_application` and locale `de_DE`, you add a resource bundle containing the window's new name (`Bundle_my_application_de_DE.properties`). Wrap this file in a locale extension archive, using the same package structure as the Platform module. This archive lives in the `modules/locale` directory. The archive uses the same suffix: `my_application_de_DE`. This mechanism is used by the NetBeans IDE to brand the modules `org-netbeans-core` and `org-netbeans-core-windows` to customize the title of the IDE.

The branding ID is defined inside the NetBeans IDE, using the properties of the application in the Build category. Additionally, the ID can be passed using the command-line parameter `branding` (see the "Customizing the Launcher" section, which follows). To query or change the ID at runtime, do this with the static methods `NbBundle.getBranding()` and `NbBundle.setBranding()`. As already described for the localization, the ordered search list of the possible suffixes can be retrieved with the static method `NbBundle.getLocalizingSuffixes()`. First, a file with product-, language-, and country-specific suffix(es) is searched. If no file can be found, the file without a suffix is used. The search order for the example resource `icon.png` looks like this:

- `icon_<branding id>_<language code>_<country code>.png`
- `icon_<branding id>_<language code>.png`
- `icon_<branding id>.png`
- `icon_<language code>_<country code>.png`
- `icon_<language code>.png`
- `icon.png`

Customizing the Launcher

Applications are started using a platform-specific launcher provided by the NetBeans Platform. For the Windows OS, the launcher is an `.exe` file. Execution and start of the application are influenced using command-line parameters (see Table 11-1). Be aware of the use of duplicate backslashes or single slashes for all path definitions.

Table 11-1. *Command-line parameters and their meanings*

Parameter	Description
`clusters <path>`	Path to clusters containing modules, separated with ; for Windows and : for UNIX.
`branding <id>`	Definition of the branding ID.
`jdkhome <path>`	Path to the JDK base path.
`J<JVM Option>`	Used to pass parameters to the virtual machine. Frequently used for defining properties (e.g., `-J-Dorg.netbeans.core.level=100`).
`cp:p <classpath>`	Used to prepend resources to the classpath of the application. These are available from the Java system classloader using `ClassLoader.getSystemResource()`.
`cp:a <classpath>`	Used to append resources to the classpath of the application. These are available from the Java system classloader using `ClassLoader.getSystemResource()`.
`laf <L&F classname>`	Definition of the class name of the look and feel to be used.
`fontsize <size>`	Sets a global font size for the application.

Table 11-1. *Command-line parameters and their meanings (Continued)*

Parameter	Description
`locale <language[:country]>`	Locale setting to be used (e.g., `de:DE`). Keep in mind that the language specifier and the country specifier are separated by a colon.
`userdir <path>`	Allows defining an alternative path to store user-specific application settings. This parameter runs multiple instances of applications, each reading its stored data from separate directories.

These parameters are defined and passed either directly to the launcher or by adding to the file `etc/<branding id>.conf`, which is part of the application distribution. In the application configuration file, options are defined using the attribute `default_options`. Additionally, the path to the selected JDK is defined, along with the user directory and further additional clusters. To control command-line parameter settings during the application development inside the NetBeans IDE, use the attribute `run.args.extra`. It is defined in the `project.properties` file of the application or the standalone module (if it does not belong to an application). For example, selecting different locale settings for testing purposes and enabling more detailed logging require adding the following entry to the `project.properties` file:

```
run.args.extra = --locale fr:FR \
                 -J-Dcom.galileo.netbeans.module.level=100
```

Distribution

There are different ways to distribute your application. The NetBeans IDE provides wizards for each distribution type. We will look at them in the following sections.

Distribution As a ZIP Archive

The most common way to ship an application, besides an installer, is the ZIP distribution. For this kind of distribution, all parts of the application are packed into one master file, where all items necessary to execute the application are contained. For a NetBeans Platform application, these are modules of the NetBeans Platform, your own application modules, a launcher to start the application, and the necessary configuration files. Creating this package is done using the build script, accessible from the context menu of the application and started with Build ZIP Distribution. The distribution is put into the `dist` directory in the project directory of the application, as shown in Figure 11-2.

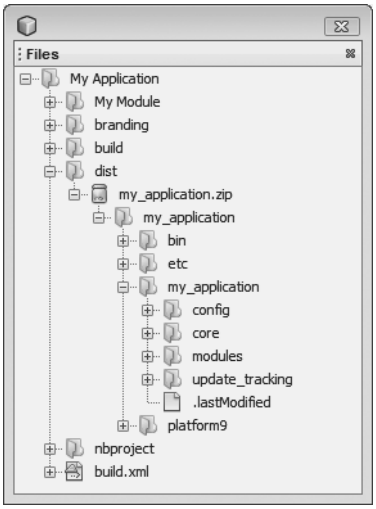


Figure 11-2. *Parts of a ZIP distribution*

- The `bin` directory contains platform-specific launchers, one of those an `.exe` file.
- The `etc` directory contains configuration files used by the launcher. One of the files is `<branding id>.conf`, which allows configuration of command-line parameters or the path to the JDK (see the preceding "Customizing the Launcher" section).
- The directories `my_application` and `platform9` are clusters. The cluster `my_application` contains application modules, including all configuration data and customized platform modules (see the "Customization of Platform Modules" section earlier in the chapter). The cluster `platform9` contains all modules from the NetBeans Platform, including the NetBeans runtime container (see Chapter 2).

Distribution via Java Web Start

Another way to distribute your application is via Java Web Start. This technique allows applications to be downloaded and started directly from the Internet. To create a Web Start–enabled distribution, select the Build JNLP Application item from the context menu of your application. This results in a `.war` file inside the `dist` directory, which contains the complete application, ready to be copied into the `deploy` directory of servlet containers. The file `WEB-INF/web.xml` is a deployment descriptor, defining the JNLP servlet distributed in the `WEB-INF/lib` directory.

Mac OS X Application

The third and final approach is to build a Mac OS X application. This is done using the corresponding menu item Build Mac OS X Application in the context menu of the application. Bear in mind that this is not possible on a Windows platform, since one of the commands used is `ln`, which is not available on Windows.

Summary

In this chapter, you learned how to create and configure your standalone NetBeans Platform application. We also looked at the customization of NetBeans Platform modules, where we adapted out-of-the-box modules to our application-specific needs.

The launcher of your created application can be influenced in various ways. Therefore, we had a look at each of the command-line parameters and how they are used. At the end of this chapter, we dealt with the creation of deployment-ready application distributions.

CHAPTER 12

Updating a NetBeans Platform Application

Let's Allow the User to Add Features at Runtime!

This chapter covers updating applications already distributed to end users. This is done by providing new or updated modules in one of several different ways. The focus of this chapter is on the creation, architecture, and distribution of update packages.

Overview

During software lifecycles, it is quite common for users to request patches when experiencing unexpected error messages. In addition, development teams often want to make new features available between one release and the next. It would be cumbersome to redistribute the entire application when providing a patch or a new feature. A major advantage of modular architectures is its support for the distribution of modules that provide patches or updates for existing applications.

For the user, installation of updates must be as simple and intuitive as possible. To that end, the NetBeans Platform provides an Auto Update service connected to the Plugin Manager. The Plugin Manager automatically searches for updated or new modules in a set of predefined update centers, dynamically loading them at runtime. Additionally, users are able to manually install downloaded updates or new modules into the Plugin Manager.

The Auto Update Service

Updates are made available as NBM files. These update packages are offered via an update center (see Figure 12-1). Within a NetBeans Platform application, users register update centers in the Plugin Manager so that the Auto Update service can search those centers for changes.

The registration of update centers in the Plugin Manager is done manually by users or by a module that automatically registers the update centers. Users configure the application so

that update centers are polled periodically for changes. Users may manually initiate this update task as well.

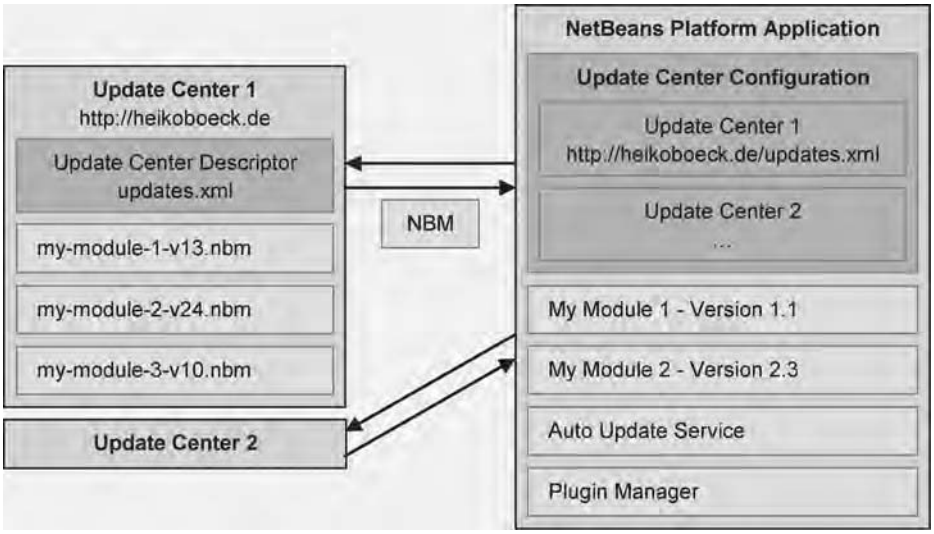


Figure 12-1. *Components making up Auto Update functionality and their dependencies*

The NBM File

Modules available as update packages are distributed in the form of NBM files. A file of this type is a JAR archive containing the actual module, together with its configuration data and update information required by the Plugin Manager (see Figure 12-2). It may also contain the libraries required by the module. The contents of the module JAR file and its related configuration files are described in Chapter 3.

One file we haven't yet discussed is the `info.xml` file. This file contains information displayed to the user when choosing modules in the Plugin Manager. The `manifest` element (see Listing 12-1) provides information from the module's manifest file, such as dependencies that must be met for the module to be installed. If the user chooses to download a module dependent on another module, the latter is automatically activated, so it can be downloaded simultaneously with the selected module. If dependencies cannot be satisfied, the user installs the module without being able to activate it.

License information may be added here, too, via the `license` element, which the user finds in the Plugin Manager before installation of the module begins. This forces the user to confirm that licensing requirements have been satisfied and the module can be installed.

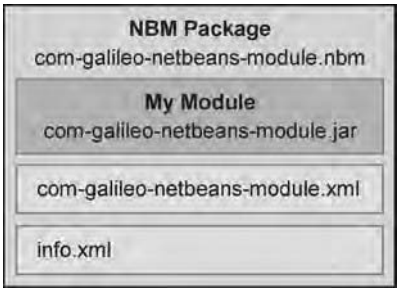


Figure 12-2. *Content of an NBM file*

Listing 12-1. *The NBM information file*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC
   "-//NetBeans//DTD Autoupdate Module Info 2.5//EN"
   "http://www.netbeans.org/dtds/autoupdate-info-2_5.dtd">
<module codenamebase="com.galileo.netbeans.module"
        distribution="./com-galileo-netbeans-module.nbm"
        downloadsize="7123"
        homepage="http://heikoboeck.de"
        license="AD9FBBC9"
        moduleauthor="Heiko Boeck"
        needsrestart="false"
        releasedate="2009/03/11">
   <manifest AutoUpdate-Show-In-Client="true"
             OpenIDE-Module="com.galileo.netbeans.module"
             OpenIDE-Module-Name="My Module"
             OpenIDE-Module-Implementation-Version="090311"
             ...
             OpenIDE-Module-Specification-Version="1.0"/>
   <license name="AD9FBBC9">Place your license information here
   </license>
</module>
```

An NBM file is created automatically by the NetBeans IDE. Right-click a module in the Projects window and choose Create NBM.

When you choose Create NBM, the IDE attempts to sign the NBM file. To make this possible, a keystore must be prepared and generated. Use the Keystores Manager in the IDE, which is only available after installation of the Mobility plugin. This is part of the full as well as the Java IDE distribution. The easiest way is to install this plugin over the Plugin Manager (Tools ➤ Plugins).

Open the Keystores Manager via Tools ➤ Keystores. Next, create a keystore file via Add Keystore, including a file name and location where the keystore is located. A recommended location is the `nbproject/private` folder of your module (see Figure 12-3). After entering a password of at least six characters, click OK to create the keystore.

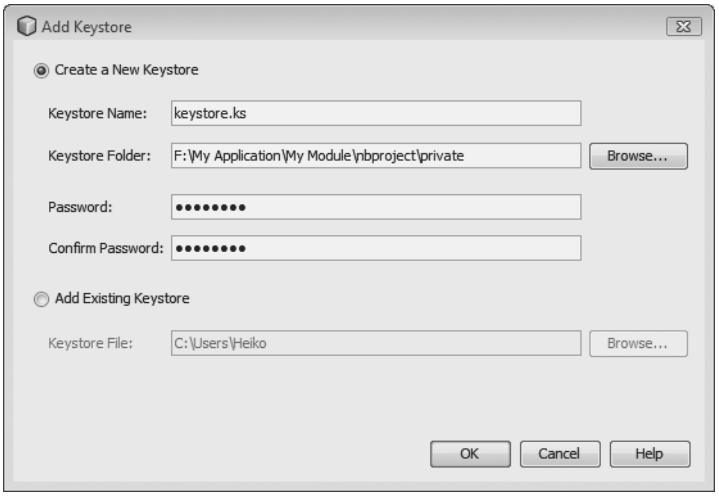


Figure 12-3. *Creating a keystore*

The newly created keystore must be provided with a key pair, with which the NBM file is signed. In the Keystores Manager, choose the keystore and then click the New button. The next dialog provides an alias, as well as certificate details, such as organization name (see Figure 12-4). Provide the same password as when creating the keystore. Next, click OK to close the dialog, which creates the key pair in the keystore.

Figure 12-4. *Creating a key pair*

To enable the IDE to find the keystore and key pair, define the following properties in the `nbproject/project.properties` file (found within the Projects window, under Important Files

➤ Project Properties). If this file does not exist, create it. Use the keystore key to define the path to the keystore relative to the module project folder. Use the `nbm_alias` key to set the alias for the key pair to be used, since one keystore may contain multiple key pairs.

```
keystore=nbproject/private/keystore.ks
nbm_alias=mymodule
```

Now invoke the Create NBM menu item again, which lets the IDE sign the NBM file. A dialog is shown for entering the password assigned to the keystore. Enter the correct password, enabling successful signature.

If you don't want to enter this password every time, you may define it in the `nbproject/private/private.properties` file (found in the Projects window, under Important Files ➤ Per-user Project Properties) as follows:

```
storepass=mypassword
```

Now, in the Plugin Manager, the user will see the module certificate, after downloading the selected module is completed, but before the installation process begins. Be aware that a warning will show that the module is not trusted. To prevent this, provide an official certificate from an official certificate vendor, such as VeriSign.

Update Centers

NBM files—that is, updates for a distributed application—are put into an update center, from which they can be downloaded. An update center is nothing more than a storage place where the module is placed, generally on a server accessible via the Internet. Use an update center descriptor, in the form of an XML file, to describe the module location and other details. In this way, the Auto Update service finds the module, while determining whether the module is updated or new.

The update center descriptor lists the content of the `info.xml` file (explained in the previous section) for each of the NBM files in the update center. As a result, the Auto Update service of the application need not open or download an NBM file to determine its version.

The `license` element is defined outside the `module` element (see Listing 12-2) to provide license information for multiple modules concurrently, while ensuring only one license is shown to the user. Multiple use of the `license` element is possible as well, providing a separate license for each module.

Listing 12-2. *Update center descriptor*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE module_updates PUBLIC
   "-//NetBeans//DTD Autoupdate Catalog 2.5//EN"
   "http://www.netbeans.org/dtds/autoupdate-catalog-2_5.dtd">
<module_updates timestamp="08/54/21/11/03/2009">
   <module codenamebase="com.galileo.netbeans.module"
           distribution="./com-galileo-netbeans-module.nbm"
           ...
   </module>
   <module codenamebase="com.galileo.netbeans.module2"
```

```
            distribution="./com-galileo-netbeans-module2.nbm"
            ...
    </module>
    <license name="AD9FBBC9">Place your license information here</license>
</module_updates>
```

The root element of the update center descriptor is the `module_updates` element. The `module_updates` element contains the `timestamp` attribute, which the Auto Update service compares with a timestamp obtained from previous connection to the update center. The Auto Update service reads an update center only when the timestamp date is more recent than the date of the last connection.

Optionally, modules may be grouped in the update center descriptor, via the `module_group` element. This allows modules to be displayed as a group in the Plugin Manager. In the `module` element, the `distribution` attribute is very important. It defines the location from which the module will be downloaded. Rather than a relative location, as shown in Listing 12-2, an absolute URL can be provided, pointing to the location of the module.

The address of an update center descriptor is defined as a URL in the update center settings in the Plugin Manager. The update center descriptor need not be manually created. Right-click the application project in the Projects window and choose Create NBMs. Aside from generating an individual NBM file for each module, an XML file named `updates.xml` is created. The `updates.xml` file is the update center descriptor. The NBMs and the update center descriptor are found in the `build/updates` folder, visible in the Files window.

Localized NBM Files

In Chapter 10, localizing the language-specific content of modules was explained. Two variations were discussed. First, localized resources can be placed directly within the module. Second, they are put in a JAR file and that file in the `locale` folder. In both cases, assume the developer has access to the module or to the application of which the module forms a part. It is possible to provide a localizing bundle for an application that has already been distributed.

Additional localizing bundles for an already distributed module are simple to create. Do this by creating an updated or new module and making it available via an update center. The only difference is that the `manifest` element in the `info.xml` file and in the update center descriptor are supplemented by the `l10n` element. The structure is as follows:

```
<l10n langcode="de"
      module_major_version="1"
      module_spec_version="1.0"
      OpenIDE-Module-Name="Mein Modul"
      OpenIDE-Module-Long-Description="German localization of My Module."/>
```

Use the `langcode` attribute to define the language for the NBM file. With version attributes, you define versions that the NBM file targets. The localizing bundle is activated only when localizing bundle versions match those of the installed module.

Configuring and Installing on the Client

To enable an application to connect to new or updated modules in an update center, the update center must be registered in the Plugin Manager, which can be opened via Tools ➤ Plugins. There, in the Settings tab (see Figure 12-5), update centers are defined. The URL of an update center must point to the location of the update center descriptor (see Figure 12-3). In the same way, update centers may be deactivated, thereby excluding them from the update process.

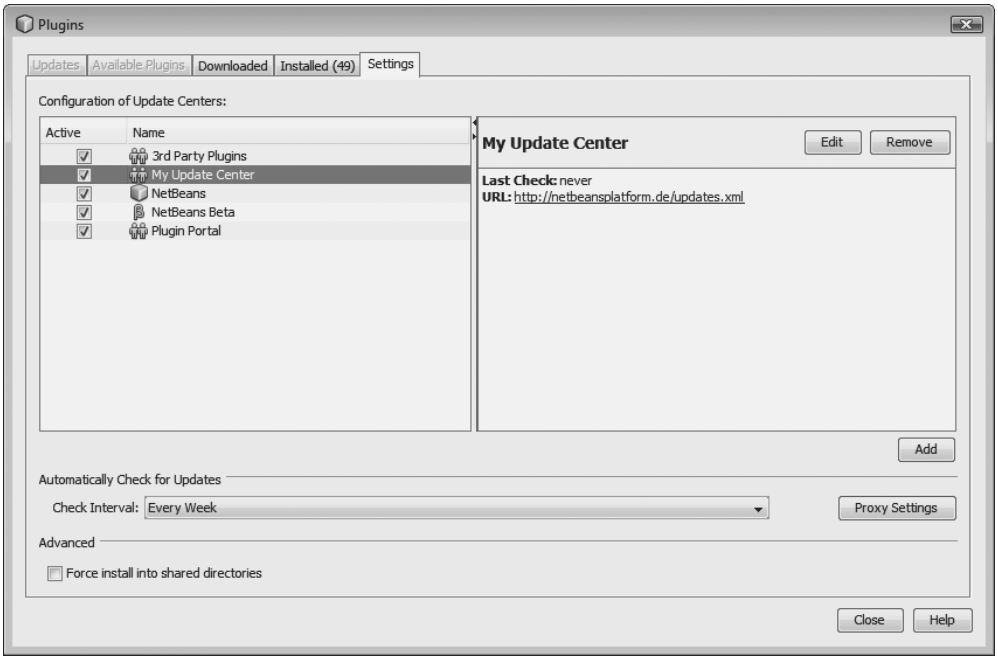


Figure 12-5. *Configuring an update center*

When switching to the Updates tab, use the Reload Catalog button to allow the Auto Update service to look for modules in the defined update centers. Updated versions of modules are sought for those modules that are already installed in the application. For new modules, use the same approach in the Available Plugins tab. Found modules are displayed immediately in a list, from which you can select those you need. Use the Update or Install buttons to add selected modules to the application.

Modules that are locally available—that is, those that are personally downloaded—can be installed as well. Switch to the Downloaded tab, and then click Add Plugins to open NBM files from disk into the Plugin Manager. Finally, click Install to install the selected modules.

The last tab discussed is the Installed tab (see Figure 12-6). All currently installed modules are listed there, organized into categories. You can deactivate modules here, as well as completely uninstall them.

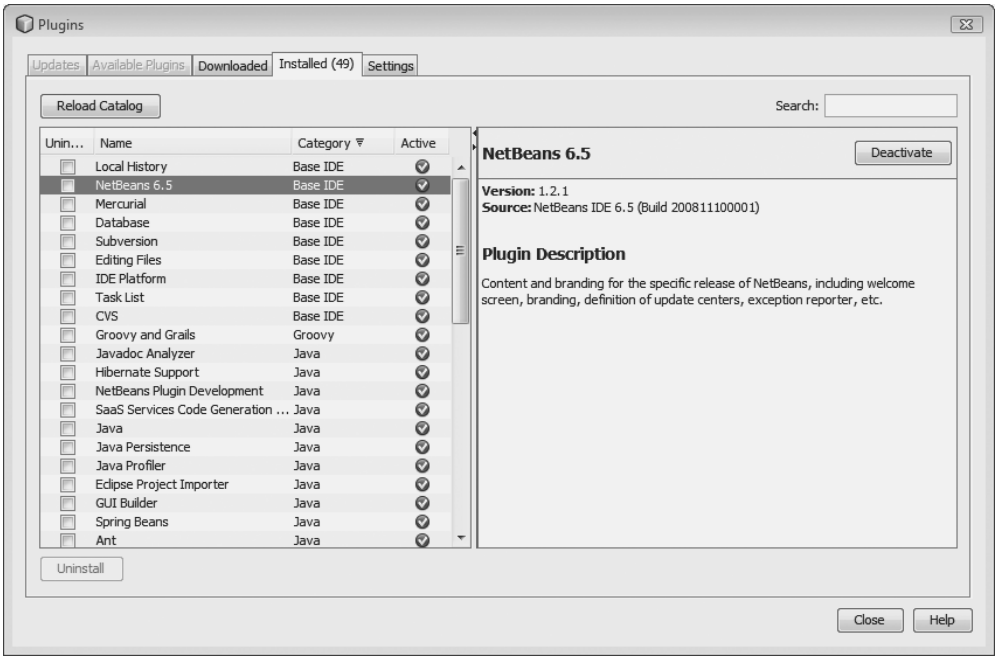


Figure 12-6. *The Installed tab lists all installed modules, where they can be activated and deactivated.*

New Update Center

As pointed out, users may register update centers in the application's Plugin Manager. Rather than by manual approach, you can add update center information to an existing or new module. Once this module is installed, the registered update centers become automatically available. As a result, a module may be provided that adds new update centers to an already distributed application.

To this end, choose File ➤ New File ➤ Module Development ➤ Update Center. The wizard registers update center information, consisting of the name and URL of the update center descriptor. This information is registered in the layer file, within the `Services/AutoupdateType` folder. Registration entries of this kind appear as follows:

```
<folder name="Services">
   <folder name="AutoupdateType">
      <file name="my_module_update_center.instance">
         <attr name="SystemFileSystem.localizingBundle"
                stringvalue="com.galileo.netbeans.module.Bundle"/>
         <attr name="enabled" boolvalue="true"/>
         <attr name="instanceCreate" methodvalue=
                "org.netbeans.modules.autoupdate.updateprovider.
                AutoupdateCatalogFactory.createUpdateProvider"/>
         <attr name="instanceOf"
                stringvalue="org.netbeans.spi.autoupdate.UpdateProvider"/>
         <attr name="url_key" stringvalue="my_module_update_center"/>
```

```
      </file>
   </folder>
</folder>
```

Data referenced in the registration entries is saved centrally in the resource bundle for localizing purposes:

```
my_module_update_center = http://heikoboeck.de/updates.xml
Services/AutoupdateType/my_module_update_center.instance = My Update Center
```

Automatically Installing Updates

An NBM file may also be installed without requiring user interaction. Put the NBM file in the `update/download` folder of a cluster. The update is installed when the application is next started, after which the NBM file is removed and the application is started afresh. Backups of original versions of the updated modules are found in the `update/backup/netbeans` folder. Remember, the update is always installed in the cluster where the update is stored, even if the module that should be updated is installed in a different cluster.

As of NetBeans 6.0, the Auto Update Services API and SPI are public and can be used by your own Platform Application to programmatically automate and adapt module installations, as well as the related update processes. More information on this NetBeans API can be found in the Javadoc.

Summary

This chapter has introduced you to the update facilities in the NetBeans Platform. Firstly, you saw how the Auto Update service works. An NBM file is an update package and can be created from your module by the NetBeans IDE. Next, you learned how to provide and configure update centers. You also saw how a module is configured to provide a localized version of an existing module. Finally, we dealt with the configuration of update centers on the client side and how updates can be installed automatically.

CHAPTER 13

Persistence
Let's Integrate Some Databases!

Most client applications, certainly most rich client applications, are able to persist business objects across restarts. In combination with other features of rich client applications on the NetBeans Platform, this chapter discusses client database solutions and the implementation of a bridge for the loosely coupled and transparent storage of business objects.

Java DB

Java DB is 100% Java and, as a result, is platform independent. For a complete DBMS, Java DB is very small, and since no special installation procedure is necessary, can be delivered as part and parcel of your application. Java DB is therefore ideal for usage within rich client applications. With Java 6, Sun Microsystems delivers it as the official JDK database solution. Additionally, the NetBeans IDE provides support in the form of tools for working with Java DB.

Integrating Java DB

You can find Java DB in the `db` subdirectory in your JDK 6 installation. You can also download the latest version from `http://developers.sun.com/javadb/downloads`. In the `lib` subdirectory, you will find `derby.jar`, which is the actual database system that makes the driver available (see Figure 13-1). In addition, there's `derbyclient.jar`, which you can use when Java DB is executed on a server and you do not want to deliver it together with your application.

In this chapter, we focus mainly on the client-related aspects of Java DB, as well as embedding it into your applications. Following the NetBeans Platform structure discussed earlier, it makes sense to deliver Java DB as an independent and separate module within your application.

To that end, create a library wrapper module. Go to File ➤ New Project, and then choose NetBeans Modules ➤ Library Wrapper Module. Next, browse to `derby.jar`. As the code name base, use `org.apache.derby`, and for the name, use `Java DB Embedded`. Then you must specify, in the module that accesses the database, a dependency on your new Java DB module. The database system will automatically be deployed at the first call to the JDBC driver.

Driver Registration

If you've ever worked with the JDBC API, the call to `Class.forName()` should be familiar to you. That's how the database driver for the database system you're using is indirectly loaded. That's also how the driver manager makes a connection to your database. With the JDBC API 4.0, which is part of Java 6, the `DriverManager` class was extended, such that database drivers can be loaded if they're registered in the `META-INF/services` folder.

In this way, you register the driver as an implementation of the `java.sql.Driver` interface. This has the advantage that the call to `Class.forName()` can fail and the driver can be loaded at the exact point needed. That's also how Java DB registers its required drivers. For our purposes, this means we can make a direct connection via the `DriverManager` to the database, without concern over the driver.

Creating and Using a Database

After you make Java DB available to your application as a library wrapper module and set a dependency on the library wrapper module from the module that needs it, you can simply access the DBMS and create your first database.

Java DB creates a separate folder for each database, with the same name as the database itself. These folders are stored in a system folder, which you must define next. An ideal place for this folder would be the application user directory, since that is where the NetBeans Platform also stores its user-specific settings. The path to this location is obtained via the system properties, and hence we need not care about cross-platform paths. This system folder is registered under the name `derby.system.home`:

```
System.setProperty("derby.system.home",
        System.getProperty("netbeans.user",
        System.getProperty("user.home")) + "/databases");
```

We assign the property `netbeans.user` to the path to the application-specific file. If it has not been set, we use the user directory obtained via the `user.home` property. By means of these lines, the database is created in the `databases` folder. If you have not set the `derby.system.home` property, Java DB will use the current working directory of the application.

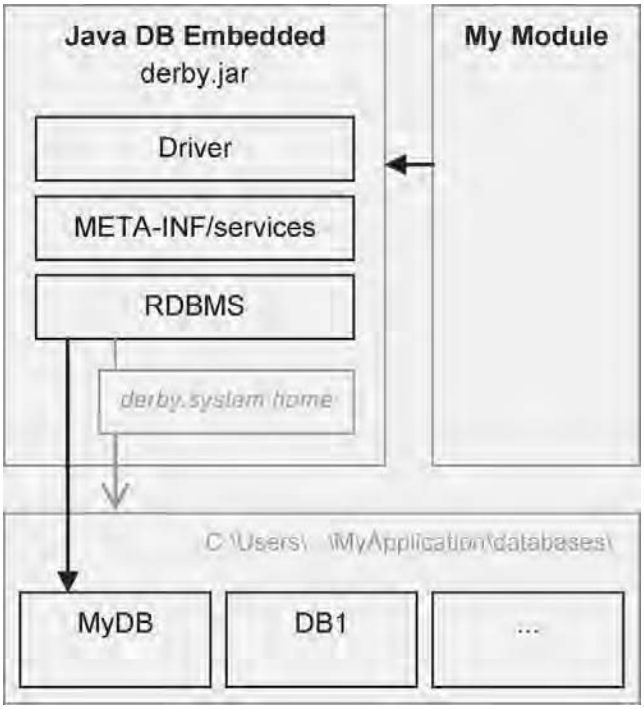


Figure 13-1. *Java DB as a library wrapper module for the integration of Java DB into your application. The physical storage place for databases is defined by the derby.system.home property.*

Very pragmatically, the database need not be started or kept running explicitly. Each database is started separately at the point it is first accessed. No extra administrative effort is required for control of the local databases, and you use them exactly as you would use databases on the server.

After you set up the system properties as described previously, make a connection to the database via the driver manager:

```
Connection connection =
   DriverManager.getConnection("jdbc:derby:MyDB;create=true", "user", "password");
```

But how to create a new database? That occurs through the declaration of the attribute `create=true`, passed via the connection URL, as shown in the preceding code. If the database `MyDB` is not available, it will be created. Then a connection to the database will be established. If the database is already available, only the connection is established. This attribute is particularly important in relation to the embedded, local use of Java DB, since it is used automatically when the application is initialized or first started.

Java DB defines a range of further attributes, none of which are particularly relevant here. Information about these attributes can be found in the Java DB Reference Manual, found with the other documents in the `docs` folder of your Java DB distribution.

Rather than putting the attribute directly in the URL, you may also save it in a `Properties` object and then pass it as a second parameter in the `getConnection()` method:

```
Properties props = new Properties();
props.put("user", "user");
props.put("password", "password");
props.put("create", "true");
Connection connection = DriverManager.getConnection("jdbc:derby:MyDB", props);
```

Shutting Down a Database

Starting a database occurs automatically when connecting for the first time. Shutting down happens in much the same way. Since the database has no way of knowing when the application exited, and hence ends abruptly, you must shut down the database explicitly when the application shuts down to ensure that the database shuts down in a consistent state. When the system shuts down, all active databases implicitly shut down as well. Optionally, you can shut down a specific database instead of all of them together.

The best approach for this task is to use the `ModuleInstall` class (see Chapter 3) or a `LifecycleManager` implementation (see Chapter 17). In a `ModuleInstall` class, use the `close()` method, defined as follows:

```
public class Installer extends ModuleInstall {
   public void close() {
      try {
         DriverManager.getConnection("jdbc:derby:;shutdown=true");
      } catch (SQLException ex) {}
   }
}
```

If you shut down the DBMS when the application closes, as in the preceding code, the complete Java DB system will shut down when the application closes. To shut down an individual database, simply specify the corresponding database name after `jdbc:derby:`. For example, when you wish to end the `MyDB` database, the command to shut down should be as follows: `DriverManager.getConnection("jdbc:derby:MyDB;shutdown=true");`. Note the fact that at shutdown (and also when you use `shutdown=true`), an exception is thrown. The exception simply passes information about the shutting down of the database.

Database Development with the Help of the NetBeans IDE

Simplifying the development of database-oriented applications, the NetBeans IDE provides tooling for Java DB integration. Using the IDE, you can start and stop a database. But you can also create a new database and make a connection to it. More than anything, the IDE simplifies development via its graphical support for the creation and configuration of tables. Hence, you can easily save your tables and their data types, and also change them without any problems.

Setting Up and Starting the Java DB System

To use the NetBeans IDE support described previously, first invoke Tools ➤ Java DB Database ➤ Settings, and specify the path to the Java DB installation. If you are using JDK 6, this path will already be pointing to the `db` subdirectory of the JDK installation. If you downloaded Java DB yourself, you can set the path to this location as well.

Further, you must specify a path where the database is created and stored. After you deal with these settings, start the database server by selecting Tools ➤ Java DB Database ➤ Start Server. That's especially necessary since we're not using the database as before (embedded within our application), but are controlling it as an independent server. The server usually accepts connections on port 1527, which is also shown in the Output window.

Integrating the Driver for the Java DB Server into Your Application

Since the Java DB database is not integrated into your application, but is being controlled as a server, you are advised to add a different driver to your application. This driver, which is necessary for establishing the connection to a Java DB server, is found in `derbyclient.jar`, as mentioned in the "Integrating Java DB" section. Add this to your application via a library wrapper module, and then set a dependency on that module from the module that requires a connection to the database.

Setting Up and Configuring the Database

At this point, your application has been set up and can access the Java DB server. Now go to Tools ➤ Java DB Database ➤ Create Database in the NetBeans IDE to create a new database. Specify a database name, a username, and a password for the database. After you provide this information, the database is established, as well as a connection to it. The connection is found in the Services window, opened via Window ➤ Services. There, you'll find all the established connections under the Databases menu (see Figure 13-2).

You can establish a connection via the context menu item Connect. If a connection is correctly established, the IDE shows the tables, indexes, and foreign keys of the database. New entries may be established via the context menu. For example, use the context menu of Tables to select Create Table and establish a new table. With View Data, you can see the content of a table, while with Execute Command, you send any SQL query to the database.

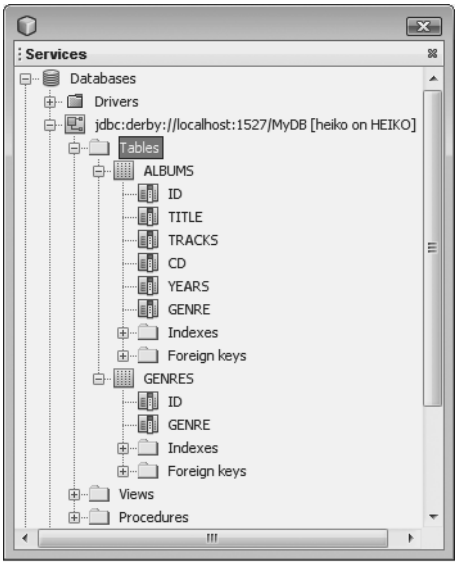


Figure 13-2. *Access to the Java DB database is established via the Services window, where configuration settings are also provided.*

Access to a Database from Your Application

Now let's take a look at the application from which you'd like to use a database. The definition of the property `derby.system.home`, which was necessary for the integration of Java DB with your application, is no longer needed. Optimally, to make a connection from your application to a Java DB server, tweak your connection URL. Specify the name (or the IP address), as well as the port on which the database server accepts connections:

```
Connection connection = DriverManager.getConnection(
        "jdbc:derby://localhost:1527/MyDB;",
        "user",
        "password");
```

Since the database server, in our case, is found on the same computer where the application is running, we use the setting localhost or the IP address 127.0.0.1. We also specify port 1527. You can also take this URL from the connection information in the IDE's Services window.

Creating and Importing Table Structures

Finally in this discussion, let's look at an extremely useful feature of the Database Explorer in the Services window. You can select the SQL source text with which a table is created and copy it into your application for initial establishment of database tables or, alternatively, into a SQL script file. For this purpose, call up the context menu for the desired table. There, you find the menu item Grab Structure. Click this menu item to store the structure of your data. After that, simply choose the context menu item Recreate Table, where you then select the data that you

just created. A window shows the SQL source needed to create a table. You can of course use this functionality in the way intended and import tables from foreign databases into your own.

Example Application

With a very simple example, we'll round off this section and in the process show you some characteristics of Java DB for creating table structures, also illustrating integration of Java DB into your application's lifecycle. In this example, we'll manage music albums, assigning them to various genres.

Configuration, Access, and Shutdown

With a module installer (also see Chapter 3), we optimally configure the Java DB database system, access it centrally, and, at the appropriate time, shut down the system (see Listing 13-1). First, use the `restored()` method, which is called at the start of the application via the `derby.system.home` property, to specify the database storage path. That will be within the `databases` folder in an application-specific user directory.

Next, call the method `initTables()`, using a `SELECT` statement to test whether the required tables already exist. When the application is started for the first time, the tables do not yet exist and a `SQLException` is thrown. Catch the exception and then create both the `albums` and `genres` tables.

First create the `genres` table, since the `albums` table will depend on it. Each entry in the table has a unique ID, which is assigned incrementally by the database. We achieve this using the command `GENERATED ALWAYS AS IDENTITY` for the ID. As a result, even when a value is specified for the physical ID when an entry is added to the table, an automatically generated value is used instead.

Alternatively, replace `ALWAYS` with `BY DEFAULT`, which means the value will be generated only if the ID is not defined explicitly. Using `PRIMARY KEY`, define the physical ID as the primary key, where connection to the entries in the `albums` table will be established. We then create this immediately and define, in the same way, the physical key ID. Further columns are `title`, `tracks`, `cds`, `years`, and `genre`.

Finally, don't write a genre directly. Instead, write the ID of a genre entry from the `genres` table. The `genre` column in `albums` is thus a foreign key. We define this via the `FOREIGN KEY (genre)` command, and connect via `REFERENCES genres (id)` to the `id` column in the `genres` table (see Figure 13-2). To allow user genre selection when creating an album, we provide three example entries in the `genres` table.

Listing 13-1. *Setting up the database system and initializing the database*

```
public class Installer extends ModuleInstall {
   private static Connection conn = null;
   public void restored() {
      System.setProperty("derby.system.home",
         System.getProperty("netbeans.user",
         System.getProperty("user.home")) + "/databases");
      initTables();
   }
   private void initTables() {
      try {
```

```
            Statement stmt = getConnection().createStatement();
            stmt.executeQuery("SELECT id FROM genres");
            stmt.close();
         } catch(SQLException e) {
            try {
               Statement stmt = getConnection().createStatement();
               stmt.execute("CREATE TABLE genres (" +
                  "id INTEGER GENERATED ALWAYS AS IDENTITY, " +
                  "genre VARCHAR(100), " +
                  "PRIMARY KEY(id))");
               stmt.execute("CREATE TABLE albums (" +
                  "id INTEGER GENERATED ALWAYS AS IDENTITY, " +
                  "title  VARCHAR(100), " +
                  "tracks VARCHAR(10), " +
                  "cds    VARCHAR(10), " +
                  "years  VARCHAR(10), " +
                  "genre  INTEGER, " +
                  "PRIMARY KEY(id), " +
                  "FOREIGN KEY(genre) REFERENCES genres (id))");
               stmt.execute("INSERT INTO genres (genre) VALUES('Trance & Dance')");
               stmt.execute("INSERT INTO genres (genre) VALUES('Rock & Pop')");
               stmt.execute("INSERT INTO genres (genre) VALUES('Country & Classic')");
               stmt.close();
            }
            catch(SQLException ex) {
               ex.printStackTrace();
            }
         }
      }
   }
```

Setting up the database system and initializing the database using the static method `getConnection()` provides central access to the database. Thus, the client need not worry about the connection URL. Also, the `Connection` object, with its connection to the database, is centralized and need not be stopped and started whenever a connection is made. `getConnection()` is, therefore, a factory method that creates a connection when none exists or one has been closed (see Listing 13-2). It then returns the `Connection` object. The `close()` method informs us when the application exits. We close the current connection in this method. We use `jdbc:derby:;shutdown=true` to close the whole Java DB system and also, automatically, our `MyDB` database.

Listing 13-2. *Centralized creation of the connection and shutting down of the database*

```
public static Connection getConnection() throws SQLException {
   if(conn == null || conn.isClosed()) {
      conn = DriverManager.getConnection(
         "jdbc:derby:MyDB;create=true",
         "user", "password");
   }
   return conn;
}
public void close() {
```

```
    try {
      conn.close();
      DriverManager.getConnection("jdbc:derby:;shutdown=true");
    } catch (SQLException ex) {}
}
```

Data Models and Data Access Models

As shown before with the creation of tables, we want to be able to choose two different classes of data. First, we have the albums, information for which can be chosen from the `albums` table; and second, we have the genres, found within the `genres` table. To assist us, we create a data model (see Listing 13-3 and Listing 13-4). The data model consists of the classes `Album` and `Genre`, which provide relevant setters and getters. Note that no persistence logic is found in these classes. We will put this logic in a separate class.

Listing 13-3. *Data model for an album*

```
public class Album {
   private int id = 0;
   private String title;
   private String tracks;
   private String cds;
   private String year;
   private Genre genre;
   public Album(
      int id, String title, String tracks, String cds, String year, Genre genre) {
      this.id = id;
      this.title = title;
      this.tracks = tracks;
      this.cds = cds;
      this.year = year;
      this.genre = genre;
   }
   public int getId() {
      return id;
   }
   public String getTitle() {
      return title;
   }
   ...
}
```

The data model for an `Album` in the `Genre` class requires overwriting both the `toString()` and `equals()` methods. This is necessary for the correct representation and selection of a genre in the dialog for creating a new album.

Listing 13-4. *Data model for a genre*

```
public class Genre {
   private int id = 0;
   private String genre;
```

```
   public Genre(int id, String genre) {
      this.id = id;
      this.genre = genre;
   }
   public int getId() {
      return id;
   }
   public String getGenre() {
      return genre;
   }
   public String toString() {
      return genre;
   }
   public boolean equals(Object obj) {
      if(obj instanceof Genre) {
         if(((Genre)obj).getId() == id) {
            return true;
         }
      }
      return false;
   }
}
```

To let the data model and business logic (which in this case is primarily the user interface for selecting data) be loosely coupled to the persistence layer, we encapsulate access to the database and SQL statements in a separate class named `DataModel` (see Listing 13-5). This class performs desired changes and requests to the database, while also providing the data via the `Album` and `Genre` data models.

The methods `getAlbums()` and `getGenres()`, implemented in the `DataModel` class, provide vectors containing the chosen data. We also have the methods `insertAlbum()`, `updateAlbum()`, and `deleteAlbum()`, with which we enter albums into the database and also use for changing and deleting them.

Listing 13-5. *The DataModel class encapsulates the access to Java DB and makes data available via the related Album and Genre data models.*

```
public class DataModel {
   public static Vector<Album> getAlbums() {
      Vector<Album> albums = new Vector<Album>();
      try {
         Statement stmt = Installer.getConnection().createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM albums"+
            " INNER JOIN genres ON albums.genre = genres.id");
         while(rs.next()) {
            albums.add(new Album(rs.getInt(1), rs.getString(2),
               rs.getString(3), rs.getString(4), rs.getString(5),
```

```
            new Genre(rs.getInt(7), rs.getString(8))));
      }
      rs.close();
      stmt.close();
   } catch(SQLException e) {
      Exceptions.printStackTrace(e);
   }
   return albums;
}
public static Vector<Genre> getGenres() {
   Vector<Genre> genres = new Vector<Genre>();
   try {
      Statement stmt = Installer.getConnection().createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM genres");
      while(rs.next()) {
         genres.add(new Genre(rs.getInt(1), rs.getString(2)));
      }
      rs.close();
      stmt.close();
   } catch(Exception e) {
      e.printStackTrace();
   }
   return genres;
}
public static void updateAlbum(Album a) throws SQLException {
   PreparedStatement stmt = Installer.getConnection().prepareStatement(
      "UPDATE albums SET title=?, tracks=?, cds=?, years=?, genre=? WHERE id=?");
   stmt.setString(1, a.getTitle());
   stmt.setString(2, a.getTracks());
   stmt.setString(3, a.getCDs());
   stmt.setString(4, a.getYear());
   stmt.setInt(5, a.getGenre().getId());
   stmt.setInt(6, a.getId());
   stmt.execute();
}
public static void insertAlbum(Album a) throws SQLException {
   PreparedStatement stmt =Installer.getConnection().prepareStatement(
      "INSERT INTO albums (title, tracks, cds, years, genre) VALUES(?,?,?,?,?)");
   stmt.setString(1, a.getTitle());
   stmt.setString(2, a.getTracks());
   stmt.setString(3, a.getCDs());
   stmt.setString(4, a.getYear());
   stmt.setInt(5, a.getGenre().getId());
   stmt.execute();
}
public static void deleteAlbum(Album a) throws SQLException {
   PreparedStatement stmt = Installer.getConnection().prepareStatement(
         "DELETE FROM albums WHERE id = ?");
```

```
      stmt.setInt(1, a.getId());
      stmt.execute();
   }
}
```

Displaying and Working with the Data

We now come to components that will display the data, allowing the user to create and edit music albums. We'll list the albums in a table within a `TopComponent` (see Figure 13-3). We begin by creating the `AlbumsTopComponent` class, containing a `JTable`. To enable the table to display the `DataModel` of our album, we need a model for the table.

Since the `DataModel` is only available to this class, we implement it as a private inner class named `AlbumTableModel` (see Listing 13-6). The data is obtained from a vector of the type `Album`. Since we later need access to the model, we create it as a private data element. We connect the `DataModel` with the table via the `setModel()` method. Typically, table entries can be edited or viewed via a double-click of the mouse. To create this functionality, we register a `MouseListener` or a `MouseAdapter` with the `JTable`, which calls the `editAlbumActionPerformed()` method on double-click. This will be discussed next.

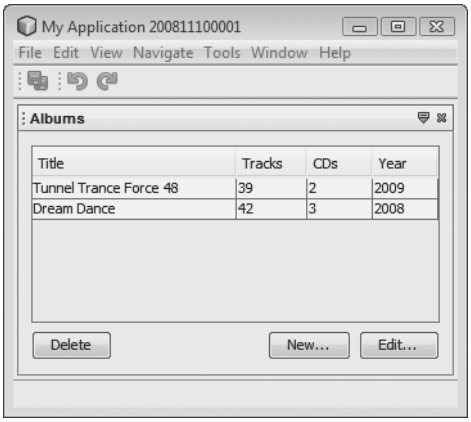


Figure 13-3. *Displaying the database entries in a table*

Listing 13-6. *TopComponent implementation with AlbumTableModel*

```
final class AlbumsTopComponent extends TopComponent {
   private JTable albums;
   private AlbumTableModel model = new AlbumTableModel();
   private AlbumsTopComponent() {
      initComponents();
      albums.setModel(model);
      albums.addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent event) {
            if(event.getClickCount() == 2) {
```

```
                editAlbumActionPerformed(null);
            }
        }
    });
}
private static final class AlbumTableModel
    extends AbstractTableModel {
    private String[] columns = {"Title", "Tracks", "CDs", "Year"};
    private Vector<Album> data = new Vector<Album>();
    public Album getRow(int row) {
        return data.get(row);
    }
    public int getRowCount() {
        return data.size();
    }
    public int getColumnCount() {
        return columns.length;
    }
    public String getColumnName(int col) {
        return columns[col];
    }
    public Object getValueAt(int row, int col) {
        Album album = data.get(row);
        switch(col) {
            case 0: return album.getTitle();
            case 1: return album.getTracks();
            case 2: return album.getCDs();
            case 3: return album.getYear();
        }
        return "";
    }
    public Vector<Album> getData() {
        return data;
    }
}
```

As the `TopComponent` opens, we need to load and display the current entries from the database. For this reason, we override the method `componentOpened()`, where we use our data access model `DataModel`, which abstracts access to the database to obtain all entries in the database, via the `getAlbums()` method. We add these to the `DataModel` in the table and inform the view, which is the `JTable`, via the `fireTableDataChanged()` method, that the data has changed.

Finally, we implement three action methods that enable the user to add, edit, and delete entries. For the creation of new albums, we have the `newAlbumActionPerformed()` method. We use it to call a static method that opens a dialog where the user can enter the required data. We create this dialog in the final step. If the method returns an `Album` instance, the dialog is immediately closed and the data is added to the database. If that code can be run without an exception being thrown, we add the album to the table.

```
public void componentOpened() {
   model.getData().addAll(DataModel.getAlbums());
   model.fireTableDataChanged();
}
private void newAlbumActionPerformed(ActionEvent evt) {
   Album album = AlbumEditDialog.newAlbum();
   if(album != null) {
      try {
         DataModel.insertAlbum(album);
         model.getData().add(album);
         model.fireTableDataChanged();
      } catch(SQLException e) {
         Exceptions.printStackTrace(e);
      }
   }
}
```

The method `editAlbumActionPerformed()` is invoked by means of the Edit button or by a double-click. Similar to the way new entries are created, we again call up a dialog. However, we need `editAlbum()` for that purpose, to which we pass an `Album` instance, allowing data to be edited in the dialog. The currently selected row in the table invokes the `getSelectedRow()` method, with the returned value allowing related data to be found in the `JTable`'s data model.

The user can now change the data. If the OK button is clicked, the `editAlbum()` method is called, which returns the changed `Album` instance (see Listing 13-7). The changes are saved in the database with the `updateAlbum()` method.

Finally, we need to address situations where the user deletes an entry from the database. That will be handled by the `deleteAlbumActionPerformed()` method. To prevent unintended deletion, the user is asked to confirm that the entry should be deleted. The dialog that is required for this functionality is created in a very simple way, via the NetBeans Dialogs API (see Chapter 8). We use the `NotifyDescriptor.Confirmation` instance. We show the dialog via the `notify()` method. Once the user has confirmed the deletion request, the entry is removed from the database via the `deleteAlbum()` method. Only when the operation can be completed successfully do we delete the album from the `JTable` and update with its current entries.

Listing 13-7. *TopComponent for displaying and working with the albums in the database*

```
private void editAlbumActionPerformed(ActionEvent evt) {
   Album album = AlbumEditDialog.editAlbum(
           model.getRow(albums.getSelectedRow()));
   if(album != null) {
      try {
         DataModel.updateAlbum(album);
         model.fireTableDataChanged();
      } catch(SQLException e) {
         Exceptions.printStackTrace(e);
      }
   }
}
```

```
    private void deleteAlbumActionPerformed(ActionEvent evt) {
        Album album = model.getRow(albums.getSelectedRow());
        NotifyDescriptor d = new NotifyDescriptor.Confirmation(
            "Are you sure you want delete the album " + album.getTitle(),
            "Confirm Album Deletion");
        if(DialogDisplayer.getDefault().notify(d) == NotifyDescriptor.YES_OPTION) {
            try {
                DataModel.deleteAlbum(album);
                model.getData().remove(album);
                model.fireTableDataChanged();
            } catch(SQLException e) {
                Exceptions.printStackTrace(e);
            }
        }
    }
}
```

Our last task is the creation of a dialog with which the data can be created and edited. Again, we need to use the classes of the Dialogs API; so we need not create a complete dialog of our own, but simply the panel with required fields (see Figure 13-4). We therefore create a simple `JPanel` class, via File ➤ New File ➤ Java GUI Forms ➤ JPanel Form.

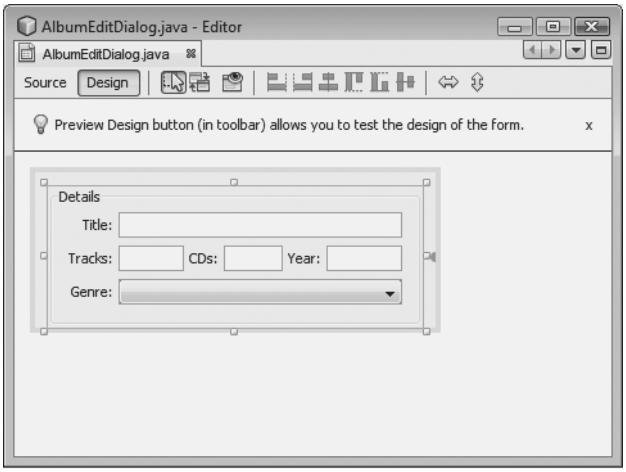


Figure 13-4. *Dialog for working with entries*

In the constructor of the panel, we load all the genres from the database and add them to the combo box. Additionally, we require the methods `newAlbum()` and `editAlbum()`, which you were introduced to in the previous section. To simplify things, implement these as static methods (see Listing 13-8). These methods are therefore factories that are concerned with the creation of the dialog. First, create an instance of the `AlbumEditDialog` class. Create a dialog

with the help of a `DialogDescriptor`, pass the recently created panel, and that's everything needed for creating a dialog. As per usual, we show the dialog via the `notify()` method.

As soon as the user clicks the OK button, we use the data to create an `Album` object and pass it back to the user; otherwise, we simply return `null` and indicate an error. In the case of the `editAlbum()` method, we take the same approach with creating the dialog. Simply fill the fields with the values of the selected album. However, when the dialog is completed, don't create a new `Album` object; simply update the data via the relevant setters and pass the updated instance back to the user.

Listing 13-8. *Dialog for editing and creating new music albums*

```
public class AlbumEditDialog extends Jpanel {
   private AlbumEditDialog() {
      initComponents();
      for(Genre g : DataModel.getGenres()) {
         genre.addItem(g);
      }
   }
   public static Album newAlbum() {
      AlbumEditDialog d = new AlbumEditDialog();
      DialogDescriptor desc = new DialogDescriptor(d, "New...");
      if(DialogDisplayer.getDefault().notify(desc) == DialogDescriptor.OK_OPTION) {
         Album album = new Album(0,
            d.title.getText(),
            d.tracks.getText(),
            d.cds.getText(),
            d.year.getText(),
            (Genre)d.genre.getModel().getSelectedItem());
         return album;
      } else {
         return null;
      }
   }
   public static Album editAlbum(Album album) {
      AlbumEditDialog d = new AlbumEditDialog();
      d.title.setText(album.getTitle());
      d.tracks.setText(album.getTracks());
      d.cds.setText(album.getCDs());
      d.year.setText(album.getYear());
      d.genre.getModel().setSelectedItem(album.getGenre());
      DialogDescriptor desc = new DialogDescriptor(d, "Edit...");
      if(DialogDisplayer.getDefault().notify(desc) == DialogDescriptor.OK_OPTION) {
         album.setTitle(d.title.getText());
         album.setTracks(d.tracks.getText());
         album.setCDs(d.cds.getText());
         album.setYear(d.year.getText());
         album.setGenre((Genre)d.genre.getModel().getSelectedItem());
         return album;
      } else {
         return null;
      }
```

```
    }
}
```

At this point, we have explained everything relating to data access and display of data found within Java DB databases. We also looked at an example application, showing how to set up and use Java DB within the NetBeans Platform.

Hibernate

In the last section, you became familiar with tasks pertaining to client database solutions in the context of a rich client application. We decomposed our Java objects to save them in our database. Likewise, we adopted a conventional approach in using SQL to extract data via JDBC interfaces from the database. And from the data we then constructed Java objects, which were named `Album` and `Genre`. Encapsulating that functionality in our `DataModel` class, you probably noticed that this might ultimately lead to a very complex and error-prone result, which we'd prefer to avoid.

These are some of the reasons why database developers created and standardized object-oriented databases. However, for a long time this approach was unable to make a dent in the success of the many existing relational systems. More than anything, that was because of the wide adoption of RDBMSs, which resulted in new applications being able to provide access to data stored in these kinds of systems.

For these reasons, object-relational bridges were developed. Their central focus is saving and loading objects to and from relational databases, thereby providing an abstraction layer over the database beneath. The most well-known and widely used implementation of such a bridge is provided by Hibernate. Hibernate handles objects and their relationships, which must be established and maintained as transparently as possible. Ideally, we needn't care where or how our business objects are stored.

This section focuses on a useful example of Hibernate integration within a rich client application based on the NetBeans Platform. We only discuss the most basic principles of Hibernate, since we're concerned specifically with its relation to the NetBeans Platform, not with the details of Hibernate's many features.

Setting Up the Hibernate Libraries

First, download the current version of the Hibernate Core distribution from `http://hibernate.org`. Along with the Hibernate library, the distribution provides examples, as well as a complete documentation set and required libraries provided by third-party vendors.

As in the previous section, begin by encapsulating the libraries as an independent NetBeans module. Go to File ➤ New Project, and then NetBeans Modules ➤ Library Wrapper Module. Now add all of the following libraries from Hibernate's `lib` folder to the module:

- `hibernate3.jar`
- `lib/antlr-x.x.x.jar`
- `lib/asm.jar`

- lib/asm-attrs.jar
- lib/cglib-x.x.x.jar
- lib/commons-collections-x.x.x.jar
- lib/commons-logging-x.x.x.jar
- lib/dom4j-x.x.x.jar
- lib/jta.jar
- lib/log4j-x.x.x.jar

The libraries listed here are the minimum required to start Hibernate. For various reasons, another set of libraries are also at your disposal. Further information about these can be found in the README.txt file.

Typically, any problems arising when using libraries on the NetBeans Platform relate to class loading. Unfortunately, this is also the case with Hibernate. The Bytecode enhancer of the CGLIB library looks for its objects on the standard classpath, which is the application classpath. The enhancer is necessary for the proxy creation of objects required for lazy-loading. The application classpath simply contains a number of standard Java libraries, but not our Hibernate library, nor the classes of our application module (further information about the NetBeans Platform classloader system can be found in Chapter 2).

There's nothing we can do other than ensure that the enhancer is used from the module classloader, rather than from the application classloader. That way, the enhancer is able to access the libraries in the Hibernate module, as well as the classes in the modules where dependencies are defined.

To that end, we need to tweak the Hibernate source code slightly. The source code is already available in your environment, within the src folder. Look in the appropriate location for the org.hibernate.proxy.pojo.cglib.CGLIBLazyInitializer.java class. Add the following statement in the method getProxyFactory(), after the creation of the enhancer:

```
e.setClassLoader(CGLIBLazyInitializer.class.getClassLoader());
```

The CGLIBLazyInitializer class is loaded by the module classloader of the Hibernate module. We obtain the classloader via the getClassLoader() method, which we pass to the enhancer object. This lets the enhancer access the required classes. After that, create the package org.hibernate.proxy.pojo.cglib in the Hibernate wrapper module and add the changed class there, so that it will be loaded instead of the original class.

Structure of the Example Application

You will learn to use Hibernate by an example based on the albums used in the "Example Application" section earlier in the chapter. Along the way, the advantages of object-relational bridges will immediately become obvious, and there will be two different approaches for comparison. In addition, we've already bundled the Java DB database system, which now needs to be used again.

Therefore, add the Hibernate module to the application created earlier. Each module that needs to use the Hibernate functionality can simply declare its own dependency. In our example, the module that declares this dependency is simply our only application module (My Module). Open the Properties window via the context menu and add the Hibernate module

under Libraries as a new dependency. Because Hibernate can access the Java DB database driver, add the same dependency from the `Hibernate` module to the `Java DB` module.

A further dependency must be defined between the application module `My Module` and the `Hibernate` module. That enables us to access the functionality made available by Hibernate.

Till now, we've used the application module to work with the `Genre` and `Album` entities. Since Hibernate accesses the classes, we set a dependency to the `Hibernate` module from the application module, resulting in an undesirable cyclic dependency. The NetBeans Platform runtime container then quickly throws an error and the application is unable to start.

For this reason, let's take our entity classes and put them into a module of their own, which solves the problem of cyclic dependencies. The architecture with the additional module (`My Entities`) is shown in Figure 13-5.

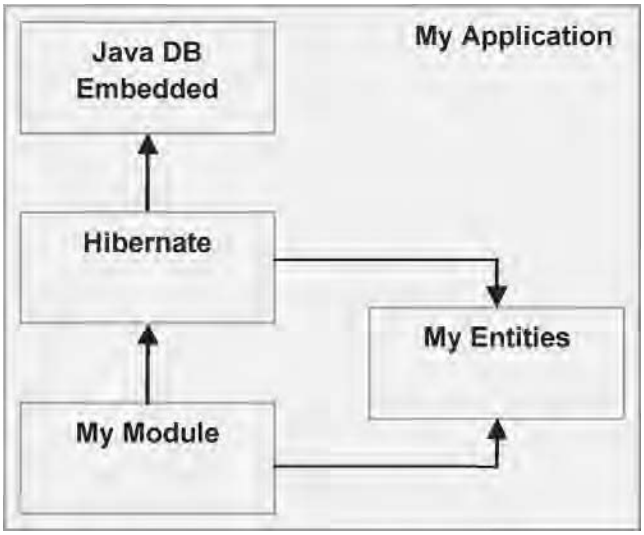


Figure 13-5. *The application components and your dependencies*

■ **Tip** When developing your application, it is highly recommended to not immediately use the embedded version of Java DB (in other words, our `Java DB Embedded` module). It's better to use the server variation, which is integrated into the NetBeans IDE. Include the Java DB driver in your application, as explained in the "Database Development with the Help of the NetBeans IDE" section earlier in the chapter. Then adapt your `hibernate.cfg.xml` file to make the URL connection and start the database server in the NetBeans IDE via Tools ➤ Java DB Database ➤ Start Server. This way, you use the Services window, which opens via Window ➤ Services, to view the database schema created by Hibernate.

Configuring Hibernate

Now that Hibernate is bundled with our application, we can set up a configuration dependency. Do this via an XML file, which is named `hibernate.cfg.xml` by default. There, we define

the database driver, the connection URL, the authentication data, and the applicable SQL dialect. The configuration data will look as shown in Listing 13-9.

Listing 13-9. *Hibernate configuration data*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="connection.driver_class">
         org.apache.derby.jdbc.EmbeddedDriver</property>
      <property name="connection.url">jdbc:derby:hibernate-db;create=true</property>
      <property name="connection.username">user</property>
      <property name="connection.password">password</property>
      <property name="dialect">org.hibernate.dialect.DerbyDialect</property>
   </session-factory>
</hibernate-configuration>
```

Next, define the database driver class, which in this case is for the Java DB database system, requiring the class `org.apache.derby.jdbc.EmbeddedDriver`. Since Hibernate creates a connection to the database, we also provide the connection URL.

The "Creating and Using a Database" section at the beginning of the chapter described putting together this information for Java DB (aside from the username and the password, which may be required for the database). We also must define the SQL dialect. For all the popular databases, Hibernate provides the package `org.hibernate.dialect`, containing classes that can be used to define the applicable dialect. The Hibernate reference documentation should help you with any additional configuration settings that might need to be set.

Where then to place these files? Since the data is located in a separate file, we are able to provide different Hibernate modules with different data. We can either put the file into the `src` folder of an application or directly within a Hibernate wrapper module. Importantly, you should place it in the appropriate classpath, since that is by default where the file is searched. You also have a choice at the time the `Configuration` object is created to provide an alternative URL or configuration file.

When the required libraries are added to the Hibernate wrapper module, you may notice that Hibernate uses Hibernate Log4J. To ensure support by useful log messages, add a configuration file for Log4J. Use the file `log4j.properties` for this purpose, from the `etc` folder in your Hibernate distribution. Then add the `src` folder to the `Hibernate` module.

Mapping Objects to Relations

Now that Hibernate is ready to go, we ask, "How exactly will Hibernate save our object into the database?" In addition, how is the object structure mapped to a relation? The answer is to create a map file for each object that we'd like to persist. That is where mapping information will be stored. For example, this file includes information about the name and type under which the attribute will be saved.

Even more importantly, the file defines relationships between different objects. Aside from the complexity of the object structure, there are many possible entries, some of which are

mandatory to define. In this discussion, we obviously cannot delve into too much detail, and we must limit ourselves to mapping information relevant to this section's example application.

Have a look at a map file for our example class `Genre` (see Listing 13-10). Remember, we defined a number as the unique ID and a string as the genre's name.

Listing 13-10. *Object-relational mapping for the Genre class: Genre.hbm.xml*

```
<!DOCTYPE hibernate-mapping PUBLIC
   "//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.galileo.netbeans.myentities">
   <class name="Genre" table="Genre" lazy="true">
      <id name="id">
         <generator class="increment"/>
      </id>
      <property name="genre" not-null="true" length="30" column="genre"/>
   </class>
</hibernate-mapping>
```

We use the `class` element to specify the class name and the name of the table where the data of the type `Genre` is stored. Using the `id` element for the object attribute of the same name, the primary key of the table is defined. The primary key will increment whenever needed.

Thus, we only need to define the second and last `genre` attribute via the `property` element. Now when looking at the map file for the `Album` class (see Listing 13-11), things become quite interesting, because a genre can be assigned to an album. However, multiple albums can use the same genre. That means a many-to-one relationship exists. Define that relationship via the element of the same name and set the `lazy` attribute to `false`.

By this, we ensure that the `Genre` object will load immediately, together with the `Album`, and not afterward or separately. Via the assignment `fetch="join"`, Hibernate connects the `Genre` simultaneously with the query for the album data, via a `JOIN` query. This is, in fact, a *request optimization*, since this ensures that only one request is needed instead of two, when loading the class from the database.

Listing 13-11. *We define the genre with a many-to-one relationship.*

```
<!DOCTYPE hibernate-mapping PUBLIC
   "//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.galileo.netbeans.myentities">
   <class name="Album" table="Album" lazy="true">
      <id name="id">
         <generator class="increment"/>
      </id>
      <many-to-one name="genre" lazy="false" fetch="join"/>
      <property name="title"
                not-null="true"
                length="30"
                column="title"/>
```

Now, put the map files into the same package as the classes. Typically, the map files end in `hbm.xml`. Next, make them known to Hibernate. This occurs via an entry in the `hibernate.cfg.xml` configuration file, where we have already defined our database settings.

Using the element mapping, connect all the data. In addition, define the property `hbm2ddl`, using the value `update` (see Listing 13-12). That is how Hibernate obtains, when the application starts, the database schema automatically. It does so via the information that you set in the map files, so long as the database schema is not already available.

Listing 13-12. *Registering the map files in the configuration file*

```
<hibernate-configuration>
   <session-factory>
      ...
      <property name="hbm2ddl.auto">update</property>
      <mapping resource="com/galileo/netbeans/myentities/Genre.hbm.xml"/>
      <mapping resource="com/galileo/netbeans/myentities/Album.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

SessionFactory and Sessions

At this stage, we've completed the configuration tasks. Now we'll try to connect to Hibernate. With that in mind, first create an object of the `Configuration` class, administering the configuration defined earlier in the `hibernate.cfg.xml` file. By default, the information for this class extracts from the `hibernate.properties` file.

However, our information is organized in an XML file. For that reason, we prompt the `Configuration` instance, via its `configure()` method, to find the data in question. The method is also available in a parameterized form, allowing you to pass in either a `File` or a `URL` pointing to the configuration data. In this case, we'll use the parameterless version, which immediately expects the configuration to be available via the name `hibernate.cfg.xml` on the classpath.

A `Configuration` object is normally created only once. Based on this configuration, create a `SessionFactory`, via the method `buildSessionFactory()`. Similar to the `Configuration` object, a `SessionFactory` is kept alive over the duration of the application lifecycle.

That means that we must set up both the `Configuration` and the `SessionFactory` instances within the same location in the module. Use the `ModuleInstall` class for that purpose (see Chapter 3). In this way, a static instance of a `SessionFactory` is created (see Listing 13-13).

Listing 13-13. *The central administration and preparation of a SessionFactory*

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class Installer extends ModuleInstall {
   private static final SessionFactory sessionFactory;
   static {
      try {
         sessionFactory = new Configuration().configure().buildSessionFactory();
```

```
      } catch (Throwable ex) {
         throw new ExceptionInInitializerError(ex);
      }
   }
   public static Session createSession() {
      return sessionFactory.openSession();
   }
   public static Session currentSession() {
      return sessionFactory.getCurrentSession();
   }
   @Override
   public void close() {
      sessionFactory.close();
   }
}
```

Access to the database takes place via sessions. A session is a short-lived object, responsible for the interaction between application and database. Hiding behind a session is a JDBC connection. Thus, a `Session` object is also responsible for the creation of `Transaction`s, containing a cache for that purpose.

Using the `openSession()` method, create a new `Session` instance. Via the `getCurrentSession()` method, we receive the current `Session`. If none is available, a new one is created and connected to the current thread. If a transaction that is created by the current session ends (via `commit()` or `rollback()`), the session is automatically closed. This usage of sessions is the simplest and most comfortable approach to adopt. It is also highly recommended for integration into your own application.

Saving and Loading Objects

Using the `Installer` class, we obtain a simple helper for saving and loading objects. We would like to use these objects right away. You'll remember that we created the `DataModel` to take responsibility for interacting with the database. We also created it to be a mediator between SQL and our objects.

All that functionality is now handled by Hibernate. However, we'd like to use the class later on, without changing the rest of our application—and we have encapsulated Hibernate. Now we introduce the interesting part (see Listing 13-14), since this is where the simplification takes place.

Listing 13-14. *The DataModel class integrates with the database via Hibernate.*

```
import org.hibernate.Session;
import org.hibernate.Transaction;
public class DataModel {
   public static List<Album> getAlbums() {
      Session s = Installer.currentSession();
      Transaction t = s.beginTransaction();
      List<Album> list = (List<Album>)s.createCriteria(Album.class).list();
```

```
        t.commit();
        return list;
    }
    public static List<Genre> getGenres() {
        Session s = Installer.currentSession();
        Transaction t = s.beginTransaction();
        List<Genre> list = (List<Genre>)s.createCriteria(Genre.class).list();
        t.commit();
        return list;
    }
    public static void updateAlbum(Album album) {
        Session s = Installer.currentSession();
        Transaction t = s.beginTransaction();
        s.update(album);
        t.commit();
    }
    public static void insertAlbum(Album album) {
        Session s = Installer.currentSession();
        Transaction t = s.beginTransaction();
        s.save(album);
        t.commit();
    }
    public static void deleteAlbum(Album album) {
        Session s = Installer.currentSession();
        Transaction t = s.beginTransaction();
        s.delete(album);
        t.commit();
    }
}
```

For each action, we provide an updated or a new `Session`. By this approach, we continually create another `Transaction`. Then we use the `Session` to execute a specific action (request, store, update, etc.), after which we close the `Transaction` via the `commit()` method. As pointed out, it isn't necessary to explicitly close the `Session`, since it automatically closes when the `Transaction` ends.

In this example, we've obviously dealt with an extremely simple scenario. As a result, we've dealt with actions over a session and a transaction that have durations of equal length. Read the Hibernate documentation for information about the correct level of granularity in using sessions and transactions.

And with that, we've migrated our example (which first used SQL over a JDBC interface to crudely transfer data from application to database) completely to Hibernate. The application now runs entirely on top of the underlying persistence layer, and the application developer can neatly and transparently store objects in and request objects from the database.

Java Persistence API

The aim of the Java Persistence API (JPA) is to specify a standardized, simple, and useful persistence model. The persistence model should be relevant to the JSE world, as well as to that of JEE. In that light, the best ideas have come from the Hibernate, TopLink, and JDO communities. As a result, applications that make use of the JPA interfaces are completely independent of any specific framework, such as Hibernate itself. Their use encompasses the same independence as you may have experienced with the JDBC interfaces.

JPA distinguishes itself, more than anything else, by the extent to which it is lightweight. Its main characteristics include the ability to specify object relations, via Java annotations, directly within the persistence object. This entails a separate map file, similar to the one shown in the previous section on Hibernate. Unsurprisingly, there are no explicit relations between object structures and their relations. Instead, annotations are used to influence the default mappings between objects. This enormously simplifies the definition of entities.

Furthermore, JPA specifies SQL-like queries in the form of Java Persistence Query Language (JPQL), for both static as well as dynamic queries. This makes your queries independent of proprietary queries, such as those provided by HQL. JPA's persistence encompasses three different aspects: the API itself, which is in the package `javax.persistence`; the query language JPQL; and the annotations used for the definition of relational information.

Over time, a range of projects and frameworks have provided their own JPA implementations. Among these are, of course, Hibernate and TopLink, as well as GlassFish and OpenJPA. Since you have already been introduced to the native Hibernate interfaces, we now turn to its JPA interfaces, which are not all that different from Hibernate's native interfaces. Here, you can see very clearly how closely Hibernate implements the JPA Specification.

Hibernate and the Java Persistence API

To be able to use JPA from Hibernate, we first need two further packages from the `http://hibernate.org` web site. These are the packages `Hibernate Annotations` and `Hibernate EntityManager`. Their libraries need be added to our Hibernate library wrapper module. Unfortunately, it isn't possible to add them to a preexisting library wrapper module. We can only add them manually, and therefore it is simpler to create a new `Hibernate` module from scratch. The libraries that we need are as follows:

- Hibernate Core:
  - hibernate3.jar
  - lib/antlr-x.x.x.jar
  - lib/asm.jar lib/asm-attrs.jar
  - lib/cglib-x.x.x.jar
  - lib/commons-collections-x.x.x.jar
  - lib/commons-logging-x.x.x.jar
  - lib/dom4j-x.x.x.jar
  - lib/javassist.jar lib/jta.jar
  - lib/log4j-x.x.x.jar
- Hibernate Annotations:
  - hibernate-annotations.jar
  - lib/hibernate-commons-annotations.jar
- Hibernate EntityManager:
  - hibernate-entitymanager.jar
  - lib/ejb3-persistence.jar
  - lib/hibernate-validator.jar
  - lib/jboss-archive-browsing.jar

Check to make sure that you have the required libraries by reading the file lib/README.txt. Depending on the version, this might have changed, so be very careful!

Java Persistence Configuration

Configuring persistence follows the native Hibernate approach very closely. The configuration is done in the persistence.xml file (see Listing 13-15), which is in the META-INF folder. In addition, the files are bundled in persistence units.

Listing 13-15. *Persistence configuration in the META-INF/persistence.xml file*

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
   <persistence-unit name="HibernateJPA" transaction-type="RESOURCE_LOCAL">
      <class>com.galileo.netbeans.myentities.Genre</class>
      <class>com.galileo.netbeans.myentities.Album</class>
      <properties>
         <property name="hibernate.connection.driver_class">
            org.apache.derby.jdbc.EmbeddedDriver</property>
         <property name="hibernate.connection.url">
```

```
            jdbc:derby:hibernatejpa-db;create=true</property>
         <property name="hibernate.connection.username">user</property>
         <property name="hibernate.connection.password">password</property>
         <property name="hibernate.dialect">
            org.hibernate.dialect.DerbyDialect</property>
         <property name="hibernate.hbm2ddl.auto">update</property>
      </properties>
   </persistence-unit>
</persistence>
```

Create a persistence unit with the name `HibernateJPA`, which we will use later, when creating an `EntityManagerFactory`. Within that, add all the classes that will be administered by the factory via its `EntityManager`. In addition, you need to define the same properties as already done with `hibernate.cfg.xml`, while giving the properties the prefix `hibernate`. The `persistence.xml` file created in that way is added via the `My Entities` module to the `src/META-INF` folder.

Entity Classes

The advantage of implementing JPA entities, thanks to the attribute accessor methods, is that no attribute needing to be persisted is required to have getters and setters. Nor does the attribute need to be explicitly exposed in any way.

JPA can also read and write private attributes. Additionally, no special interface needs to be implemented, nor does any need to be extended. Entities that are administered via JPA are therefore completely normal Java objects. Only a few annotations are necessary within the class.

In essence, the only annotations needed are those defining a class as an entity, via the `@Entity` annotation or the `@Id` attribute and the declaration of the attribute. The extent to which a class is constrained by specification is therefore extremely marginal. By default, a hierarchy of objects is administered within a relation. This mapping strategy can be adapted via annotations, for example, to customize a newly implemented object to a database schema.

From the perspective of the entity definition, once again use the classes `Genre` and `Album`. Simply add the `@Entity` annotation to the class definition, after which you add the `id` class attribute as the identity of the class via the `@Id` annotation (see Listing 13-16). At the same time, specify the persistence provider (in our case Hibernate) that will provide a value for the attribute. The `genre` attribute is a normal attribute and does not need to be identified in any particular way. It is automatically considered, so long as it has not been marked as transient.

Optionally, you can add the `@Basic` annotation to normal attributes. Take note of the fact that there is no requirement to add getters and setters for attributes. In this way, we omit the `getId()` and `setId()` methods from the `Genre` class because we don't need to do anything special with them ourselves.

Listing 13-16. *With only a few annotations, we define our class as an entity. It can then be saved in a relational database. The previous map file is completely superfluous in this case.*

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
@Entity
```

```
public class Genre {
   @Id
   @GeneratedValue
   private Long id;
   private String genre = new String();
   public Genre() {
   }
   ...
}
```

Little more than that needs be done for the `Album` class. To be exact, define a column name for the `year` attribute. By default, the column is named after the attribute itself. However, in the case of `year`, that would lead to failure at the first point of access, since `year` is part of SQL. Therefore, use the `@Column` annotation to define a user-specific name (see Listing 13-17). Finally, define the association for the `Genre` class as `@ManyToOne`, and right away both entities will be ready.

Listing 13-17. *Definition of the album entity with the association of the Genre class*

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
@Entity
public class Album {
   @Id
   @GeneratedValue
   private Long   id;
   private String title  = new String();
   private String tracks = new String();
   private String cds = new String();
   @Column(name = "years")
   private String year = new String();
   @ManyToOne
   private Genre genre;
   public Album() {
   }
   ...
}
```

To let entities use annotations, set a dependency on the `My Entities` module so it uses the `Hibernate` module. Fortunately, the Hibernate `EntityManager` makes use of the system class-loader (see Chapter 2), so there is no need to set a dependency in the `Hibernate` module on our entities, which would have led to a cyclic dependency. Had that been the case, wrapping the entities in a separate module could have resulted in a rather complex architecture. In this way, we arrive at the division and dependencies shown in Figure 13-6.

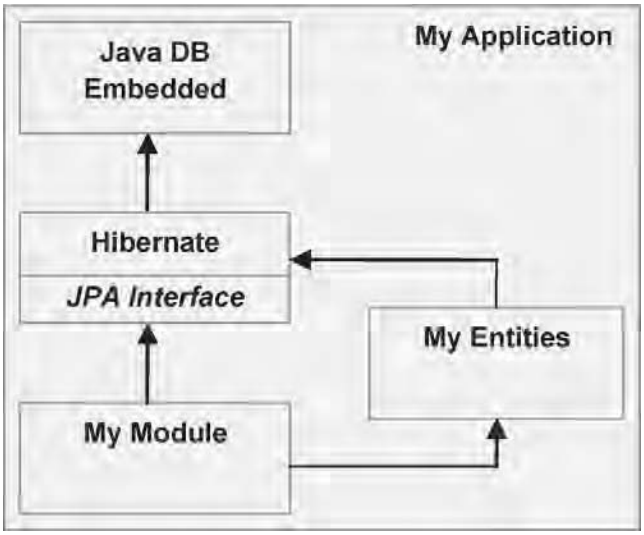


Figure 13-6. *Dependencies between the modules when using Hibernate's JPA interface*

EntityManagerFactory and EntityManager

Comparable to the `SessionFactory` in the native Hibernate interfaces, we have an `EntityManagerFactory` in the world of JPA. The factory creates a specific persistence unit for us. The persistence unit is the `EntityManager`, which is created by this factory. It is able to save objects that correspond to the persistence unit in their defined database and administer them there. An `EntityManagerFactory`, just like the `SessionFactory`, normally performs its creation task once during the application lifecycle.

We obtain an instance of this factory via the bootstrap class `Persistence`, using the following call:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("HibernateJPA");
```

The parameter `HibernateJPA` passes the name of the persistence unit, as defined in `persistence.xml`.

The counterpart to a session that creates a wrapper for a JDBC connection is handled by JPA in the `EntityManager` class. Via this manager, we gain access to the database, where we can save, delete, query, and find objects. As a rule, one `EntityManager` is used for one procedure.

However, it is bad practice to create a new `EntityManager` for each query or action performed. You must choose an appropriate lifecycle for an `EntityManager` in relation to the application context. For example, since we're dealing with only a few (and trivial) database actions in Listing 13-18, we simply use a single `EntityManager` for the entire lifecycle, which is not recommended.

As with the SessionFactory, you can manage an EntityManagerFactory in a module's ModuleInstall class. In that way, you can easily end the lifecycle of the factory when the application closes.

Listing 13-18. *The central administration and creation of the EntityManagerFactory*

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
public class Installer extends ModuleInstall {
   public static final EntityManagerFactory EMF;
   public static final EntityManager EM;
   static {
      try {
         EMF = Persistence.createEntityManagerFactory("HibernateJPA");
         EM = EMF.createEntityManager();
      } catch (Throwable ex) {
         throw new ExceptionInInitializerError(ex);
      }
   }
   @Override
   public void close() {
      EM.close();
      EMF.close();
   }
}
```

Saving and Loading Objects

Finally, we must discuss how to use the EntityManager to access objects. Again, take the DataModel class, in which interaction with the native Hibernate interfaces was implemented (see Listing 13-19). This is something we need now to let JPA handle for us. Take, for example, the methods getAlbums() and getGenres(). As per usual, create a transaction that executes our queries or actions. An instance of the EntityTransaction is received from the EntityManager via the getTransaction() method. Start the transaction with the begin() statement, and then create a new Query instance for the JPQL query SELECT a FROM Album a. The query provides us all the objects from the albums table. We place this result in a List via the getResultList() method. Using the commit() method, we successfully close the transaction.

Listing 13-19. *Interaction with the database via the EntityManager*

```
import javax.persistence.Query;
public class DataModel {
   public static List<Album> getAlbums() {
     Installer.EM.getTransaction().begin();
     Query q = Installer.EM.createQuery("SELECT a FROM Album a");
     List<Album> list = (List<Album>) q.getResultList();
     Installer.EM.getTransaction().commit();
     return list;
   }
```

```
    public static List<Genre> getGenres() {
      Installer.EM.getTransaction().begin();
      Query q = Installer.EM.createQuery("SELECT g FROM Genre g");
      List<Genre> list = (List<Genre>) q.getResultList();
      Installer.EM.getTransaction().commit();
      return list;
    }
    public static void updateAlbum(Album album) {
       Installer.EM.getTransaction().begin();
       Installer.EM.persist(album);
       Installer.EM.getTransaction().commit();
    }
    public static void insertAlbum(Album album) {
       updateAlbum(album);
    }
    public static void deleteAlbum(Album album) {
       Installer.EM.getTransaction().begin();
       Installer.EM.remove(album);
       Installer.EM.getTransaction().commit();
    }
}
```

Summary

In this chapter, you learned how to add database support to your application. There are a number of different approaches. Firstly, we looked at the Java DB database system. With Java DB, you have the possibility of integrating the complete system into your application, so there is no need for a database server. In this case, you can store your data via SQL on the client. Secondly, Hibernate is a very popular object-relational mapping (ORM) framework for storing objects in relational databases. We also looked at this framework and how to integrate and use it within a NetBeans Platform application. Thirdly, to be independent of a special ORM framework, you can use the Java Persistence API (JPA). It provides an abstraction over specific interfaces, enabling you to interchange ORM frameworks easily. We ended by adapting our previously created example to demonstrate how JPA is used.

CHAPTER 14

Web Services
Let's Integrate the Web!

Web services have been gaining popularity over the last several years, presenting themselves as interoperable services. In the NetBeans IDE, this trend is supported by a range of tools. When creating NetBeans Platform applications, it is useful to be aware of how web services can be used and integrated, which is the focus of this chapter.

Using Amazon E-Commerce Services (ECS) as an example, this chapter shows how to use the NetBeans IDE to create classes required for the use of web services, and includes an explanation of how to call them from a NetBeans Platform application. Taking this approach, your users will be able to search for products and product information, and execute other operations relating to ECS.

Creating a Web Service Client

Before creating a NetBeans module, create an independent Java application. Do this, as in the case of NetBeans modules, via a wizard. Go to File ➤ New Project. In the Java category, choose the Java Application project type. Assign the name Amazon E-Commerce Service to the application. A main class is not required in this case, so uncheck the related check box, and end the wizard by clicking Finish.

Next, right-click the application and choose New ➤ Web Service Client. If this menu item is not available, find it instead under Other ➤ Web Services. Specify the required WSDL file as follows:

```
http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
```

Of course, this only works if you are online. Rather than taking the online approach, you can download the file and feed it to the wizard as a local file (see Figure 14-1).

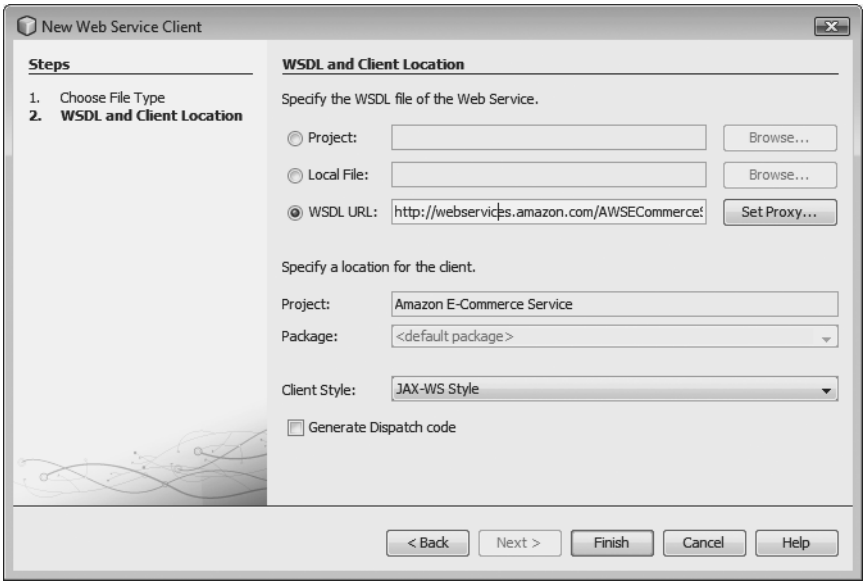


Figure 14-1. *Creation of a web service client for Amazon ECS*

Click Finish. The files that are required for the use of the web service are thereby created, using the information provided by the WSDL file. For a result that is easier to understand, deactivate the Wrapper Style setting, which is switched on by default. To this end, open the `Web Service References` node in the Projects window. There, find an entry for `AWSECommerceService`, right-click it to open the context menu, and choose Edit Web Service Attributes. On the WSDL Customization tab, expand `AWSECommerceServicePortType` under the `Port Types` node, and then deactivate the Enable Wrapper Style option (see Figure 14-2). The setting is applied to all operations of this port type.

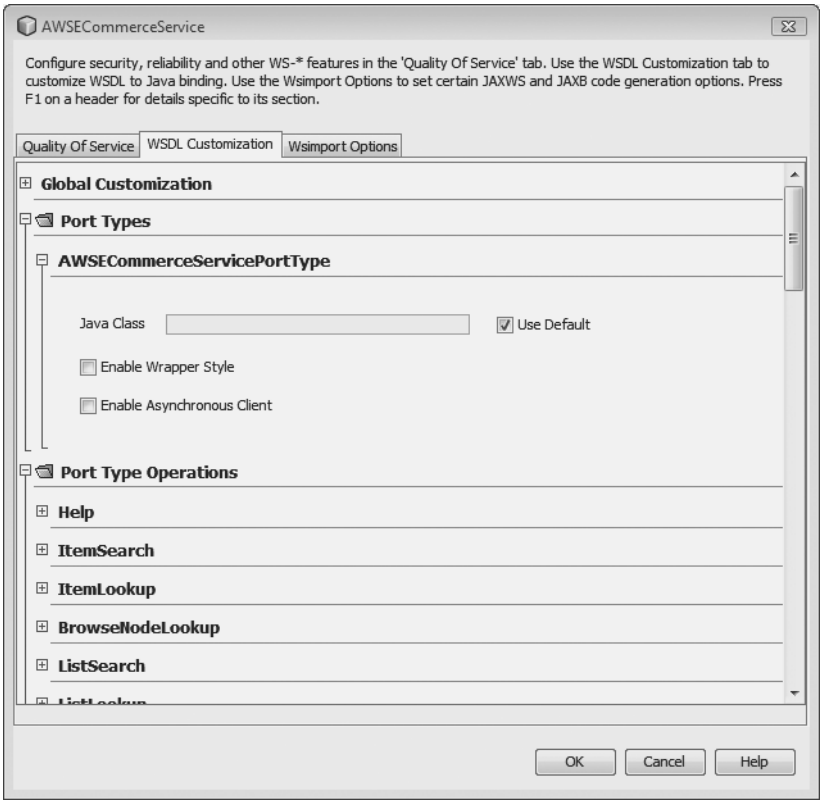


Figure 14-2. *Deactivating the wrapper style for the Amazon web service*

When you click OK, the web service client classes are updated. Now do a build, resulting in a web service client in the form of the JAR file `dist/Amazon_E-Commerce_Service.jar` in the main project directory. In the next section, we'll wrap this JAR file in a NetBeans module and use it as our Amazon ECS API.

Using a Web Service

The previous section showed how to create a JAR file containing the Java classes that allow using Amazon ECS. In exactly the same way, you also create client files for other web services. We will now examine how to use these files to let our NetBeans Platform application access the web service.

To that end, create a new NetBeans Platform application by choosing File ➤ New Project ➤ NetBeans Modules ➤ NetBeans Platform Application. Give the application any name you like. However, we need more than the modules that the NetBeans Platform provides—specifically, we need the modules `JAX-WS 2.1 API`, `JAXWS 2.1 Library`, and `JAXB 2.1 Library`. Activate the first two of these in the `java` cluster of the application's Project Properties dialog, and activate the third in the `ide` cluster. With that, our application is configured, and we can now add the JAR file containing the web service client to our application.

Start by creating a new library wrapper module, right-clicking the `Modules` node, and choosing Add New Library. In the Library field, browse to the JAR file created earlier. Click Next. Name the module `Amazon E-Commerce Service` and click Next. Enter `com.amazon` as the code name base, and then click Finish. Expand the new module project in the Projects window, right-click the `Libraries` node, and choose Add Module Dependency. Click Show Non-API Modules, and then select `JAX-WS 2.1 API`, `JAXWS 2.1 Library`, and `JAXB 2.1 Library` as dependencies. Finally, within the module project's `Libraries` node, right-click the `JAXWS 2.1 Library` and `JAXB 2.1 Library` nodes, choose Edit, and then choose Implementation Version.

With that, the web service client is ready to be used within the NetBeans Platform application. We'll now create a new module from which we'll call out to the web service client classes. In the Projects window, right-click the application's `Modules` node, and choose `Add New`. Create a new module and set a dependency on the `Amazon E-Commerce Service` module. To set up a small example, create a new `TopComponent` using the Window Component wizard.

In the example, use the Amazon Standard Identification Number (ASIN) to search for available pictures of a product. Looking ahead to the MP3 Manager that we'll create in Chapter 18, adapt the example as follows: save the ASIN into an MP3 file's ID3 tag, enabling you to show Amazon's CD cover of the MP3 file currently being played. Alternatively, use the example to search for albums of the currently playing MP3 file and display them to the user.

In the example application, as you can see in Figure 14-3, we use the ASIN to search for a product and show its cover, as is done on the Amazon.com site. To that end, let's look more closely at the queries that we use to interact with the Amazon web service. The requests need to be performed asynchronously, avoiding a situation where the whole application is blocked. At the same time, be aware that operations on the GUI components should only be performed over the EDT. In the end, we want to be informed when the query has succeeded, which is when we can display the image. The simplest approach to solving these related concerns is to use the `SwingWorker` class. To that end, we create our own class, deriving from `SwingWorker<String, Object>`.

The `doInBackground()` method, which we need to override, is automatically called asynchronously. This is where web service queries are coded (see Listing 14-1). When the queries succeed, the `done()` method in the `SwingWorker` is called. That is where we then use the `get()` method to access the value returned by the `doInBackground()` method. In the example, the value is the URL of the product image.

Listing 14-1. *Executing a web service request and showing the results with the help of the SwingWorker class*

```
import com.amazon.webservices.awsecommerceservice.AWSECommerceService;
import com.amazon.webservices.awsecommerceservice.AWSECommerceServicePortType;
import com.amazon.webservices.awsecommerceservice.ImageSet;
import com.amazon.webservices.awsecommerceservice.Item;
import com.amazon.webservices.awsecommerceservice.ItemLookup;
import com.amazon.webservices.awsecommerceservice.ItemLookupRequest;
import com.amazon.webservices.awsecommerceservice.ItemLookupResponse;

final class ECSTopComponent extends TopComponent {
   private static final String AWS_KEY = "<your personal aws key>";
   ...
   private final class ImageLookupByASIN extends SwingWorker<String, Object> {
      private String asin = new String();

      public ImageLookupByASIN(String asin) {
         this.asin = asin;
      }

      @Override
      public String doInBackground() {
         String url = new String();
         try {
            AWSECommerceService service = new AWSECommerceService();
            AWSECommerceServicePortType port = service.getAWSECommerceServicePort();
            ItemLookupRequest request = new ItemLookupRequest();
            request.setIdType("ASIN");
            request.getItemId().add(asin);
            request.getResponseGroup().add("Images");
            ItemLookup il = new ItemLookup();
            il.setAWSAccessKeyId(AWS_KEY);
            il.getRequest().add(request);
            ItemLookupResponse response = port.itemLookup(il);
            Item i = response.getItems().get(0).getItem().get(0);
            ImageSet is = i.getImageSets().get(0).getImageSet().get(0);
            url = is.getThumbnailImage().getURL();
         } catch (Exception e) {
            e.printStackTrace();
         }
         return url;
      }
      @Override
      protected void done() {
         try {
            cover.add(new JLabel(new ImageIcon(new URL(get()))));
            cover.updateUI();
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
```

```
    }
    private void searchActionPerformed(ActionEvent evt) {
        new ImageLookupByASIN(asin.getText()).execute();
    }
}
```

In the asynchronously called `doInBackground()` method, we get the port for Amazon ECS. Then we invoke the `ItemLookup` operation. Configure the operation via the `ItemLookup` and `ItemLookupResult` objects. Using the `Request` object, specify the use of the ASIN to look for a product. Add the ASIN, which the `ImageLookupByASIN` object passes as a parameter, to the Item ID list. Using the Response group, process the information received from the web service. Since we are only interested in the URL of the product image, use the Images group type.

To use Amazon Web Services (AWS), you need an AWS access key, which you can get for free after subscribing at `http://aws.amazon.com`. Pass the key in to the `ItemLookup` object, via the method `setAWSAccessKeyId()`. Then pass in the `ItemLookupRequest` object. In doing so, you prepare the required parameters for the request and can perform the `itemLookup()` request on the web service port obtained earlier. The response is received in the `ItemLookupResponse` object containing a list of found products in the form of an `Item` object. Since there is only one product per ASIN, immediately take the first item from the list. An item includes an `ImageSet`, from which the URL of the product image is extracted. That is the value returned from the method.

And with that, the request ends and the `SwingWorker` class calls the `done()` method. In this method, we obtain the URL via the `get()` method and use it to access the `ImageIcon` object, which is then shown to the user (see Figure 14-3).

Figure 14-3. *Product information query over the Amazon e-commerce service*

Summary

This chapter dealt with the topic of web services. We created a Web Service API (a web service client) for AWS from the corresponding WSDL file. To this end, we made use of the related NetBeans IDE tooling support. Next, you learned how to integrate the web service client into your own NetBeans Platform application. You also saw which additional libraries are necessary in this scenario. All in all, the example showed how easy it is to use a web service in your own application.

CHAPTER 15

Extending the NetBeans IDE

Let's Add Some Features to the NetBeans IDE!

The NetBeans IDE itself is a rich client application. It provides its functionality in the form of modules on top of the NetBeans Platform. That means you can extend the functionality the IDE provides the same way you would your own rich client application—by adding modules. For that purpose, in this chapter we discuss aspects that are important when dealing with the IDE.

Palettes

In Chapter 9, we developed a palette. From this palette, we could drag and drop music albums onto a specially created `TopComponent`. The possibility of registering a palette to a specific file type was mentioned. Registering automatically opens a registered palette whenever a file of that type is opened in the NetBeans editor. We will be working out an example of how that is achieved.

Assume that we would like to create a palette for manifest (`.mf`) files (see Figure 15-1). To achieve this, we do the following. For every entry the palette provides, we register an XML file in the layer file. Then we implement a class that creates a palette for the registered palette entries. Finally, we register that class to the manifest file type in the layer file.

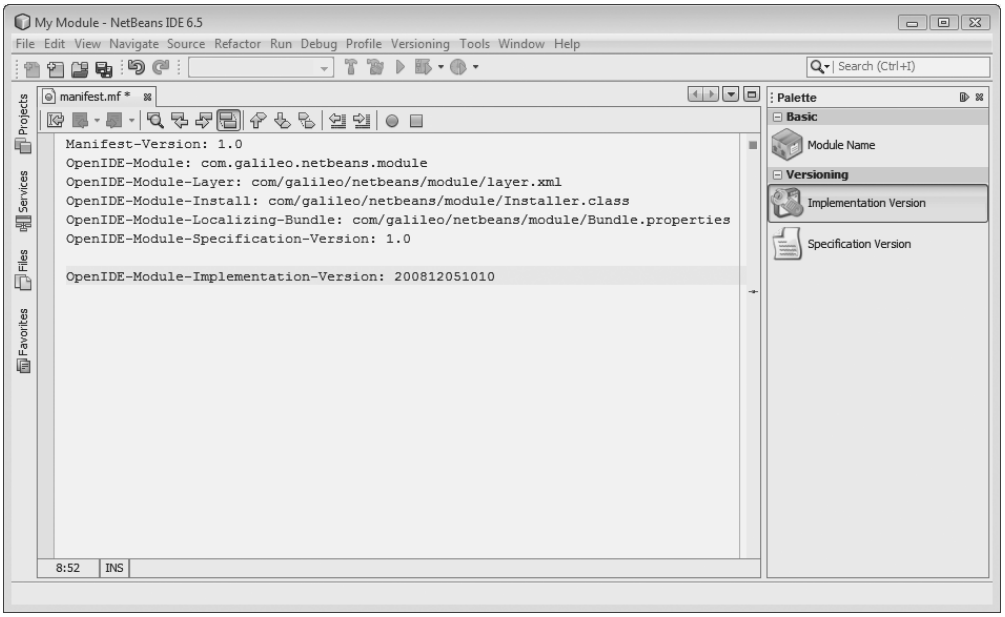


Figure 15-1. *Palette for manifest files*

Defining and Registering Palette Entries

Every palette entry is defined by an XML file of the `editor-palette-item` type (see DTD in the Appendix). In that file, we declare a class that is called when dragging and dropping to handle the inserts. We also declare two icons of differing sizes, as well as the text and tooltip for the entry. For the palette entry `Module Name`, the file looks like Listing 15-1.

Listing 15-1. *Definition of the Module Name palette entry by means of an XML file*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE editor_palette_item PUBLIC
   "-//NetBeans//Editor Palette Item 1.1//EN"
   "http://www.netbeans.org/dtds/editor-palette-item-1_1.dtd">
<editor_palette_item version="1.1">
  <class name="com.galileo.netbeans.module.items.ModuleName"/>
  <icon16 urlvalue="com/galileo/netbeans/module/resources/ModuleName16.png"/>
  <icon32 urlvalue="com/galileo/netbeans/module/resources/ModuleName32.png"/>
  <inline-description>
     <display-name>Module Name</display-name>
     <tooltip>Module Name</tooltip>
  </inline-description>
</editor_palette_item>
```

INTERNATIONALIZATION OF PALETTE ENTRIES

To enable internationalization of palette entries, you can move the values of the text elements `display-name` and `tooltip` into a resource bundle file. For that purpose, the `inline-description` element is replaced by the `description` element. Instead of the values for those text elements, we declare the keys with which the values can be retrieved from the resource bundle. Of course, we will also declare the resource bundle to be used:

```
<description localizing-bundle="com.galileo.netbeans.module.Bundle"
  display-name-key="DISPLAY"
  tooltip-key="TOOLTIP"/>
```

In the `class` element, we declared the class `ModuleName`. That class is called whenever the user drags the entry from the palette into the editor of a manifest file. Therefore, the class has to implement the `ActiveEditorDrop` interface. The interface is a component of the Text API to which you must declare a dependency. Upon a drop event, the method `handleTransfer()` from the `ActiveEditorDrop` interface is called automatically. In that call, a `JTextComponent` is passed as a parameter. Via the `JTextComponent`, we get access to the current document—our manifest file—where we want to insert our entry. Since the process is very repetitive for each palette entry and differs only in the text to be inserted, we will be implementing the abstract class `ManifestItem` (see Listing 15-2). That class is responsible for inserting text into the manifest document. Text is supplied by the method `getItem()`, which subclasses must implement.

Listing 15-2. *Abstract class taking responsibility of inserting text into the manifest file*

```
import org.openide.text.ActiveEditorDrop;
public abstract class ManifestItem implements ActiveEditorDrop {
   public abstract String getItem();
   public boolean handleTransfer(JTextComponent editor) {
      try {
         Document doc = editor.getDocument();
         int pos = editor.getCaretPosition();
         doc.insertString(pos, getItem() + "\n", null);
      } catch (BadLocationException ex) {
         Logger.getLogger(ManifestItem.class.getName()).log(Level.SEVERE, null, ex);
      }
      return true;
   }
}
```

The classes for specific palette entries are trivial to implement:

```
public class ModuleName extends ManifestItem {
   public String getItem() {
      return "OpenIDE-Module-Name: My Module";
   }
```

```
}

public class ModuleSpecVersion extends ManifestItem {
   public String getItem() {
      return "OpenIDE-Module-Specification-Version: 1.0";
   }
}
```

You can extend these classes to allow the user to actively declare the values—in this case the name or the version of the module—for the entries in, e.g., a dialog.

To finish the first step of defining the entries, we need to register them in the layer file in a specially created folder. In this case, we use the folder `ManifestPalette` (see Listing 15-3). Every folder declared there will be a category in the palette by which entries can be grouped.

Listing 15-3. *Registration of palette entries in a separate folder*

```
<folder name="ManifestPalette">
   <folder name="Basic">
      <file name="ModuleName.xml" url="items/ModuleName.xml"/>
   </folder>
   <folder name="Versioning">
      <file name="ModuleSpecVersion.xml" url="items/ModuleSpecVersion.xml"/>
      <file name="ModuleImplVersion.xml" url="items/ModuleImplVersion.xml"/>
   </folder>
</folder>
```

Creating and Registering a PaletteController

We implemented the palette entries and registered them in the `ManifestPalette` folder in the layer file. We will now create a `PaletteController` instance for this folder that manages the entries. Therefore, we create a class named `ManifestPalette`. There, we add the method `createPalette()`, which will, with the help of the `PaletteFactory` class from the Palette API, create a `PaletteController` instance, as shown in Listing 15-4.

Listing 15-4. *The PaletteController is the manager of our entries.*

```
import org.netbeans.spi.palette.PaletteActions;
import org.netbeans.spi.palette.PaletteController;
import org.netbeans.spi.palette.PaletteFactory;
public class ManifestPalette {
   private static PaletteController palette;
   public static PaletteController createPalette() {
      try {
         if (palette == null) {
            palette = PaletteFactory.createPalette(
                    "ManifestPalette",
                    new MyPaletteActions());
         }
         return(palette);
      } catch (Exception ex) {
         Logger.getLogger(
```

```
                ManifestPalette.class.getName()).log(Level.SEVERE, null, ex);
        }
        return null;
    }
    private static final class MyPaletteActions extends PaletteActions {
        ...
    }
}
```

As the final part of this example, we must register the `PaletteController` to the manifest file type. To do that, we determine the MIME type of manifest files first. That we can do quite easily with the layer tree in the project view of our module. You will see the MIME type `text/x-manifest` under `Important Files` ➤ `XML Layer` ➤ `<this layer in context>` ➤ `Editors`. Therefore, we register the controller in the folder `Editors/text/x-manifest`:

```
<folder name="Editors">
    <folder name="text">
        <folder name="x-manifest">
            <file name="ManifestPalette.instance">
                <attr name="instanceOf"
                    stringvalue="org.netbeans.spi.palette.PaletteController"/>
                <attr name="instanceCreate" methodvalue=
                    "com.galileo.netbeans.module.ManifestPalette.createPalette"/>
            </file>
        </folder>
    </folder>
</folder>
```

From now on, every time a manifest file is opened in the editor, a controller for manifest palette entries is created by calling the method `createPalette()`. This controller is made available for the `Palette` module via the Lookup. It will display the corresponding entries as depicted in Figure 15-1.

Expanding Existing Palettes

Aside from creating a palette for a file type that previously had no palette, you can add entries to a preexisting palette. The name for the folder in the layer file must be known beforehand. The layer tree (`Important Files` ➤ `XML Layer`) is a good place to search for already existing palette folders. The folder for, e.g., `.html` files, is named `HTMLPalette`. Add entries to existing folders in the same way you would add them to your own `ManifestPalette` folder. It should look like Listing 15-5.

Listing 15-5. *Adding entries to an existing palette*

```
<folder name="HTMLPalette">
    <folder name="My HTML Items">
        <file name="item1.xml" url="items/item1.xml"/>
        <file name="item2.xml" url="items/item2.xml"/>
    </folder>
</folder>
```

Task List API

The `Task List` module of the NetBeans IDE allows the display of all-purpose information, like tasks, notifications, or error messages, as can be seen in Figure 15-2. Task list entries can be grouped so that the user is provided a better overview of information. The user can determine from what sections entries shall be shown. By default, three sections—called scopes—are defined in the `Task List` module. One scope corresponds to the currently opened file, another to the main project and its opened dependent projects, and the third to all opened projects. The entries are supplied by scanners working with the fixed scope.

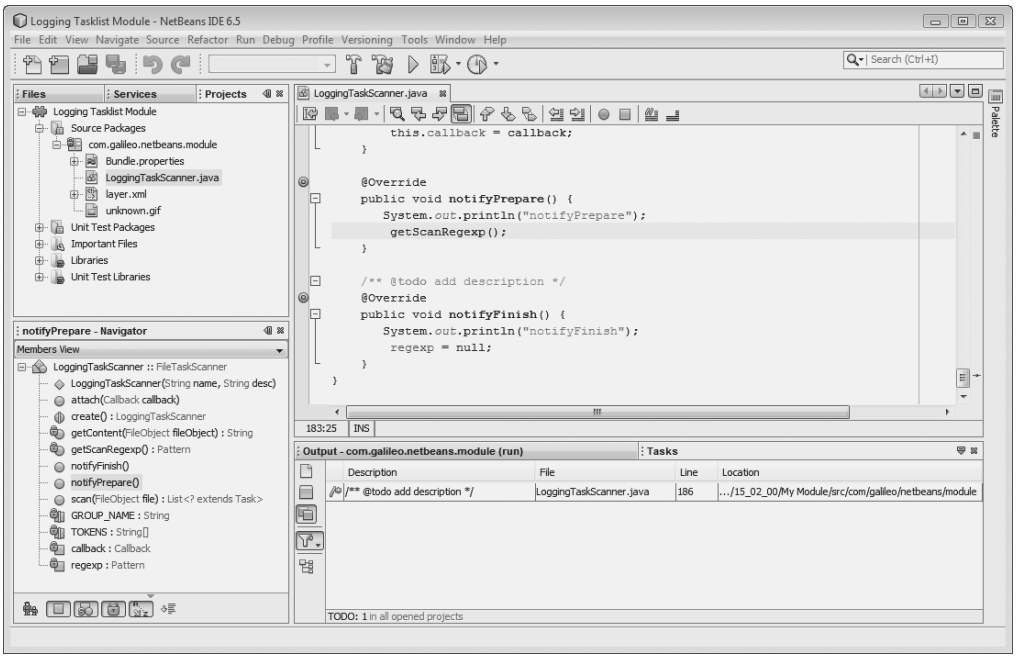


Figure 15-2. *Task List module of the NetBeans IDE*

The Task List API dynamically expands the scope of operation of the `Task List` module, primarily by providing additional scanners. These extensions integrate with the help of extension points in the layer file. We will show how that works in an example. We will need to implement a scanner displaying all code fragments with a direct output of information, like `System.out.println()`. Doing that enables us to ascertain that, prior to the release of a product, every unwanted direct output is replaced by logging output or removed.

Our `LoggingTaskScanner` extends from the abstract class `FileTaskScanner`. Each scanner has a name and description of its capability. There is an optional link to an options panel from which the scanner can be configured individually. An example is the `ToDo` scanner of the IDE. The tokens identified as `ToDo` tasks can be configured there. For reasons of simplicity, we define these tokens directly in our scanner. The constructor of our scanner calls the superconstructor with three parameters: the name, the description, and the path to the options panel (`null` if there is none). Since the scanner is registered declaratively in the layer file to be initialized by the

Task List framework, we provide the factory method `create()` to create a `LoggingTaskScanner` instance.

The important part of the scanner—as you may have guessed—is the `scan()` method (see Listing 15-6). In a parameter, we access a file to be searched. With the help of a `Pattern` (to identify the tokens) and a `Matcher`, we search the file. For every occurrence of the tokens, we create a task instance that we add to a list that is returned once we are done searching the file. The `TodoTaskScanner` provides the template for the following implementation. A task instance is created via the static method `Task.create()`, with parameters for the searched file, the group the entry belongs to, a description (usually the line with the occurrence), and the line number.

Listing 15-6. *Scanner implementation*

```
import org.netbeans.spi.tasklist.FileTaskScanner;
import org.netbeans.spi.tasklist.Task;
import org.openide.filesystems.FileObject;
public class LoggingTaskScanner extends FileTaskScanner {
   private static final String GROUP_NAME = "logging-tasklist";
   private static final String[] TOKENS = {
      "System.out.println",
      "System.err.println",
      "printStackTrace"};
   private Pattern  regexp   = null;
   private Callback callback = null;
   public LoggingTaskScanner(String name, String desc) {
      super(name, desc, null);
   }
   public static LoggingTaskScanner create() {
      String name = NbBundle.getBundle(LoggingTaskScanner.class).
                             getString("LBL_loggingtask");
      String desc = NbBundle.getBundle(LoggingTaskScanner.class).
                             getString("HINT_loggingtask");
      return new LoggingTaskScanner(name, desc);
   }
   public List<? extends Task> scan(FileObject file) {
      List<Task> tasks = new LinkedList<Task>();
      try {
         String text = getContent(file);
         int index  = 0;
         int lineno = 1;
         int len    = text.length();
         Matcher matcher = getScanRegexp().matcher(text);
         while (index < len && matcher.find(index)) {
            int begin = matcher.start();
            int end   = matcher.end();
            ...
            String description = text.subSequence(begin, nonwhite + 1).toString();
            Task task = Task.create(file, GROUP_NAME, description, lineno);
            tasks.add(task);
         }
      } catch(Exception e) {
         Logger.getLogger(getClass().getName()).info(e);
```

```
        }
        return tasks;
    }
    private String getContent(FileObject file) {
        // extract the content from the file
    }
    private Pattern getScanRegexp() {
        if (regexp == null) {
            // create pattern for the tokens
        }
        return regexp;
    }
    public void attach(Callback callback) {
        if(callback == null && this.callback != null) {
            regexp = null;
        }
        this.callback = callback;
    }
    @Override
    public void notifyPrepare() {
        getScanRegexp();
    }
    @Override
    public void notifyFinish() {
        regexp = null;
    }
}
```

In the context menu of the Task List window, users can activate and deactivate the scanner. We are informed of changes in scanner states of activation with a call to the method `attach()`. If the value of the `callback` parameter is `null`, the scanner has been disabled. Via the `callback` instance, we access the Task List framework. The method `notifyPrepare()` is called, prior to the initiation of a scan, by the Task List framework. It allows us to prepare the upcoming call of `scan()`. The `notifyFinish()` method is called last.

The Task List framework defines the following three extension points that allow the registration of extensions:

- `TaskList/Groups`
- `TaskList/Scanners`
- `TaskList/Scopes`

Initially, we want to create a new group for logging tasks. We have already declared the group ID as `logging-tasklist` in the scanner. That allows us to assign tasks created in the scanner to a group. Creating a group is done by simply calling the `createGroup()` method of the `Task` class. We must specify attributes to configure the group. Among those attributes are an ID and keys from a resource bundle (see Listing 15-7). Registering the scanner, we must declare the base class and factory method.

Listing 15-7. *Creation of a Task group and registration of the scanner via the extension points of the Task List framework*

```
<filesystem>
  <folder name="TaskList">
    <folder name="Groups">
      <file name="LoggingTaskGroup.instance">
        <attr name="instanceCreate"
           methodvalue="org.netbeans.spi.tasklist.Task.createGroup"/>
        <attr name="localizingBundle"
           stringvalue="com.galileo.netbeans.module.Bundle"/>
        <attr name="groupName" stringvalue="logging-tasklist"/>
        <attr name="diplayNameKey" stringvalue="LBL_loggroup"/>
        <attr name="descriptionKey" stringvalue="HINT_loggroup"/>
        <attr name="iconKey" stringvalue="ICON_logging"/>
        <attr name="position" intvalue="400"/>
      </file>
    </folder>
    <folder name="Scanners">
      <file name="LoggingTaskScanner.instance">
        <attr name="instanceOf"
           stringvalue="org.netbeans.spi.tasklist.FileTaskScanner"/>
        <attr name="instanceCreate" methodvalue=
           "com.galileo.netbeans.module.LoggingTaskScanner.create"/>
      </file>
    </folder>
  </folder>
</filesystem>
```

Summary

In this chapter, the NetBeans IDE was presented as a NetBeans Platform application. Its features can be extended in the same way as is done with your own NetBeans Platform applications. In addition to the NetBeans Platform modules, you were exposed to several APIs provided by the NetBeans IDE. In particular, you learned about the Palette API and the Task List API and SPI.

CHAPTER 16

From Eclipse RCP to the NetBeans Platform

Let's Abandon SWT!

This chapter guides the migration of applications developed on top of Eclipse RCP to the NetBeans Platform. Related differences between the Eclipse IDE and the NetBeans IDE are also highlighted where appropriate in migrating an application from Eclipse RCP to the NetBeans Platform.

The NetBeans IDE

This section introduces techniques, fundamental characteristics, and functions of the NetBeans IDE. It helps smooth the transition from the Eclipse IDE to the NetBeans IDE, and provides an introduction to the NetBeans Platform.

Standard Components

Table 16-1 provides a basic overview of where you can find the windows and functions you know from the Eclipse IDE.

Table 16-1. *The NetBeans IDE equivalents for Eclipse components*

Eclipse Component	NetBeans Menu Item
`Project Explorer/Package Explorer`	Window ➤ Projects
`Projects/Navigator`	Window ➤ Files
`Outline`	Window ➤ Navigating ➤ Navigator
`Properties`	Window ➤ Properties
`Console`	Window ➤ Output ➤ Output
`Problems/Tasks`	Window ➤ Task List
`Javadoc`	Window ➤ Other ➤ Javadoc View

Table 16-1. *The NetBeans IDE equivalents for Eclipse components (Continued)*

Eclipse Component	NetBeans Menu Item
`Error Log`	View ➤ IDE Log File
`Plug-in Registry`	Tools ➤ Plugins
`Preferences`	Tools ➤ Options

Handling Projects

In the NetBeans IDE, mapping keyboard shortcuts to provided functionalities is done via a *keymap*. This keymap is viewed and edited under Tools ➤ Options ➤ Keymap. Adapt the actions to your own needs. It is also possible to administrate several keymaps at the same time. The NetBeans IDE provides an Eclipse keymap, making it possible to switch to Eclipse keymaps (also under Tools ➤ Options ➤ Keymap) and use the well-known shortcuts further on—a benefit for experienced Eclipse users.

Something often missed by Eclipse users in the NetBeans IDE is the *perspective* feature. However, a module providing the perspective feature is available at `http://contrib.netbeans.org/perspective`. Thus, you can use perspectives in the NetBeans IDE. This module can be downloaded and installed through the Plugin Manager (Tools ➤ Plugins).

From Eclipse Plugins to NetBeans Modules

The concept of a plugin in the Eclipse world is equivalent to a NetBeans module. As in the Eclipse IDE, the NetBeans IDE offers a wizard providing the basic structure for a module in a few clicks. Perform this via File ➤ New Project ➤ NetBeans Modules ➤ Module.

While creating a plugin with the Eclipse IDE, several parameters must be declared from the beginning. Among them is the activator, the GUI, and the rich client application functionality of the new plugin. The NetBeans IDE Module wizard takes a more general approach. In all three points specified before, decisions can be made later on whether the functionality is needed. An activator—called a NetBeans module installer (more on this later on)—can be added anytime via a separate wizard. This wizard is found under File ➤ New File ➤ Module Development ➤ Module Installer.

No need to bother about whether the module will come with a graphic interface. On this point, separate wizards are available, and you can use them later as needed. One of the most important wizards is the Window Component wizard, for the construction of windows, which are docked and administered in the NetBeans window system. Start this wizard via File ➤ New File ➤ Module Development ➤ Window Component.

You can decide whether to provide a rich client application that the module will be a part of, or whether this module is to become an extension of an already existing application. Although the module wizard asks whether a standalone module or an application module is needed, this is easily changed in the Properties dialog of a module, or simply through adding and removing the module from an application.

For modules to become self-contained rich client applications, a NetBeans Platform Application project is needed. This project is a container for your modules, and is responsible for branding your application. To create a NetBeans Platform application, a wizard is also

provided. Find it under File ➤ New Project ➤ NetBeans Modules ➤ NetBeans Platform Application. Both new and existing modules can be added to the new application.

The range of NetBeans modules used by applications, and hence used by modules, is determined under Properties ➤ Libraries. Applications are by no means limited to NetBeans Platform modules. Arbitrary NetBeans IDE modules can be added to your application.

Plugin: Lifecycle and Events

An Eclipse plugin may contain an `Activator`. This class extends the abstract class `Plugin` or `AbstractUIPlugin`, depending on whether the plugin contains graphic elements. This optional class serves as the conceptual representation of the plugin. Containing no application logic, it reacts to distinguished events—for instance, the methods `start()` and `stop()` specified by the interface `BundleActivator` and implemented by the classes `Plugin` and `AbstractUIPlugin`.

The methods are called by the Eclipse Platform when the plugin is loaded or closed. By overwriting these methods, special platform-specific tasks are executed at these times. An `Activator` in its simplest form looks as shown in Listing 16-1.

Listing 16-1. *Activator class of an Eclipse plugin*

```
package com.galileo.eclipse.plugin;
import org.eclipse.core.runtime.Plugin;
import org.osgi.framework.BundleContext;
public class Activator extends Plugin {
   private static Activator plugin;
   public void start(BundleContext context) throws Exception {
      super.start(context);
      plugin = this;
   }
   public void stop(BundleContext context) throws Exception {
      plugin = null;
      super.stop(context);
   }
   public static Activator getDefault() {
      return plugin;
   }
}
```

The counterpart to the Eclipse plugin activator is the module installer of a NetBeans module. This module installer is optional. The NetBeans platform instantiates an installer during module startup. The installer extends the class `ModuleInstall` (see Listing 16-2).

This class specifies the methods `restored()` and `close()`, which are equivalent to methods in the `BundleActivator` interface. Also available are `validate()`, for the examination of the starting conditions; `closing()`, for the examination of stop conditions; and `uninstalled()`, for uninstallation of the module. As in activators, these methods can be overwritten and used as required.

Listing 16-2. *The counterpart to the activator is a NetBeans module installer.*

```
package com.galileo.netbeans.module;
import org.openide.modules.ModuleInstall;
```

```
public class Installer extends ModuleInstall {
   @Override
   public void restored() {
      // module started
   }
   @Override
   public void close() {
      // module stopped
   }
   public static Installer getDefault() {
      return findObject(Installer.class, true);
   }
}
```

While the activator of an Eclipse plugin is automatically created by a new plugin project, the installer can be created anytime with the Module Installer wizard, found under File ➤ New File ➤ Module Development. Thus, the installer is also registered in the manifest file. You can find more detailed information on this in Chapter 3.

Plugin Information

Apart from reaction to starting and stopping plugins, the activator class has further functionalities. It can also provide plugin and manifest information via a `Bundle` object. Information on NetBeans modules is offered by the NetBeans Platform in the `ModuleInfo` objects. Instances for all modules within applications are available on the Lookup. `ModuleInfo` instances can be found in this map. They are provided to the module user via the `Installer` class and the `getModuleInfo()` method, as shown in Listing 16-3.

Listing 16-3. *Providing the ModuleInfo instance, which contains information on the module*

```
package com.galileo.netbeans.module;
import org.openide.modules.ModuleInfo;
import org.openide.modules.ModuleInstall;
import org.openide.util.Lookup;
public class Installer extends ModuleInstall {
   public static final String MODULE_ID = "com.galileo.netbeans.module";
   private ModuleInfo info = null;
   ...
   public ModuleInfo getModuleInfo() {
      if(info == null) {
         Collection<? extends ModuleInfo> all =
                Lookup.getDefault().lookupAll(ModuleInfo.class);
         for(ModuleInfo mi : all) {
            if(mi.getCodeNameBase().equals(MODULE_ID)) {
               info = mi; break;
            }
         }
      }
      return info;
   }
}
```

Images

Pictures and icons used within an application are not loaded over the installer, but over a central `ImageUtilities` class in the NetBeans world. The method `loadImage()` should be used.

An icon manager manages images and icons, preventing repeated loading of resources. Use it to load icons from all available modules. It is also possible to load localized resources, as in the following example. If the second parameter is set to `true` and there is an icon named `icon_de_DE.png` available, then it is loaded (if the locale setting of application is `de_DE`):

```
Image img = ImageUtilities.loadImage("resources/icon.png", true);
```

Resources

Any plugin resource may be accessed by using the `FileLocator` class in Eclipse. In order to load resources simply from a NetBeans module, extend the `Installer` class by the method `getModuleResource()` (see Listing 16-4). Use the module classloader that has access to all module resources. This returns a URL which, using the `URLMapper` class, maps to the `FileObject` instance.

Listing 16-4. *The getModuleResource() method helps load module resources.*

```
package com.galileo.netbeans.module;
import java.net.URL;
import org.openide.filesystems.FileObject;
import org.openide.filesystems.URLMapper;
import org.openide.modules.ModuleInstall;
public class Installer extends ModuleInstall {
   ...
   public FileObject getModuleResource(String path) {
      URL url = getClass().getClassLoader().getResource(path);
      FileObject resource = URLMapper.findFileObject(url);
      return resource;
   }
}
```

The `FileObject` class provides extensive methods for working with resources. The following example proves that by loading content of the `myprops.properties` file out of the `resources` directory of the module into the `Properties` object.

```
public final class TestAction implements ActionListener {
   public void actionPerformed(ActionEvent evt) {
      FileObject res =
         Installer.getDefault().getModuleResource("resources/myprops.properties");
      Properties props = new Properties();
      try {
         props.load(res.getInputStream());
      } catch(Exception e) {}
   }
}
```

Settings

Plugin-specific settings, used internally as well as by the user, are managed via the `Preferences` class or the `IPreferenceStore` class in Eclipse RCP, which is set up via the `Activator` class. The NetBeans Platform takes a slightly different approach. Managing settings is handled by the Java Preferences API. Access to the `Preferences` instance is obtained via the `NbPreferences` class. An advantage of this implementation is that data is stored in the NetBeans Platform user directory.

A distinction is made between module-specific data and application-specific data. The `root()` method gives access to settings saved in the `config/Preferences.properties` file. The `forModule()` method, on the other hand, handles access to data found in module-specific properties files. For example, if the code name base is `com.galileo.netbeans.module`, settings will be stored in the `config/Preferences/com/galileo/netbeans/module.properties` file.

```
NbPreferences.forModule(MyClass.class).put("key", "value");
NbPreferences.root().put("key", "value");
```

You can find more detailed information on this topic in Chapter 9.

Application Lifecycle

The lifecycle of Eclipse RCP applications is handled by the `IApplication` class. It implements the `start()` and `stop()` methods. The first is responsible for starting the application, typically used for creating and opening the main window. The `stop()` method handles the shutting down of the application, where the workbench is closed and other application-specific tasks are carried out.

NetBeans Platform applications have no access to a lifecycle manager of this kind. On the other hand, that considerably simplifies development. For example, the main window opens and closes on its own, without coding.

Most aspects relating to the lifecycle of an application are handled within individual modules and dealt with via a module installer. Another possibility is to react to the closing of the whole application, rather than the uninstalling of individual modules—this is done via the abstract class `LifecycleManager`.

The default implementation of the `LifecycleManager` class, provided by the NetBeans Platform, is responsible for shutting down applications. You can insert your own implementation of this class before the default implementation, so application-specific tasks are handled as the application shuts down.

Do not forget to call the default implementation from your own implementation. Some implementations of this class are illustrated in Chapter 17. Finally, the application may be explicitly ended via the `LifecycleManager` (also described in Chapter 17).

Views and Editors

The windows the Eclipse workbench displays and docks are of two types: *views* and *editors*. The NetBeans Platform makes no such distinction. A window displayed within the NetBeans window system is a `TopComponent`. The implementation of a window is derived from the `TopComponent` class. This superclass integrates itself as a window into the NetBeans window system and makes a great deal of information available, giving access to its current state as well as its lifecycle.

Similar to how views and editors in an Eclipse application are organized, via the relevant extension points, `TopComponent`s are declaratively made available, via the layer file, within the `Mode` folder. A mode is a container for `TopComponent`s. Here, we return to the view and editor distinction, since modes may be one of these types. `TopComponent`s are created within one of these modes (see Figure 5-8 in Chapter 5). A mode's size, position, and type are described in an XML file, registered within the layer file (see Chapter 5 and the Appendix for further information). However, `TopComponent`s are not required to be registered within a mode, where they are displayed in default mode. As with the Eclipse workbench, the user places `TopComponent`s in various positions while the application is running.

To simplify creation and registration of `TopComponent`s, the NetBeans IDE provides a very useful wizard. Choose File ➤ New File ➤ Module Development ➤ Window Component and start using it. For example, the wizard puts the `TopComponent` into a mode, and then handles the registration of the window into the layer file. In addition, an action is registered in the layer file, with which the user opens the window.

Detailed information about this and other topics relating to the design of the user interface can be found in Chapter 5.

Summary

In this chapter, you were assumed to be an Eclipse user. You were introduced to the NetBeans IDE as a tool, and to the NetBeans Platform as a desktop framework. We started by looking at the most commonly used functions and windows in Eclipse and showing where they can be used in the NetBeans IDE. You learned how to adapt the keymap to Eclipse settings. Next, we compared Eclipse plugins to NetBeans modules. We compared the terminology of two platforms, and you learned about their major similarities and differences.

CHAPTER 17

Tips and Tricks
Let's Get Some Cool Expert Tips!

This chapter looks at several helpful interfaces, classes, and concepts of the NetBeans Platform. Some of the JDK's newest features are also demonstrated in action on the NetBeans Platform, such as the `SwingWorker` class.

Asynchronous Initialization of Graphic Components

When developing GUIs, it is important to maintain fast response time throughout the component lifecycle. This is especially true for the phase when components are initialized. A good example of this is with wizards, as discussed in Chapter 8. If the user starts a wizard, the wizard should open and be available immediately. Sometimes data for components needs to load from a relatively slow data source or must be calculated from dependent data. In this case, initialize your components in a separate thread asynchronously for all initializations in the user interface. When doing so, take care to not access GUI components from outside the event dispatch thread (EDT).

The NetBeans Utilities API provides an easy way to meet this requirement: the service provider interface `AsyncGUIJob`. This interface specifies two methods to help initialize components asynchronously. The `construct()` method is executed automatically in a separate thread, so the EDT is not blocked. This lets you load data or perform other long-running initializations without performance being affected. Do not access GUI components in the `construct()` method, however. Rather, as soon as the `construct()` method has returned, the `finished()` method is called, within the EDT. Here, you can add data previously loaded in the `construct()` method.

In the example in Listing 17-1, data is added (loaded in `construct()`) to a `DefaultComboBoxModel`. After loading, you add the created data model to the `JComboBox` within the `finished()` method. This asynchronous process is started and connected to the component using the method `Utilities.attachInitJob()`. This way, a number of components are defined and started independently.

Listing 17-1. *Asynchronously initializing graphical components using the AsyncGUIJob interface*

```
public final class AsynchTopComponent extends TopComponent {
   private JComboBox items = new JComboBox(new String[] { "Loading..." });
   private DefaultComboBoxModel m = new DefaultComboBoxModel();
   private AsynchTopComponent() {
      initComponents();
      Utilities.attachInitJob(items, new AsyncGUIJob(){
         public void construct() {
            // long-lasting loading of data
            for(int i = 0; i < 20; i++) {
               Thread.sleep(200);
               m.addElement(new String("Item " + i));
            }
         }
         public void finished() {
            items.setModel(m);
         }
      });
   }
}
```

Another possibility for asynchronously initializing GUI components is the `SwingWorker` class, which became part of the standard Java API in version 6. It is an abstract class, initializing components in almost the same way as via the `AsyncGUIJob` interface. Using the `SwingWorker` class, the previous example with `AsyncGUIJob` looks like Listing 17-2.

Listing 17-2. *Asynchronously initializing graphic components using the SwingWorker class*

```
SwingWorker<DefaultComboBoxModel, String> worker =
   new SwingWorker<DefaultComboBoxModel, String>() {
   protected DefaultComboBoxModel doInBackground()
      throws Exception {
      // long-lasting loading of data
      for(int i = 0; i < 20; i++) {
         Thread.sleep(200);
         m.addElement(new String("Item " + i));
      }
      return m;
   }
   protected void done() {
      try {
         items.setModel(get());
      } catch (Exception ignore) {}
   }
};
worker.execute();
```

Similar to the `construct()` method, data is created (or loaded) within the method `doInBackground()`. The difference occurs when passing the created data as a return value of the function (see Listing 17-3). The return type is defined by the first template of the `SwingWorker` class—in this example, `DefaultComboBoxModel`. This method is also executed outside the EDT.

The done() method is the counterpart to the finished() method, which is called from within the EDT as soon as the doInBackground() method has finished. Using the get() method, we receive data prepared by doInBackground().

Other very useful features of the SwingWorker class are the publish() and process() methods. By using publish(), data can be sent from the asynchronously executed doInBackground() method to the EDT that is processed by calling process().

Listing 17-3. *Publishing and processing with the SwingWorker class*

```
items.setModel(m);
SwingWorker<DefaultComboBoxModel, String> worker =
   new SwingWorker<DefaultComboBoxModel, String>() {
   protected DefaultComboBoxModel doInBackground()
      throws Exception {
      for(int i = 0; i < 20; i++) {
         Thread.sleep(200);
         publish(new String("Item " + i));
      }
      return m;
   }
   protected void process(List<String> chunks) {
      m.addElement(chunks.iterator().next());
   }
};
worker.execute();
```

Rather than setting the data model in the done() method, the elements are added immediately. In the doInBackground() method, single entries are immediately sent to the EDT using publish(). Those entries are received with the process() method and inserted into the combo box, so they appear right away. The parameter types publish() and process() are defined in the second template of the SwingWorker class.

Undo/Redo

TopComponents and multiview elements provide undo/redo-functionality for the user. This functionality is specified by the UndoRedo interface, implemented by the UndoRedo.Manager class. It handles the undo and redo actions provided by the NetBeans Platform in the Edit menu and toolbar. This manager derives from the class UndoManager in the Java API, which administrates changes liable to being undone or restored. An instance of this manager is retrieved by calling the getUndoRedo() function.

Events added to the manager are strongly dependent on context. The interface for those events is specified by UndoableEdit. Java already provides abstract classes for this interface. The class AbstractUndoableEdit, for example, provides a standard implementation for all methods, limiting override to only the classes needing special implementation. The StateEdit class and the related StateEditable class are very handy in this context as well.

The StateEditable interface is implemented by objects whose data may be changed by users. An example of this would be a DataObject class representing an MP3 file whose ID3 information requires a change by the user.

The example in Listing 17-4 demonstrates this principle with a very simple class that has one attribute in a text field that is editable by the user. The `UndoRedo.Manager` is held as a private data element, retrieved by calling the `getUndoRedo()` method. The `TopComponent` has two buttons: one to read the attribute from the data object, the other to save the data from the text field.

If a change is performed, a `StateEdit` object is created that implements the `UndoableEdit` interface. This object needs an instance of the `StateEditable` interface. This, in turn, is the data object. The `UndoableEdit` instance is passed to the manager by calling the `undoableEditHappened()` method that updates all listeners. That way, all platform Undo and Redo buttons are activated or deactivated automatically. Now all changes can be applied to the data object, and the event is ended by the `end()` method.

Listing 17-4. *Providing an undo/redo manager and adding an element when data is changed by the user*

```
public class MyTopComponent extends TopComponent {
   private UndoRedo.Manager manager = new UndoRedo.Manager();
   private MyObject obj = new MyObject();
   public UndoRedo getUndoRedo() {
      return manager;
   }
   private void loadActionPerformed(ActionEvent evt) {
      textField.setText(obj.getProp());
   }
   private void saveActionPerformed(ActionEvent evt) {
      StateEdit edit = new StateEdit(obj);
      manager.undoableEditHappened(new UndoableEditEvent(obj, edit));
      obj.setProp(textField.getText());
      edit.end();
   }
}
```

The data object needing changes undone or restored implements the `StateEditable` interface. This interface specifies the methods `storeState()` and `restoreState()`. The principle of the `StateEdit` class is based on storing data object attributes to a `Hashtable` object. This hashtable is managed by the `StateEdit` object. The `storeState()` method is called when the `StateEdit` object is created. Attributes are stored to the hashtable (that is, passed) before changes are applied.

To undo changes, the `StateEdit` object calls the `restoreState()` method, and a hashtable that holds the original values is passed as a parameter. Those values need only be read and applied as shown in Listing 17-5.

Listing 17-5. *Data object whose changed attributes need to be restored*

```
public class MyObject implements StateEditable {
   private String prop = new String("init value");
   public void storeState(Hashtable<Object, Object> props) {
      props.put("prop", prop); // save original state
   }
   public void restoreState(Hashtable<?, ?> props) {
```

```
        prop = (String)props.get("prop"); // read original state
    }
    public void setProp(String value) {
        prop = value;
    }
    public String getProp() {
        return prop;
    }
}
```

Finally, we'll demonstrate how easy it is to add undo/redo functionality to a text component. Text components especially need this feature often. All subclasses of `JTextComponent` (e.g. `JEditorPane`, `JTextArea` und `JTextField`) use a `Document` as a data model.

An `UndoableEditListener` can be added to a `Document` instance by calling the `addUndoableEditListener()` method. This listener interface is implemented by the NetBeans `UndoRedo.Manager`. This manager, previously stored in the `TopComponent` and returned by the `getUndoRedo()` method, is added to a `Document` instance as a listener. By appending a single line of code, you can add undo/redo functionality to a text component:

```
textField.getDocument().addUndoableEditListener(manager);
```

Now the text component is able to report its events to the manager, automatically activating or deactivating the Undo and Redo buttons. You can add undo support to a text component, as well as any component whose data model implements the `Document` interface or uses an implementation of `Document`, as the `HTMLDocument` and `PlainDocument` classes do.

Ending an Application's Lifecycle

When a NetBeans Platform application is shut down, all user-specific settings (such as the information about open `TopComponent`s, application window size, and toolbars) are saved to the application user directory. In addition, all modules that implement a module installer (see Chapter 3) are asked if the application can be shut down. Thus, an application is not only closed, but it is shut down properly. Usually, an application is closed using the menu or the Close button in the title bar. In some cases, you might close an application programmatically. This could be an option if wrong data is entered in a login dialog, and the application should then be closed. In this case, you must not or cannot—as usual in Java applications—close the application using `System.exit()`. The process for shutting down an application is specified by the Utilities API in the global service `LifecycleManager`. The NetBeans `Core` module offers a service provider for that purpose, responsible for executing the tasks mentioned earlier. This standard implementation of the `LifecycleManager` can be obtained by calling the `getDefault()` method. Close an application by calling the following line:

```
LifecycleManager.getDefault().exit();
```

Since this `LifecycleManager` is implemented as a service, you can provide your own implementation of this abstract class. This does not mean that the standard implementation of the NetBeans Platform is no longer available—you simply need to call it. This way, it is possible to execute custom tasks before an application is closed. Listing 17-6 demonstrates how to call the standard implementation after executing custom tasks, and shut down applications properly.

Listing 17-6. *A custom LifecycleManager implementation, which calls the standard implementation*

```
public class MyLifecycleManager extends LifecycleManager {
   public void saveAll() {
      for(LifecycleManager manager :
              Lookup.getDefault().lookupAll(LifecycleManager.class)) {
         if(manager != this) { /* only call the Core Manager */
            manager.saveAll();
         }
      }
   }
   public void exit() {
      // perform application-specific shutdown tasks
      for(LifecycleManager manager :
              Lookup.getDefault().lookupAll(LifecycleManager.class)) {
         if(manager != this) { /* only call the Core Manager */
            manager.exit();
         }
      }
   }
}
```

This implementation must be registered as a service provider. It's important to note that a position must be declared to ensure that the custom implementation is delivered by the Lookup and called first. The `LifecycleManager` would be called only if this is not done. The class is registered in the `META-INF/services` folder (see Chapter 6), in the file `org.openide.LifecycleManager`, which contains the following two lines:

```
com.galileo.netbeans.module.MyLifecycleManager
#position=1
```

WarmUp Tasks

The NetBeans Platform offers an extension point named `WarmUp`, for executing asynchronous tasks when starting applications:

```
<folder name="WarmUp">
   <file name="com-galileo-netbeans-module-MyWarmUpTask.instance"/>
</folder>
```

You can add any number of instances (that implement the `Runnable` interface) to this extension point in the layer file:

```
public class MyWarmUpTask implements Runnable {
   public void run() {
      // do something on application startup
   }
}
```

Critical tasks—for example, tasks that are necessary as module-starting conditions—must not be started here. These tasks are executed asynchronously at the start of applications, which

means there is no guarantee about when the task is started or finished. In this case, a module installer should be used (see Chapter 3).

System Tray

Java includes enhanced desktop integration in version 6 and provides access to the system tray of the underlying operating system. You can add one or more icons with a context menu or double-click action. A good way to do this for a NetBeans application is with the `restored()` method of the module installer (see Chapter 3). First, check whether the operating system has a system tray. If so, gain access using the `getSystemTray()` method. In order to add a context menu, create a `PopupMenu` whose actions are defined via an extension point in the layer file. Thus, actions are added to the tray icon from different modules. The extension point used is called `TrayMenu`, whose values are read using a Lookup. The registered actions only need to implement the `Action` interface, which means NetBeans Platform action classes may also be used. After creating the context menu, pass the menu, an icon, and a tooltip to a `TrayIcon` object and add it to the system tray, as shown in Listing 17-7.

Listing 17-7. *Adding a system tray icon whose context menu is built using a layer file*

```
public void restored() {
   if (SystemTray.isSupported()) {
      SystemTray tray = SystemTray.getSystemTray();
      PopupMenu popup = new PopupMenu();
      popup.setFont(new Font("Arial", Font.PLAIN, 11));
      for(Action a : Lookups.forPath("TrayMenu").lookupAll(Action.class)) {
         MenuItem item = new MenuItem((String)a.getValue(Action.NAME));
         item.addActionListener(a);
         popup.add(item);
      }
      Image img = ImageUtilities.loadImage("com/galileo/netbeans/module/icon.gif");
      TrayIcon trayIcon = new TrayIcon(img, "My Tray Menu", popup);
      trayIcon.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            System.out.println("double click on tray icon");
         }
      });
      try {
         tray.add(trayIcon);
      } catch (AWTException e) {
         System.err.println(e);
      }
   }
}
```

Desktop

The `Desktop` class in Java 6 allows execution of standard applications like an Internet browser or an e-mail client. Pass a `File` or `URI` object to the methods provided by the `Desktop` class. On

the basis of these objects, an associated standard application is launched. For example, if `Desktop.open(new File("myfile.pdf"))` is executed, Acrobat Reader is started (if this is the standard application for `.pdf` files). Table 17-1 shows all methods of the `Desktop` class.

Table 17-1. *Methods in the Desktop class*

Method	Function
`isDesktopSupported()`	Checks for `Desktop` class support on the operating system.
`isSupported(Desktop.Action a)`	Checks if actions like `BROWSE`, `OPEN`, `EDIT`, `PRINT`, and `MAIL` are available.
`getDesktop()`	Used to get an instance of the `Desktop` class. This method throws an `UnsupportedOperationException` if the `Desktop` class is not supported.
`browse(URI uri)`	Opens the given URI in the file browser.
`open(File file)`	Opens the file in the associated program (or in a file browser, if it is a folder).
`edit(File file)`	Opens the file in the standard editor for this file type.
`print(File file)`	Sends the file directly to the printer using the standard file application print functionality.
`mail()`	Opens the e-mail edit window.
`mail(URI uri)`	Opens the e-mail edit window where the mailto field is filled with the e-mail address from URI.

Logging

A very important and helpful (but often disregarded) topic is logging. Logging is the practice of recording status, warning, and error messages. Logging in the NetBeans Platform is based on the Java Logging API.

Logger

Log output is recorded by the Logging API using a `Logger` object. Typically, different components have different `Logger` instances. You get an instance of a `Logger` via the factory method `getLogger()`.You can also use a global logger, but you should use a named, component-specific logger whenever possible. This way, different loggers can be turned on or off, which is very helpful when searching for bugs. A named logger is obtained by the following:

```
Logger log = Logger.getLogger(MyClass.class.getName());
```

Typically, the full name of the class that creates the log output is used as the name for the logger. This name is obtained from the `getName()` method. If a logger already exists for this name, it is returned. The global logger can be obtained using the name `Logger.GLOBAL_LOGGER_NAME`.

Record log output (of a defined level) using the `log()` methods in the `Logger` class. The following log levels are provided in the `Level` class: `FINEST`, `FINER`, `FINE`, `CONFIG`, `INFO`, `WARNING`, and `SEVERE`. The methods `finest()`, `finer()`, `fine()`, `config()`, `info()`, `warning()`, and `severe()` are also provided; these record the given message at the declared level.

LogManager

The Java Logging API specifies a central `LogManager`. This manager controls a hierarchical namespace holding all named loggers. That's why it's reasonable to use the full names of classes (that hold the hierarchical package structure) for logger names. For access to this manager, use the following:

```
LogManager manager = LogManager.getDefault();
```

The `LogManager` provides all names of all loggers as well as the name of a NetBeans Platform logger whose level may be changed for debugging purposes. A list of all loggers can be retrieved as follows:

```
LogManager manager = LogManager.getLogManager();
for(String name : Collections.list(manager.getLoggerNames())) {
   System.out.println(name);
}
```

Configuration

The manager also administers configuration files, which are initially loaded from the `lib/logging.properties` file in the JRE folder. Define configuration files by setting them to the system property `java.util.logging.config.file`. Configuration data may be loaded from a database. Implement a class that reads the data from a database and register it to the system property `java.util.logging.config.class`. Registration causes it to be automatically instantiated. Within this class, provide the configuration data for the `LogManager` via an `InputStream` for the `readConfiguration(InputStream)` method in the `LogManager`.

Register `Handler` implementations in the configuration file so they output log data to the console (`ConsoleHandler`) or into a file (`FileHandler`). You can register implementations like the handler from the NetBeans Platform that displays log messages graphically. The logging system comes with a root logger. All other loggers forward their logs to this root logger. For this root logger, register a handler with the following property:

```
handlers = java.util.logging.ConsoleHandler
```

Multiple handlers can be listed using commas. To disable forwarding logs to the root logger, do so by using the following:

```
<logger name>.useParentHandlers = false
```

Define a handler especially for this logger in order to obtain log output:

```
<logger name>.handlers = java.util.logging.FileHandler
```

Setting the log level is important in the configuration. A log level defines which kind of logs are recorded. For example, set the log level globally as follows to hide simple status messages but show warning or error messages when debugging:

```
.level = WARNING
```

Or overwrite a single logger's log level by using its name as a prefix:

```
<logger name>.level = INFO
```

Configuration data is not only set in the configuration file, but also as system properties. Set it at runtime using the `System.setProperty()` method. It is important to call the `LogManager`'s `readConfiguration()` method in order to apply the new configuration data. Alternatively, select the configuration at the application's startup using command-line parameters. During development in NetBeans, set your NetBeans Platform application start parameters in the project properties file (under `Important Files`) using the property `run.args.extra`:

```
run.args.extra = -J-Dcom.galileo.netbeans.myclass.level=INFO
```

For distribution of your application, set command-line parameters using the property `default_options` in the `etc/<application>.conf` file.

Error Reports

The NetBeans Platform implements and registers a special log handler that displays recorded error messages for the user in a dialog. Therefore, use either the `SEVERE` or `WARNING` log level, and pass the `Exception` directly to the `log()` method.

```
Logger logger = Logger.getLogger(MyClass.class.getName());
try {
   ...
} catch(Exception e) {
   logger.log(Level.SEVERE, null, e);
   // or
   logger.log(Level.WARNING, null, e);
}
```

Summary

This chapter provided a range of useful tips. We first looked at an approach for initializing GUI components asynchronously on the EDT. In this context, you also learned how to use the `SwingWorker` class, which is part of JDK 6.

Another tip covered undo/redo functionality in several components in a NetBeans Platform application. You also saw how to execute tasks when the application shuts down, as well as how to execute long-running tasks asynchronously during the application's startup process.

In the final sections, we looked at the new JDK 6 `SystemTray` and `Desktop` classes, as well as the NetBeans Platform logging facilities.

CHAPTER 18

Example: MP3 Manager
Let's Put It All Together!

Now, as we know almost everything about the most important aspects of the NetBeans Platform in detail, we will implement a full-featured example step by step. Most of what has been learned will be incorporated into it.

The purpose of this chapter is to demonstrate the design and implementation of an executable application, playing MP3 files on the NetBeans Platform. This application will be as flexible and modular as possible. We will reuse much of the previously implemented and handled advantages and features of the NetBeans Platform.

This chapter is useful for those who did not yet read all the previous chapters, as well as those trying to dive directly into the NetBeans Platform world. Some parts of the chapter requiring those instructions contain references to the applicable chapters. The following pages will cover only the most important parts of the implementation.

The complete example can be downloaded from the Apress web site (`www.apress.com`).

Design

Essentially, the application should be able to play MP3 files, manage these files in playlists, and display the relevant ID3 information. In addition, it will provide support for simple editing functionality, and also let the user add ID3 information. The `Favorites` module of the NetBeans Platform is used as base for the MP3 library. Using the `Palette` module, we can manage whole MP3 albums. Also, the `Output` module can be useful, giving feedback to the user while processing ID3 jobs. One of the main advantages of the application is its easy extensibility, due to the module-based architecture of the NetBeans Platform. A well-designed architecture and structure for developed applications is required as well. We must think about how much functionality we provide and how many modules are required to do the job. To provide proper interfaces and extension points, we also have to think about where and how the application needs be most extensible.

Figure 18-1 clarifies the design of the application, including the NetBeans Platform, as well as the underlying Java Runtime Environment (JRE). The application portion is marked with "MP3 Manager" in Figure 18-1, and can be roughly divided into three layers:

- Modules, which can be seen as integral parts of the application, are collected in the lowest layer. GUIs (like the toolbar for navigation), implementations of the Java Sound SPI for MP3 files, and Java DB databases (used by the application for several purposes) belong to this layer.
- The middleware layer encapsulates service interfaces, providing us a decoupling of application components, as these no longer depend on each other's implementation, but in most cases only on the provided interfaces.
- The third layer, based on the second, implements the application's components, providing actual functionality, using the independent modules (where applicable).

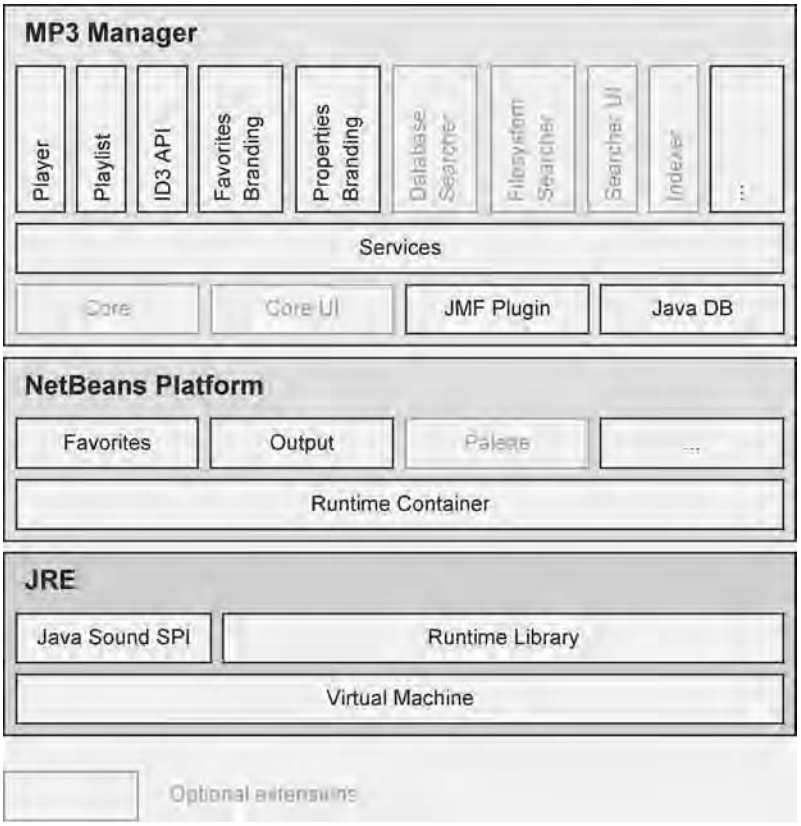


Figure 18-1. *Partitioning of application components into modules*

All modules and their respective responsibilities are listed below. We will implement them, step by step, now.

`Core`: This module may be used to contain all components specifically required for proper use of the application.

`Core UI`: Additional user interface parts are encapsulated by this module.

`JMF Plugin`: The Java Media Framework (JMF) and the MP3 implementation of the Java

Sound SPI (provided by Sun Microsystems as a JAR archive) are required to play back audio files encoded as MP3. This module encapsulates these functionalities.

`Java DB`: Using this module, we access the Java DB system.

`Services`: Services provided by the application should be dynamically extensible. To achieve this, we define and bundle service interfaces into a module.

`Player`: This module provides an MP3 player with a GUI.

`Playlist`: This module provides MP3 file management by manipulating file and player lists.

`ID3 API`: This module contains an API to read and write ID3 information for MP3 files.

`Favorites Branding`: This module customizes module entries of the `Favorites` module.

`Properties Branding`: This module customizes module entries of the `Properties` module.

Creating the NetBeans Platform Application

Every NetBeans rich client application is based on a NetBeans Platform Application (or Module Suite), representing the whole application and containing several modules. Application branding (naming of the application, splash screens, and so on) is also provided out of the box. You can easily create a NetBeans Platform Application by calling File ➤ New Project in the NetBeans IDE, followed by selecting the NetBeans Platform Application in the category NetBeans Modules. Click Next and enter a project name, in this case MP3 Manager. The NetBeans Platform Application is created the moment Finish is clicked.

The application contains only those modules related to the NetBeans Platform. If IDE modules are required by the application later on, they can be added by using the Libraries category in the Project Properties dialog. In the Project Properties dialog, select the Build tab to choose another icon. Define the splash screen under Build ➤ Splash Screen. Click OK, and the foundation of the application will be created.

Next, we will proceed with the application components: NetBeans modules.

Support for MP3

To support playing MP3 files in the `Player` module, we use JMF and an MP3 plugin. Like JMF, the plugin is created by Sun Microsystems and represents an implementation of the Java Sound SPI (contained in the JSE) for MP3 files. The plugin is sufficient to play back MP3s, but using JMF simplifies further implementations, and we shouldn't pass up this chance to use it. Both components (JMF and the MP3 plugin) will be included in a library wrapper module that integrates them into the application.

Creating the JMF Module

Download JMF as a cross-platform edition, as well as the MP3 plugin from Sun Microsystems' official web site, at `http://java.sun.com/products/java-media/jmf`. The JAR archives `lib/jmf.jar` in the JMF distribution and `lib/ext/mp3plugin.jar` in the MP3 plugin distribution

need to be copied to a directory now, so we can add them to a module. Therefore, click File ➤ New Project ➤ NetBeans Modules ➤ Library Wrapper Module. Use the Browse button to add the copied JAR archives (hold the Ctrl key to select both). Enter **JMF Plugin** as the project name on the next wizard page and select the previously created Platform Application (Module Suite). The following page defines the code name base of the modules. Please use `javax.media` here. Click the Finish button to create the wrapper module and to add it to the MP3 Manager.

Registering the MP3 Plugin

By design, the MP3 plugin is not immediately usable by JMF. We must register it as the JMF plugin manager. For our purposes, it would be good to ensure that the plugin is always registered at application startup, using a module installer (see Chapter 3), which is executed during loading of the JMF plugin. In order to create such a module installer, switch to the `Source Packages` folder of the MP3 plugin in the Projects window, and right-click to open the context menu. Then select New ➤ Module Installer, and click the Finish button in the dialog box to create the installer. The `restored()` method is used to implement the required registration to the `PlugInManager`. Specify the plugin class, the input and output format, and the plugin type. In our case, it is a codec plugin, so we insert the lines shown in Listing 18-1.

Listing 18-1. *Registering the MP3 plugin at the JMF plugin manager during startup*

```
package javax.media;
import javax.media.format.AudioFormat;
import org.openide.modules.ModuleInstall;
public class Installer extends ModuleInstall {
   public void restored() {
      Format input1 = new AudioFormat(AudioFormat.MPEGLAYER3);
      Format input2 = new AudioFormat(AudioFormat.MPEG);
      Format output = new AudioFormat(AudioFormat.LINEAR);
      PlugInManager.addPlugIn(
              "com.sun.media.codec.audio.mp3.JavaDecoder",
              new Format[]{input1, input2},
              new Format[]{output},
              PlugInManager.CODEC);
   }
}
```

MP3 File Type

Another important issue for easy-to-use and professional management of MP3 files in our application is an MP3 file type infrastructure. The file type infrastructure of the NetBeans Platform is used to manage files of a particular type. This infrastructure consists of three main parts. First, we have the `FileObject`, which wraps a `File` object, representing the actual MP3 file. Based on this, there is a `DataObject`, which extends the `FileObject` by flexible properties and functionalities. Finally, a `Node` object is used, representing a `DataObject` in the user interface that includes the ability to accept actions. More information relating to this can be found in Chapter 7.

Normally, the MP3 file type belongs to the core functionality of the MP3 Manager, which means we could manage it in the `Core` module. But for flexible usage and to avoid cyclic

dependencies, we will create a separate module for it. This can be done by invoking File ➤ New Project ➤ NetBeans Modules ➤ Module. You can use `File Type` as the name, and, e.g., `com.hboeck.mp3manager.filetype`, as the code name base. All other values remain as they are. Clicking the Finish button closes the wizard and creates the module.

All components of a file type are created completely by the wizard provided by the IDE. It can be brought up from the context menu of the `File Type` module using New ➤ File Type. The MIME type for MP3 files is `audio/mpeg`, and the extension is `mp3`. On the next page, we prefix the created class with `Mp3` and define an icon for this file type. Now all required information is collected, and the MP3 file type can be created by clicking Finish.

An instance named `Mp3DataLoader`, registered to the MIME type of `MP3` and used to load the `Mp3DataObject`, will be registered by the wizard (see Listing 18-2).

Listing 18-2. *Registration of the default DataObject factory to load Mp3DataObjects*

```
<folder name="Loaders">
   <folder name="audio">
      <folder name="mpeg">
         <folder name="Factories">
            <file name="Mp3DataLoader.instance">
               <attr name="SystemFileSystem.icon"
                       urlvalue="nbresloc:/com/hboeck/mp3manager/filetype/mp3.png"/>
               <attr name="dataObjectClass"
                       stringvalue="com.hboeck.mp3manager.filetype.Mp3DataObject"/>
               <attr name="instanceCreate"
                       methodvalue="org.openide.loaders.DataLoaderPool.factory"/>
               <attr name="mimeType" stringvalue="audio/mpeg"/>
            </file>
         </folder>
      </folder>
   </folder>
</folder>
```

For any `FileObject` of type `MP3`, this factory creates an `Mp3DataObject`, which consists of the skeletal structure in Listing 18-3.

Listing 18-3. *The class Mp3DataObject implements the logic of an MP3 file.*

```
import org.openide.filesystems.FileObject;
import org.openide.loaders.DataNode;
import org.openide.loaders.DataObjectExistsException;
import org.openide.loaders.MultiDataObject;
import org.openide.loaders.MultiFileLoader;
import org.openide.nodes.Node;
import org.openide.nodes.Children;
import org.openide.util.Lookup;
public class Mp3DataObject extends MultiDataObject {
   public Mp3DataObject(FileObject pf, MultiFileLoader loader)
      throws DataObjectExistsException, IOException {
      super(pf, loader);
   }
```

```
    @Override
    protected Node createNodeDelegate() {
        return new DataNode(this, Children.LEAF, getLookup());
    }
    @Override
    public Lookup getLookup() {
        return getCookieSet().getLookup();
    }
}
```

This class puts logic into a generic `FileObject` and provides a `Node` object, useful for easily presenting the MP3 file to several views, like the `Favorites` module, or in a playlist (which we will implement next).

Automatically generated classes like those just shown are just the skeleton for what we will develop on the following pages. We will extend these classes with additional functionality if required.

SPECIFYING PUBLIC PACKAGES

When we implement the first module, which uses the MP3 file type, we need to set a dependency on the `File Type` module. Then you will notice that it is not accessible. This is caused by NetBeans defining all packages of a module (by default) as not `public`. Therefore, we must explicitly define which packages can be accessed from outside. We do so in the Properties dialog of a module using the API Versioning category. The module is only shown in the list if at least one package is defined as `public` and contains modules that another module depends on.

ID3 Support

Inside an MP3 file, information about the file exists in the ID3 tag. Currently, two different versions are in use. The ID3v1 tag uses a fixed number of fields (e.g., artist and title), where each has a fixed size. The most important information is stored in the file with that tag. With the ID3v2 tag, a more flexible concept is introduced: more standardized fields are defined, and further customized fields may be added (a field is referred to as *frame*). Nonetheless, these fields can be read by applications that do not know about the fields. A frame of an ID3v2 tag may vary in length. Also, a frame will only exist if it is required, which means there are no empty frames.

ID3 API

We will reuse information stored in this manner in our application by using an API supporting retrieval and storage of ID3 data, according the specification. The Internet provides many such APIs for free. Most of them are reasonably useful, but we created one of our own, aiming at easy handling and integration of personal requirements. For reuse in other applications, we intentionally avoided using NetBeans APIs. Although this library is still under development (only the editing of ID3v1 tags is possible at the moment), it is sufficient for this example, which merely demonstrates advantages and strengths of the NetBeans Platform. Of course, you are free to

use another library. If you do, however, you then must adapt resulting source locations to the selected API.

As with the JMF libraries, we put the ID3 library into a library wrapper module. It is created by clicking File ➤ New Project and NetBeans Modules ➤ Library Wrapper Module. On the wizard's first page, select the library named `com-hboeck-mp3manager-id3.jar` (which can be downloaded from the book's page on the Apress web site, at `www.apress.com`). Name the module `ID3 API` and add it to the MP3 manager. All other fields can remain with their default values.

The ID3 API is available to the `File Type` module only if a dependency to it is defined. We do this using the Libraries category in the Properties dialog of the `File Type` module (see Figure 18-2). Click `Add Dependency` and select the `ID3 API` module.

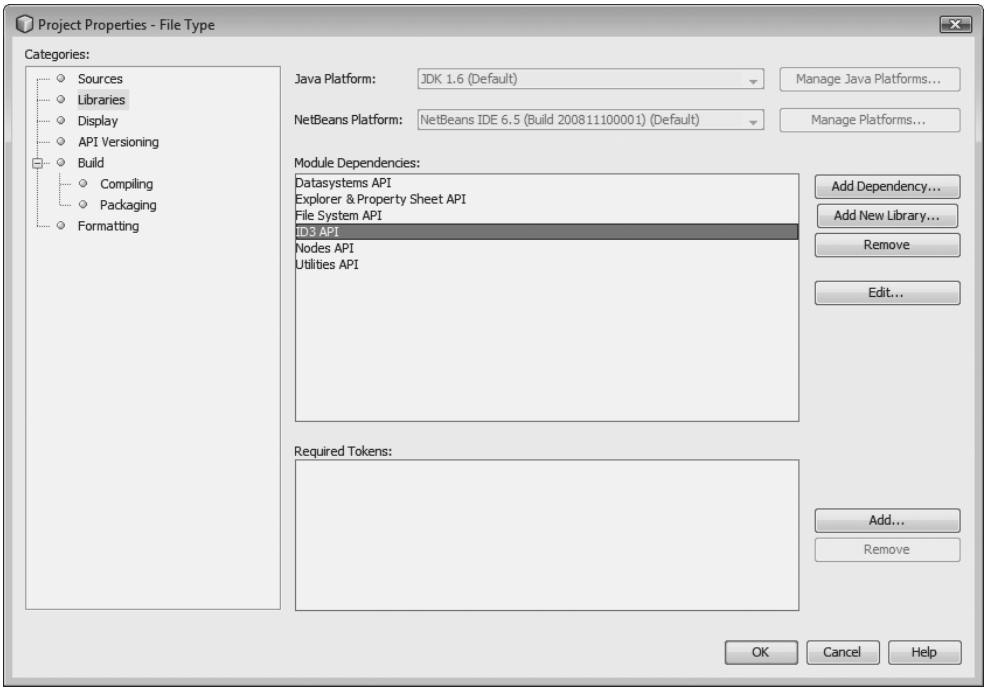


Figure 18-2. *Defining dependencies to the ID3 API*

As mentioned before, the class `Mp3DataObject` is responsible for the information and methods specific to MP3. Extending this class with two methods provides us access to the ID3v1 as well as the ID3v2 tags (see Listing 18-4). It is highly important to create the tags only when accessing the file itself. Imagine creating folders with a lot of MP3 files in the Favorites window. For each of these file, an individual `Mp3DataObject` would be created. If we read the ID3 tags of every file, we would eventually encounter a measurable delay—a behavior to be avoided.

Listing 18-4. *Extending the Mp3DataObject class with support for ID3*

```
import com.hboeck.mp3manager.id3.v1.ID3v1Tag;
import com.hboeck.mp3manager.id3.v2.ID3v2Tag;
```

```
public class Mp3DataObject extends MultiDataObject {
   private ID3v1Tag id3v1 = null;
   private ID3v2Tag id3v2 = null;
   public Mp3DataObject(FileObject pf, MultiFileLoader loader)
           throws DataObjectExistsException, IOException {
      super(pf, loader);
   }
   ...
   public ID3v1Tag getID3v1Tag() {
      if(id3v1 == null) {
         id3v1 = new ID3v1Tag(FileUtil.toFile(getPrimaryFile()));
      }
      return id3v1;
   }
   public ID3v2Tag getID3v2Tag() {
      if(id3v2 == null) {
         id3v2 = new ID3v2Tag(FileUtil.toFile(getPrimaryFile()));
      }
      return id3v2;
   }
}
```

Using the `DataObject` method `getPrimaryFile()` returns the `FileObject` of the MP3 file, which is managed by the `Mp3DataObject` instance. We must pass a `FileObject` to the ID3 tag constructor. This file is obtained by using the method `FileUtil.toFile()`, which determines the file encapsulated by the `FileObject`.

Another way to obtain instances of the `ID3v1Tag` and `ID3v2Tag` classes is to provide them using the Lookup of the `Mp3DataObject`. This enables us to retrieve these instances from a simple `Node` or `DataObject` instance, without special type safety:

```
Node n = ...
ID3v1Tag id3v1 = n.getLookup().lookup(ID3v1Tag.class);
```

ID3 Editor

The next step is to display and edit the ID3 data, as shown in Figure 18-3. The NetBeans Platform ships with a `Properties` module that is useful in combination with the MP3 file type. Providing properties shown in the user interface is the responsibility of nodes. Rather than a generic `DataNode` instance (see Listing 18-3, shown earlier), we will create our own node class. This class is named `Mp3DataNode` and overrides the `createSheet()` method (see Listing 18-5). This method provides the properties of nodes using a `Sheet` object. Usually, properties are only readable, but some are also writable. We have such a case: ID3v1 data must be both readable and writable, but ID3v2 data must be readable only.

First, we invoke the `createSheet()` method of the `DataNode` superclass, which creates a default `Sheet` object containing base properties, such as file name and size, as well as the date of last modification. If you do not want this data displayed, you can create your own sheet using `Sheet.createDefault()`. Inside a `Sheet` object, properties are grouped using `Set` objects, which may be hidden or shown in the Properties window. A user selects whether these groups will be displayed or not. The static method `createPropertiesSet()` creates just such a grouping set. We create two of them, to manage the ID3v1 and ID3v2 data separately. Each set should be

given a unique name, using the method `setName()`; otherwise, the sets will be overridden inside the sheets.

Listing 18-5. *Providing ID3 information in a properties sheet for display and edit purposes*

```
import com.hboeck.mp3manager.id3.v1.ID3v1Tag;
import com.hboeck.mp3manager.id3.v2.ID3v2Tag;
import org.openide.loaders.DataNode;
import org.openide.nodes.Children;
import org.openide.nodes.PropertySupport;
import org.openide.nodes.Sheet;
import org.openide.util.Lookup;
public class Mp3DataNode extends DataNode {
   public Mp3DataNode(Mp3DataObject obj) {
      super(obj, Children.LEAF);
   }
   public Mp3DataNode(Mp3DataObject obj, Lookup lookup) {
      super(obj, Children.LEAF, lookup);
   }
   @Override
   protected Sheet createSheet() {
      Sheet sheet = super.createSheet();
      Sheet.Set set1 = Sheet.createPropertiesSet();
      Sheet.Set set2 = Sheet.createPropertiesSet();
      set1.setName("id3v1");
      set1.setDisplayName("ID3 V1");
      set2.setName("id3v2");
      set2.setDisplayName("ID3 V2");
      Mp3DataObject m = getLookup().lookup(Mp3DataObject.class);
      ID3v1Tag id3v1 = m.getID3v1Tag();
      ID3v2Tag id3v2 = m.getID3v2Tag();
      try {
         /* ID3v1 Properties */
         Property title1 =
            new PropertySupport.Reflection<String> (id3v1, String.class, "title");
         ...
         title1.setName("Title");
         set1.put(title1);
         /* ID3v2 Properties */
         Property album2 = new PropertySupport.Reflection<String>
            (id3v2, String.class, "getAlbum", null);
         ...
         album2.setName("Album");
         set2.put(album2);
      } catch (Exception e) { }
      sheet.put(set1);
      sheet.put(set2);
      return sheet;
   }
}
```

The Lookup of the `Node` provides us an instance of the `Mp3DataObject` represented by that node. Using the previously created methods `getID3v1Tag()` and `getID3v2Tag()` allows access to the ID3 information of the MP3 file. Next, we create an instance of the class `PropertySupport.Reflection<T>` for every property. With the help of a template, the type of property is specified (in this example, it is `String`). The name of the method for the read/write properties (the means to read and write properties) should not contain the prefixes `get` or `set`. Passing `title` to the constructor sets the title (e.g., to create the methods `getTitle()` and `setTitle()`). Properties that are read-only are passed to a special version of the constructor taking the names of the get and set methods separately. Passing `null` will prevent modification of the property. Each property created is named using the method `setName()`. This name is displayed in the Properties window. Finally, we add each instance to the `Sheet` object using the `put()` method, and return that sheet.

After creating the `Mp3DataNode` class, we need to adapt the `Mp3DataObject` class, which uses a `DataNode` instance (see Listing 18-3, shown previously). Instead of this, we now return an `Mp3DataNode` instance in the `createNodeDelegate()` method:

```
@Override
protected Node createNodeDelegate() {
   return new Mp3DataNode(this, getLookup());
}
```

Executing the application, opening the Favorites and Properties windows using the Window menu, and then adding an MP3 file to the Favorites window will give you the results shown in Figure 18-3.

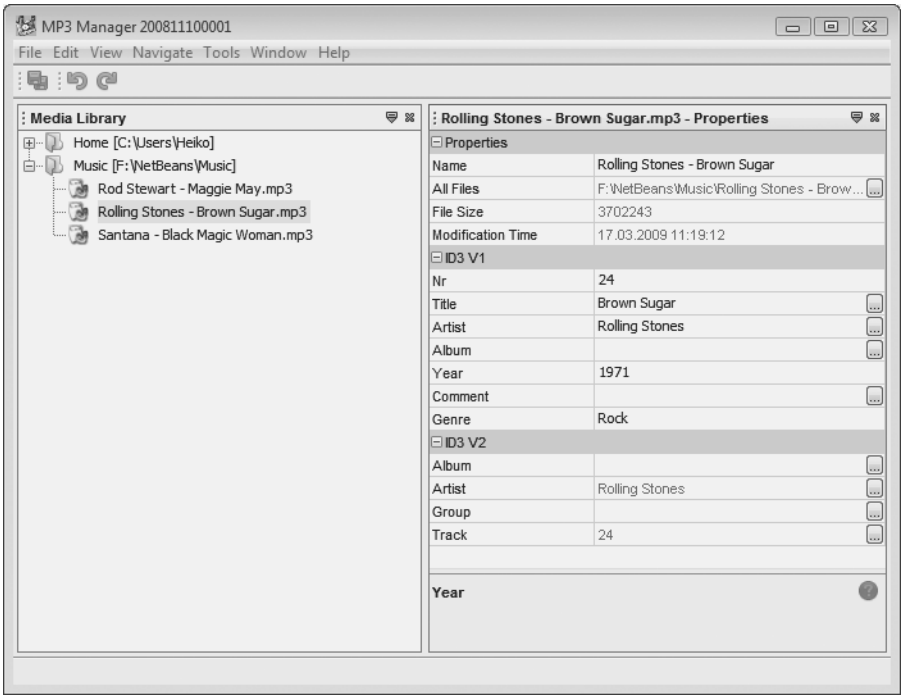


Figure 18-3. *Using the Properties window as ID3 editor*

The "User-Defined Properties Editor" section in Chapter 9 provides an overview for creating and providing special editors (e.g., a combo box providing different values) for properties. This is useful for properties like the genre, which can have several predefined values.

Media Library

In the previous chapter, we made use of the `Favorites` module provided by the NetBeans Platform. We can use this module again for a media library, where the user can add or remove arbitrary files or directories easily. This feature is not restricted to MP3 files, but can be used for any file type. For example, if we use the `Image` module from the `ide` cluster, we can manage and display covers of records in JPG format. Since we can bind actions to a particular MIME type via the layer file, we are able to work with MP3 files directly in the Favorites window. That means we can play back files by merely double-clicking, or using the drag-and-drop feature to push files to a separate window (like a playlist).

The Favorites window can be used only if it is activated in the application. Therefore, we open the properties of the MP3 Manager via the context menu and ensure that the `Favorites` module is activated in the Libraries category under the `platform9` cluster.

Now we can change the name and the menus of the `Favorites` module using a branding module, created via File ➤ New Project ➤ NetBeans Modules ➤ Module. We name it `Favorites Branding` and add it to the MP3 manager. It is vitally important to use the same value for the code name base as given by the original `Favorites` module (`org.netbeans.modules.favorites`) (see Figure 18-4). Additionally, we must fix the `Bundle.properties` file by appending the branding token (`mp3_manager` in our case). As an option, a locale can be appended.

The file `Bundle_mp3_manager.properties` should contain all the entries we want to change from the `Bundle.properties` file of the `Favorites` module. In our case, it's the name of the window as well as several menu entries. So we must add the following entries to our properties file:

```
ACT_Add=&Add to Media Library
ACT_AddOnFavoritesNode=&Add to Media Library...
ACT_Remove=&Remove from Media Library
ACT_View=&Media Library
ACT_Select=Media Library
ACT_Select_Main_Menu=Select in Media Library
Favorites=Media Library
```

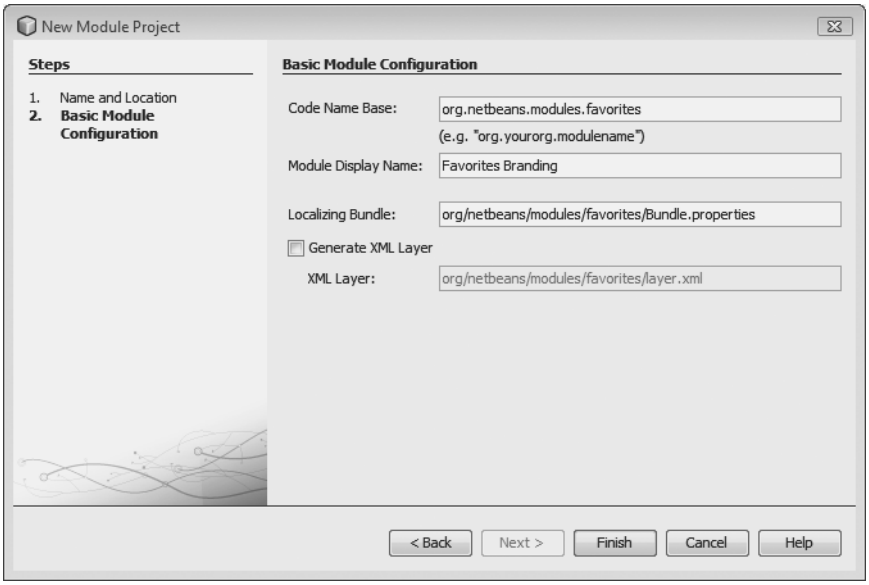


Figure 18-4. *Creating a branding component for the Favorites module*

Finally, we ensure the module is located in the directory `modules/locale` (see the "Customization of Platform Modules" section in Chapter 11). The best way to do this is, of course, automatically. For that, we add the following properties to the project properties file (which can be found in the `Important Files` folder of the `Favorites Branding` module), which overrides default configurations of the properties in the build script:

```
module.jar.dir=modules/locale
module.jar.basename=org-netbeans-modules-favorites_mp3_manager.jar
```

Services

Following on, we will implement the main functionality of the application. It is divided into two sections: the service interface and the service provider. In conjunction with the registry mechanism of the service provider and the Lookup, we can implement functionality that's absolutely decoupled and independent of the specific module. For this purpose, we create a new module to bundle service interfaces for a central provision (see Figure 18-1). From this point of view, the module can be seen as a link between different application modules. As usual, we use File ➤ New Project to create a new module. For convenience, we name it `Services` and set the code name base to `com.hboeck.mp3manager.services`. Even for this module, we have remember to enable public access for the packages to be created (Properties ➤ API Versioning), because only then can we define a dependency to the `Services` module and be able to use its classes.

MP3 Player

In the previous section, we created the base for our player design by dividing the `Services` module into two parts: a service interface and a service provider. Now we'll consider the interfaces the player should provide, as well as other components of our application. These interfaces are described in an abstract class—not an interface—as the player should be seen as a global service, which means that requesting modules are normally interested in a single player instance only. You will notice, in this and the following sections, that this behavior can be ensured much more easily using an abstract class than an interface.

Service Interface

Inside the `Services` module, we create a new package named `player` with an abstract class named `Mp3Player`. Of course, a player must be able to play back, pause, and stop MP3 files. Additionally, a user should be able to mute the playback, control the volume, and see the current playback position and total duration. Beside that, seek functionality would be great. All these desired functionalities of the player are specified in the abstract class in Listing 18-6.

Listing 18-6. *Defining the player's interfaces and providing an implementation using the getDefault() method*

```
package com.hboeck.mp3manager.services.player;
import com.hboeck.mp3manager.filetype.Mp3DataObject;
import org.openide.util.Lookup;
public abstract class Mp3Player {
   public static Mp3Player getDefault() {
      Mp3Player p = Lookup.getDefault().lookup(Mp3Player.class);
      if (p == null) {
         p = new DefaultMp3Player();
      }
      return p;
   }
   public abstract void play(Mp3DataObject mp3);
   public abstract void play(ListIterator<Mp3DataObject> mp3s);
   public abstract void pause();
   public abstract void stop();
   public abstract void previous();
   public abstract void next();
   public abstract void setMute(boolean mute);
   public abstract void setVolume(int volume);
   public abstract int getDuration();
   public abstract int getMediaTime();
   public abstract void setMediaTime(int seconds);
}
```

The most important method for *service requesters*—the modules that want to use the player—is `getDefault()`, which searches for registered `Mp3Player` implementations using the Lookup. If an implementation is found, the Lookup returns an instance of it. If no implementation is found, we nonetheless ensure that a requester never obtains a `null` reference, but always an instance of the `Mp3Player` class. Therefore, we provide a default implementation

inside the abstract `Mp3Player` class, which is named `DefaultMp3Player` and will do—in the simplest case—nothing but telling the user it will do nothing (see Listing 18-7). Another easy and smart solution would be to delegate the MP3 file to an external application.

Listing 18-7. *Providing a default implementation inside the abstract class*

```
public abstract class Mp3Player {
   ...
   private static class DefaultMp3Player extends Mp3Player {
      public void play(Mp3DataObject mp3) {
         Logger.getLogger(Mp3Player.class.getName()).info("not supported");
      }
      public void stop() { }
      ...
   }
}
```

Considering a module using the player, we notice that a module needs to be notified about the events that may occur in the player. A user interface, for example, needs to notice the end of the playback of a file. For this purpose, we define a listener interface named `Mp3PlayerEventListener`, which might look like this in its simplest version:

```
package com.hboeck.mp3manager.services.player;
public interface Mp3PlayerEventListener extends EventListener{
   public void playing(Mp3DataObject mp3);
   public void stopped();
}
```

The functionality to add and remove `Mp3PlayerEventListener`s to the player is implemented directly in the abstract class (see Listing 18-8), so the concrete player implementation does not need to care about this. Additionally, we provide two fire methods for listener interfaces to notify listeners about events.

Listing 18-8. *Methods to manage listeners interested in events*

```
public abstract class Mp3Player {
   ...
   private final Set<Mp3PlayerEventListener> listeners =
      new HashSet<Mp3PlayerEventListener>(1);
   public void addEventListener(Mp3PlayerEventListener l) {
      synchronized (listeners) {
         listeners.add(l);
      }
   }
   public void removeEventListener(Mp3PlayerEventListener l) {
      synchronized (listeners) {
         listeners.remove(l);
      }
   }
   protected final void firePlayEvent(Mp3DataObject mp3) {
      Iterator<Mp3PlayerEventListener> it;
```

```
      synchronized (listeners) {
         it = new HashSet<Mp3PlayerEventListener>(listeners).
            iterator();
      }
      while (it.hasNext()) {
         it.next().playing(mp3);
      }
   }
   protected final void fireStopEvent() {
      Iterator<Mp3PlayerEventListener> it;
      synchronized (listeners) {
         it = new HashSet<Mp3PlayerEventListener>(listeners).
            iterator();
      }
      while (it.hasNext()) {
         it.next().stopped();
      }
   }
}
```

Service Provider

Our first service has been defined, so we can now start implementing the service providers. This means we use the JMF module and its functionality for the playback of MP3 files. This brings us back to the MP3 file type. As indicated previously in Figure 18-1, the MP3 player is implemented by a separate module. We create this module with the name `Player` and the code name base `com.hboeck.mp3manager.player`. Using Properties ➤ Libraries, we define dependencies to the other required modules: `File Type`, `JMF Plugin`, and `Services`.

First, we create an `Mp3PlayerImpl` class, which inherits the service interface `Mp3Player`, and implement its methods by means of JMF (see Listing 18-9). Let's start with the method `play()`, by which an MP3 file is given as `Mp3DataObject`. The central class of JMF is the `Manager` class. It is used to obtain system-dependent resources. This manager creates a `Player` instance for the MP3 file passed as a URL. But before starting that `Player` using the `start()` method, we register a `ControllerListener` so that we'll be informed of the different states of the `Player`.

Listing 18-9. *Implementation of the service provider using JMF*

```
package com.hboeck.mp3manager.player;
import com.hboeck.mp3manager.filetype.Mp3DataObject;
import com.hboeck.mp3manager.services.player.Mp3Player;
import javax.media.ControllerEvent;
import javax.media.ControllerListener;
import javax.media.EndOfMediaEvent;
import javax.media.GainControl;
import javax.media.Manager;
import javax.media.Player;
import javax.media.RealizeCompleteEvent;
import javax.media.Time;
public class Mp3PlayerImpl extends Mp3Player implements ControllerListener {
   private static final Logger LOG =
```

```
        Logger.getLogger(Mp3PlayerImpl.class.getName());
private Player player = null;
private GainControl volumeControl = null;
private int volume = 20;
private boolean mute = false;
private Mp3DataObject mp3 = null;
private Enumeration<Mp3DataObject> list = null;
public Mp3PlayerImpl() {
}
public void play(Mp3DataObject mp3) {
   try {
      this.mp3 = mp3;
      if(player != null) {
         player.stop();
         player.close();
      }
      player = Manager.createPlayer(mp3.getPrimaryFile().getURL());
      player.addControllerListener(this);
      player.start();
   } catch(Exception e) {
      LOG.log(Level.SEVERE, e.getMessage(), e);
   }
}
public void play(ListIterator<Mp3DataObject> mp3s) {
   list = mp3s;
   if(list.hasNext()) {
      play(list.next());
   }
}
public void pause() {
   if(player != null) {
      player.stop();
   }
}
public void stop() {
   if(player != null) {
      fireStopEvent();
      player.stop();
      player.setMediaTime(new Time(0));
      player.close();
   }
}
public void previous() {
   if (list != null && list.hasPrevious()) {
      play(list.previous());
   }
}
public void next() {
   if (list != null && list.hasNext()) {
```

```
            play(list.next());
        }
    }
```

The `ControllerListener` interface defines the `controllerUpdate()` method, which is used to get the current state of the player. We are particularly interested in two states. First, we're interested in the `RealizeCompleteEvent` state of the player, because only if the player is fully initialized can we access the volume control. Thereby we notify listeners about the playback start of the MP3 file, using `firePlayEvent()` (see Listing 18-10). The second state of interest is `EndOfMediaEvent`, which allows us to stop the player, and then reset the current playback position to the beginning. If the `play()` method was provided with a list of MP3 files, we start playback with the next file in the list.

Listing 18-10. *Handling events of the JMF player*

```
public void controllerUpdate(ControllerEvent evt) {
    if (evt instanceof RealizeCompleteEvent) {
        LOG.info("Realized");
        firePlayEvent(mp3);
        volumeControl = player.getGainControl();
        setVolume(volume);
        setMute(mute);
    } else if (evt instanceof EndOfMediaEvent) {
        LOG.info("End of Media");
        stop();
        if(list != null && list.hasNext()) {
            play(list.next());
        } else {
            list = null;
        }
    }
}
```

Finally, we implement the missing control and information methods in our service provider class, as shown in Listing 18-11.

Listing 18-11. *Methods to control volume and playback position*

```
public void setVolume(int volume) {
    this.volume = volume;
    if(volumeControl != null) {
        volumeControl.setLevel((float)(volume/100.0));
    }
}
public void setMute(boolean mute) {
    this.mute = mute;
    if(volumeControl != null) {
        volumeControl.setMute(mute);
    }
}
public int getDuration() {
```

```
        return (int)player.getDuration().getSeconds();
    }
    public int getMediaTime() {
        return (int)player.getMediaTime().getSeconds();
    }
    public void setMediaTime(int seconds) {
        player.setMediaTime(new Time((double)seconds));
    }
}
```

Accessing this implementation of the MP3 player should be done via the `Mp3Player.getDefault()` method. However, we must register the `Mp3PlayerImpl` class to enable this method to find the implementation using the Lookup. This is done by creating a `META-INF/services` folder in the `Source Packages` folder of the `Player` module. In this folder, we add a file with a name reflecting the full class name of the abstract class `Mp3Player`. Now we can insert the fully qualified class name of our implementation to this file:

```
META-INF/services/com.hboeck.mp3manager.services.player.Mp3Player
    com.hboeck.mp3manager.player.Mp3PlayerImpl
```

Playback of MP3 Files

We are done with the implementation of the MP3 player's service provider and can proceed to registering an action for our MP3 file type. Using this action enables us to begin the playback via the context menu of an MP3 file in the Favorites window, or by merely double-clicking the file. Such an action can be easily created and registered using the NetBeans Action wizard (File ➤ New File ➤ Module Development ➤ Action). For the action type, select Conditionally Enabled, and select `DataObject` as the cookie class. On the next page, we can associate this action with a predefined category or create a new one. Additionally, we associate the action with a menu and enable the File Type Context Menu Item option to add the action to the context menu of an MP3 file. While creating the MP3 file type, we already used the content type `audio/mpeg`, so we do so here again. The last page of the wizard is used to specify a class name, a label, and an icon. Click Finish. Now we only need to add a few lines within the `performAction()` method, as shown in Listing 18-12.

Listing 18-12. *Context-sensitive action to play MP3 files*

```
public final class PlayAction extends CookieAction {
    protected void performAction(Node[] activatedNodes) {
        Mp3DataObject mp3 = activatedNodes[0].getLookup().lookup(Mp3DataObject.class);
        if(mp3 != null) {
            Mp3Player.getDefault().play(mp3);
        }
    }
}
```

With this action, we immediately test the MP3 player. Start the application and open the Favorites window. Add an MP3 file or a directory with MP3 files to it. Now you can start the playback by double-clicking or using the context menu.

User Interface

Having only an action is not really useful, so we will create a complete user interface for an MP3 player in this section. This user interface uses the functionality provided by the MP3 player service. Therefore, we create a new package in the `Player` module named `com.hboeck.mp3manager.player.gui`. With the help of the Window Component wizard, invoked via File ➤ New File ➤ Module Development, we create a `TopComponent` class. Specify a mode (e.g., the `explorer` mode), and use `Mp3Player` as the class name prefix. Next, build a `TopComponent` that looks like Figure 18-5 using the Matisse GUI Builder.

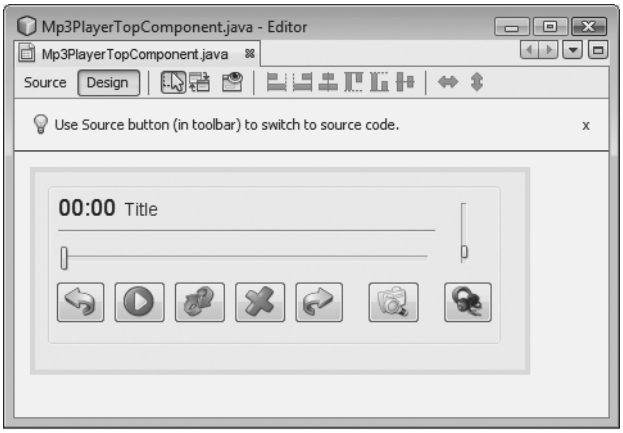


Figure 18-5. *User interface for the MP3 player*

Of course, it's not very pretty, but it's functional and offers all the relevant MP3 player functionalities. Most of the work required for designing the `TopComponent` is done by the Matisse GUI Builder; we simply have to implement the actions (see Listing 18-13). But first, we provide access to an instance of the `Mp3Player` delivered by the `getDefault()` method in the constructor. For this instance, register an `Mp3PlayerEventListener`, for which we defined an interface in the "Service Interface" section to notify starting and stopping of an MP3 file (as you may remember). Now the player events are required to update information displayed on the user interface.

Listing 18-13. *Most of the methods are really simple and only delegate the relevant values to the MP3 player.*

```
final class Mp3PlayerTopComponent extends TopComponent
        implements Mp3PlayerEventListener {
   private static final SimpleDateFormat SDF = new SimpleDateFormat("mm:ss");
   private JSlider duration;
   private JSlider volume;
   private JToggleButton mute;
   private JButton next;
   private JButton open;
   private JButton pause;
   private JButton play;
   private JButton previous;
   private JButton stop;
   private JLabel time;
   private JLabel title;
   private Timer t = null;
   private Mp3Player player = null;
   private Mp3PlayerTopComponent() {
      initComponents();
      ...
      player = Mp3Player.getDefault();
      player.addEventListener(this);
   }
   private void pauseActionPerformed(ActionEvent evt) {
      player.pause();
   }
   private void stopActionPerformed(ActionEvent evt) {
      player.stop();
   }
   private void nextActionPerformed(ActionEvent evt) {
      player.next();
   }
   private void previousActionPerformed(ActionEvent evt) {
      player.previous();
   }
   private void muteActionPerformed(ActionEvent evt) {
      player.setMute(mute.isSelected());
   }
   private void volumeStateChanged(ChangeEvent evt) {
      player.setVolume(volume.getValue());
   }
   private void durationMouseReleased(MouseEvent evt) {
      player.setMediaTime(duration.getValue());
   }
}
```

Clicking the Play button invokes the `playActionPerformed()` method, in which we can access `TopComponent.Registry` (see Listing 18-14). This will provide the currently activated nodes independently of the `TopComponent` they belong to. Resulting from this behavior, an MP3 file will be played if it is selected in any `TopComponent` (regardless of whether that's the Media Library window, the Favorites window, or somewhere else) when the Play button is clicked.

Listing 18-14. *Using TopComponent.Registry, the currently selected MP3 file can be played.*

```
private void playActionPerformed(ActionEvent evt) {
   Node n[] = getRegistry().getActivatedNodes();
   if(n != null) {
      Mp3DataObject mp3 = n[0].getLookup().lookup(Mp3DataObject.class);
      if(mp3 != null) {
         player.play(mp3);
      }
   }
}
private void openActionPerformed(ActionEvent evt) {
   JFileChooser c = new JFileChooser();
   c.setFileFilter(new FileNameExtensionFilter("MP3 Files", "mp3"));
   if(c.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
      try {
         player.play(Mp3DataObject.find(c.getSelectedFile()));
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Within the `playing()` method, called by `Mp3Player`, we can display title and time information in the user interface. Thus, we are not restricted to the file name, and we can access the ID3 tag and its stored information (see Listing 18-15). The timer is used to update playback time. In the `stopped()` method (indicating that the playback of the MP3 file was stopped), we reset all displayed information and stop the timer.

Listing 18-15. *Updating the displayed information of the current MP3 file*

```
public void playing(Mp3DataObject mp3) {
   resetInfos();
   title.setText(mp3.getName());
   duration.setMaximum(player.getDuration());
   ID3v1Tag id3v1 = mp3.getID3v1Tag();
   title.setText(id3v1.getArtist()+" - "+id3v1.getTitle());
   ActionListener updateInfo = new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
         duration.setValue(player.getMediaTime());
         time.setText(SDF.format(new Date(player.getMediaTime() * 1000)));
```

```
            }
        };
        if (t != null) {
            t.stop();
        }
        t = new Timer(1000, updateInfo);
        t.start();
    }
    public void stopped() {
        resetInfos();
        if(t != null) {
            t.stop();
        }
    }
    private void resetInfos() {
        duration.setValue(0);
        time.setText("00:00");
        title.setText("Title");
    }
}
```

Playlist

The object of this section is to create a playlist, with the additional goal of enabling the user to manage multiple playlists simultaneously. And of course we want the user to be able to add MP3 files from the media library to the playlist by merely using drag-and-drop. All this functionality is provided in a separate module. So, we create a new module via File ➤ New Project ➤ NetBeans Modules ➤ Module, name it `Playlist`, and set the code name base to `com.hboeck.mp3manager.playlist`. The `File Type` and `Services` modules are added as dependencies.

For the playlist, another `TopComponent` is used, containing a `TreeTableView` taken from the Explorer API. Using such a view eases the management of MP3 files with the help of the `Mp3DataNode` class.

Node View

Let's display our nodes. First, create a subclass of `TreeTableView` and name it `PlaylistView`. This class is used to hide the configuration and to have a handier class. The only thing we need to configure is the default action processor, because by default a double-click executes the default action of a node, which is the `PlayAction` we created in the "Playback of MP3 files" section. But this action plays only one single file, while the desired behavior of a playlist is to play the complete list automatically. Therefore, we implement the `setDefaultActionProcessor()` method (see Listing 18-16), which takes an instance of an `ActionListener`. The `actionPerformed()` method of this listener is executed (instead of the node's default action) when the node is double-clicked or the Enter key is pressed.

Listing 18-16. *This view is used to represent MP3 files in a list view.*

```
package com.hboeck.mp3manager.playlist;
import org.openide.explorer.view.TreeTableView;
public class PlaylistView extends TreeTableView {
   public PlaylistView() {
      setRootVisible(false);
   }
   public void setDefaultActionProcessor(
      final ActionListener action) {
      setDefaultActionAllowed(false);
      tree.addMouseListener(new MouseAdapter() {
         @Override
         public void mouseClicked(MouseEvent me) {
            if (me.getClickCount() == 2) {
               action.actionPerformed(null);
            }
         }
      });
      treeTable.registerKeyboardAction(action,
             KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0, false),
             JComponent.WHEN_FOCUSED);
   }
}
```

Node Container

All nodes represented in the `PlaylistView` are managed by a container. A container is based on the class `Children`, which can have several characteristics, depending on its purpose. We will use the class `Index.ArrayChildren` as superclass for our node container (see Listing 18-17). The nodes to be added to a playlist are stored in an object of type `ArrayList`, delivered by the method `initCollection()`. It will initially be empty, because the nodes are inserted via drag-and-drop from the media library. Using the `getRemaining()` method, we return a list of remaining MP3 files, which can be directly shown to the player to play back the playlist.

Listing 18-17. *Container class to manage MP3 files contained in a playlist*

```
package com.hboeck.mp3manager.playlist;
import com.hboeck.mp3manager.filetype.Mp3DataObject;
import org.openide.nodes.Index;
import org.openide.nodes.Node;
public final class NodeContainer extends Index.ArrayChildren {
   private ArrayList<Node> list = new ArrayList<Node>();
   @Override
   protected List<Node> initCollection() {
      return list;
   }
```

```
    public ListIterator<Mp3DataObject> getRemaining(Node n) {
        Vector<Mp3DataObject> v = new Vector<Mp3DataObject>();
        for (Node n : list.subList(indexOf(n), list.size())) {
            v.add(n.getLookup().lookup(Mp3DataObject.class));
        }
        return v.listIterator();
    }
    public void add(Node n) {
        add(new Node[]{n});
    }
}
```

TopComponent

Now we begin creating the playlist, again based on a `TopComponent`. This time, we will not create it using the Window Component wizard, as this wizard creates `TopComponent`s as singletons. Rather, we will use the JPanel wizard, which is accessible via File ➤ New File ➤ Swing GUI Forms ➤ JPanel Form. Name it `Playlist` and change the base class from `JPanel` to `TopComponent`, using the source view of the Matisse GUI Builder. (To have this class on hand, we specify a dependency to the Window System API.) In addition, we must override the `preferredID()` and `getPersistenceType()` methods as shown here:

```
package com.hboeck.mp3manager.playlist;
import com.hboeck.mp3manager.filetype.Mp3DataObject;
import com.hboeck.mp3manager.services.player.Mp3Player;
import org.openide.explorer.ExplorerManager;
import org.openide.explorer.ExplorerUtils;
import org.openide.nodes.AbstractNode;
public class Playlist extends TopComponent implements ExplorerManager.Provider {
    public static final String ICON_PATH =
            "com/hboeck/mp3manager/playlist/playlist.png";
    private static final String PREFERRED_ID = "Playlist";
    private static final String PREF_CURRENTDIR = "currentdir";
    private Preferences PREF = NbPreferences.forModule(Playlist.class)
    private ExplorerManager manager = new ExplorerManager();
    private NodeContainer container = new NodeContainer();
    private PlaylistView playlist = new PlaylistView();
    public Playlist() {
        initComponents();
        setName(NbBundle.getMessage(Playlist.class, "CTL_Playlist"));
        setToolTipText(NbBundle.getMessage(Playlist.class, "CTL_Playlist"));
        setIcon(ImageUtilities.loadImage(ICON_PATH, true));
        manager.setRootContext(new AbstractNode(container));
        playlist.setDefaultActionProcessor(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Mp3Player.getDefault().play(
                    container.getRemaining(manager.getSelectedNodes()[0]));
            }
        });
        associateLookup(ExplorerUtils.createLookup(manager, getActionMap()));
    }
```

```
    @Override
    protected String preferredID() {
        return PREFERRED_ID;
    }
    @Override
    public int getPersistenceType() {
        return TopComponent.PERSISTENCE_ALWAYS;
    }
    public ExplorerManager getExplorerManager() {
        return manager;
    }
    ...
}
```

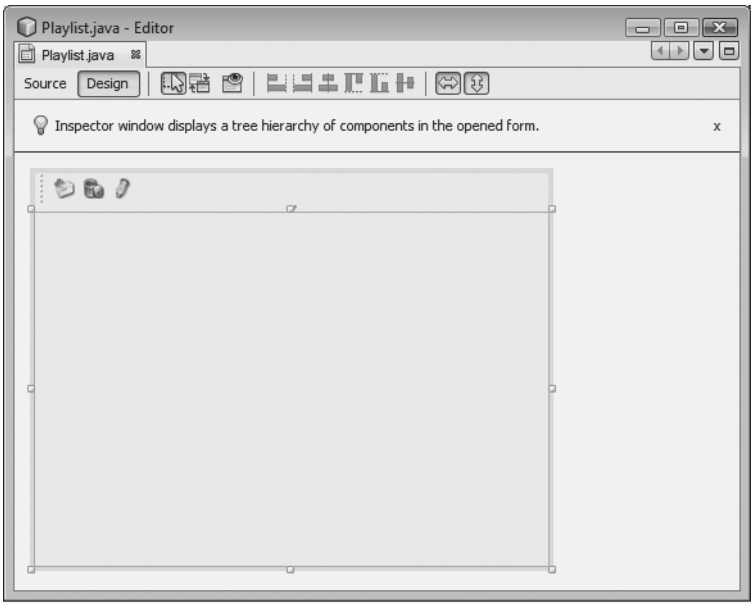


Figure 18-6. *Playlist TopComponent*

We enhance the `TopComponent` by adding a toolbar with three buttons, using the Matisse GUI Builder (see Figure 18-6). These buttons are used to add and remove files, and to name the playlist. Finally, we add a panel that uses the `BorderLayout` and occupies the whole area of the `TopComponent` (see Figure 18-6). This panel is used as the container for the node view.

Management of the nodes in our `PlaylistView` is done by an `ExplorerManager`. Therefore, we implement the interface `ExplorerManager.Provider`, create a private instance of the `ExplorerManager`, and return this manager in the `getExplorerManager()` method. Additionally, we have to create a `NodeContainer` instance. Every manager has a *root context*, which is a node that is used as root for all the other nodes. This context is set by the method `setRootContext()`. We will use an `AbstractNode` as the root context (as we do not want to display it anyway) and pass it to the container carrying the MP3 files of the playlist.

Finally, we create an instance of `PlaylistView` to which we pass the action to be performed when double-clicking an MP3 file in the playlist. To pass the action, we use the method `setDefaultActionProcessor()`. Our default behavior is to play back the complete list starting at the selected file. Therefore, the method `getRemaining()` delivers all files still remaining in the list, except those above the selected one. Now the view needs to be added to the panel we created in a previous step with the Matisse GUI Builder. To do so, select Customize Code from the context menu of the panel and insert the following lines after the layout initialization:

```
panel.add(playlist, BorderLayout.CENTER);
```

Lastly, we shouldn't forget the buttons in the toolbar, as they are used to add and remove MP3 files by invoking a file chooser dialog, as well as to rename the playlist itself. It should be possible to select multiple files and folders to add to the playlist. Therefore, we add the method `addAllFiles()` (see Listing 18-18), which recursively parses the selection and adds all files to the node container. Removing files (done by the `removeActionPerformed()` method) is much easier, as the `ExplorerManager` returns all selected entries, and the `remove()` method of the container removes an array of nodes in one step. Renaming the playlist (invoked by the method `renameActionPerformed()`) is easy as well, using the Dialogs API.

Listing 18-18. *Actions to edit the playlist*

```
public class Playlist extends TopComponent implements ExplorerManager.Provider {
   ...
   private void addActionPerformed(ActionEvent evt) {
      JFileChooser fc = new JFileChooser(PREF.get(PREF_CURRENTDIR, ""));
      fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
      fc.setFileFilter(new FileNameExtensionFilter("MP3 Files", "mp3"));
      fc.setMultiSelectionEnabled(true);
      if(fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
         addAllFiles(fc.getSelectedFiles());
         PREF.put(PREF_CURRENTDIR, fc.getCurrentDirectory().getAbsolutePath());
      }
   }
   private void addAllFiles(File[] files) {
      for(File f : files) {
         if(f.isFile()) {
            try {
               container.add(Mp3DataObject.find(f).getNodeDelegate());
            } catch(Exception e) {}
         } else if(f.isDirectory()) {
            addAllFiles(f.listFiles());
         }
      }
   }
   private void removeActionPerformed(ActionEvent evt) {
      container.remove(manager.getSelectedNodes());
   }
   private void renameActionPerformed(ActionEvent evt) {
      NotifyDescriptor.InputLine nf = new NotifyDescriptor.InputLine(
            "New Playlist Name", "Rename");
      nf.setInputText(getName());
```

```
        if(DialogDisplayer.getDefault().notify(nf) == NotifyDescriptor.OK_OPTION) {
            setName(nf.getInputText());
        }
    }
}
```

Since we didn't create the `TopComponent` for the playlist using the Window Component wizard, a menu entry allowing us to open a new playlist is missing. The simplest way to access such an action is using the Action wizard, via File ➤ New File ➤ Module Development ➤ Action. We will use the always enabled action, add it to the Window menu, and name the new class `NewPlaylist`. The wizard will then create an action class (see Listing 18-19). Next, we add the following lines to the method `actionPerformed()`.

Listing 18-19. *Action class to open a new playlist*

```
public final class NewPlaylist implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        Playlist pl = new Playlist();
        pl.open();
        pl.requestActive();
    }
}
```

We are ready to execute the application. Open one or more playlists and add MP3 files using the toolbar (see Figure 18-7).

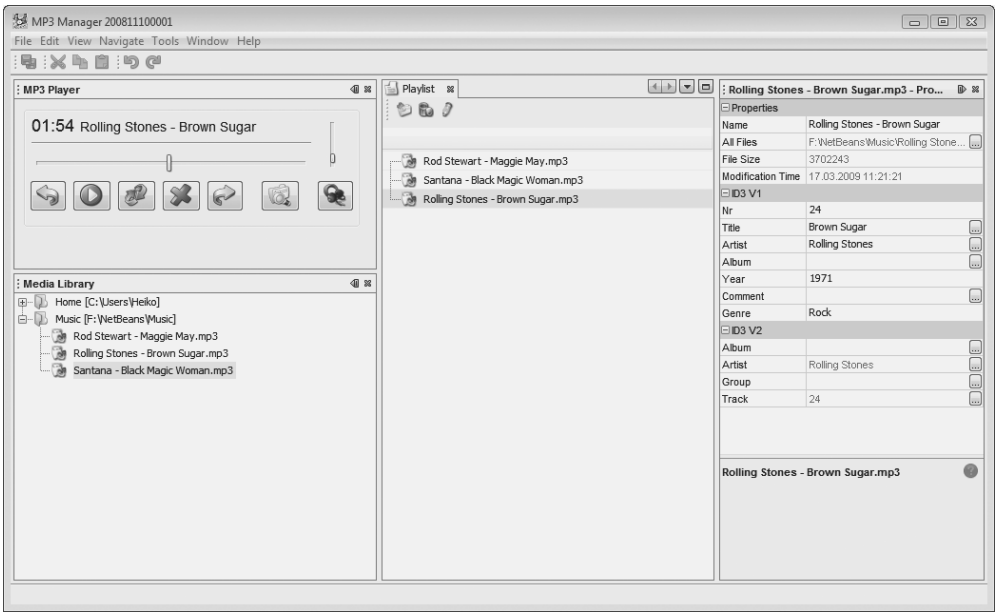


Figure 18-7. *Using the playlist toolbar, files can be added to the playlist.*

It isn't possible to drag files from the Media Library window to the playlist, because the `Mp3DataNode` class (containing the objects we want to transfer between windows) and the `PlaylistView` class are not yet prepared for dragging actions. We will cover this in the next section, as drag-and-drop is very important for easy and intuitive use.

Drag-and-Drop

First, extend the `Mp3DataNode` class. Its superclass, `AbstractNode`, already implements the `drag()` method, which is invoked if a drag event occurs. For example, a drag event will be fired when we drag files from the media library to the playlist. This method delivers an instance of type `Transferable`. So we will implement the `Transferable` interface and its methods in the class `Mp3DataNode` (see Listing 18-20). The `drag()` method is overridden and just returns a reference to itself. To access the data, and for identification purposes during a drag-and-drop operation, we have to create a `DataFlavor` object that can be accessed from outside.

Listing 18-20. *Extension to the Mp3DataNode class to enable drag-and-drop*

```
package com.hboeck.mp3manager.filetype;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
import java.awt.datatransfer.UnsupportedFlavorException;
public class Mp3DataNode extends DataNode implements Transferable {
   public static final DataFlavor DATA_FLAVOR =
      new DataFlavor(Mp3DataNode.class, "Mp3DataNode");
   ...
   @Override
   public Transferable drag() {
      return this;
   }
   public DataFlavor[] getTransferDataFlavors() {
      return new DataFlavor[]{DATA_FLAVOR};
   }
   public boolean isDataFlavorSupported(DataFlavor flavor) {
      return flavor == DATA_FLAVOR;
   }
   public Object getTransferData(DataFlavor flavor)
      throws UnsupportedFlavorException {
      if(flavor == DATA_FLAVOR) {
         return this;
      } else {
         throw new UnsupportedFlavorException(flavor);
      }
   }
}
```

An `Mp3DataNode` can now be transferred, but our playlist is still not able to accept it. We can enable it to do so by adding a `DropTarget` to our `PlaylistView`. We create an object of type `DropTarget` and pass a `DropTargetAdapter` to it (see Listing 18-21). Now we are notified regarding drag as well as drop events. The only methods we need to implement are `dragEnter()` and `drop()`. The first one is called at the moment a file is dragged to our playlist. As we only want

to allow drops of MP3 files, we check the type of data using the `DataFlavor`. In case it is not an `Mp3DataNode`, we call `rejectDrag()` to prevent a drop. The second method of implementation is invoked during the real drop event. Herein we extract the `Mp3DataNode` from the parameter and add the node to the `ExplorerManager`, or rather the container responsible for the view.

Listing 18-21. *To enable adding of MP3 files via drag-and-drop, a DropTarget is required for the PlaylistView.*

```
package com.hboeck.mp3manager.playlist;
import java.awt.dnd.DropTarget;
import java.awt.dnd.DropTargetAdapter;
import java.awt.dnd.DropTargetDragEvent;
import java.awt.dnd.DropTargetDropEvent;
public class PlaylistView extends TreeTableView {
   public PlaylistView() {
      setRootVisible(false);
      setDropTarget();
   }
   private void setDropTarget() {
      DropTarget dt = new DropTarget(this, new DropTargetAdapter() {
         @Override
         public void dragEnter(DropTargetDragEvent dtde) {
            if(!dtde.isDataFlavorSupported(
               Mp3DataNode.DATA_FLAVOR)) {
               dtde.rejectDrag();
            }
         }
         public void drop(DropTargetDropEvent dtde) {
            try {
               Mp3DataNode n = (Mp3DataNode)dtde.getTransferable().
                   getTransferData(Mp3DataNode.DATA_FLAVOR);
               ExplorerManager.find(getParent()).
               getRootContext().getChildren().add(new Node[]{n});
            } catch(Exception e) {
               e.printStackTrace();
               dtde.rejectDrop();
            }
         }
      });
      setDropTarget(dt);
   }
}
```

Thus, we are able to drag MP3 files from the media library or other sources directly into a playlist.

Saving the Playlist

You may have already noticed that the content of the playlist is lost when restarting the application. This is because the window system stores the playlist itself, but is unable to store the

contained data. In other words, we have to extend the load and store functions for our application. A good approach is to store the lists into embedded databases (e.g., Java DB). This client-side database system was already used in the "Java DB" section in Chapter 13. To use it for our playlist, we need to bind it to our application, using a module again.

The module is a library wrapper module, and is created like any other module—via File ➤ New Project ➤ NetBeans Modules ➤ Library Wrapper Module. We add to it the files `lib/derby.jar` and `lib/derbyLocale_de_DE.jar` from the Java DB distribution. Further information on how to bind and use Java DB, as well as where to obtain a distribution, can be found in the "Java DB" section of Chapter 13. Name the module `Java DB` and use `org.apache.derby` for the code name base. After creating the module, we add a module installer, which is used to initialize centralized access. Such an installer can be created via File ➤ New File ➤ Module Development ➤ Module Installer. Afterward, rename it with Refactor ➤ Rename to Database.

In the `restored()` method, called while starting the module, we set the system directory of Java DB and execute the `initTables()` method. This method will first check whether the table playlist exists, by performing a `SELECT` query (see Listing 18-22). If the table does not exist, a `SQLException` will be thrown, which we will catch in order to create the table. Using the `getConnection()` method, we obtain a connection to the database. The `close()` method allows the database system to be correctly shut down after the application is finished.

Listing 18-22. *The database class initializes the database and provides a central connection.*

```
package org.apache.derby;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import org.openide.modules.ModuleInstall;
public class Database extends ModuleInstall {
   private static Connection conn = null;
   public void restored() {
      System.setProperty("derby.system.home",
         System.getProperty("netbeans.user",
         System.getProperty("user.home")) + "/databases");
      initTables();
   }
   private void initTables() {
      try {
         Statement stmt = getConnection().createStatement();
         stmt.executeQuery("SELECT id FROM playlist");
         stmt.close();
      } catch(SQLException e) {
         try {
            Statement stmt = getConnection().createStatement();
            stmt.execute("CREATE TABLE playlist (" +
               "id VARCHAR(12)," +
               "filename VARCHAR(100))");
            stmt.close();
         } catch(SQLException ex) {
            ex.printStackTrace();
         }
```

```
      }
   }
   public static Connection getConnection() throws SQLException {
      if(conn == null || conn.isClosed()) {
         conn = DriverManager.getConnection(
            "jdbc:derby:Mp3Manager;create=true",
            "user", "password");
      }
      return conn;
   }
   public void close() {
      try {
        conn.close();
        DriverManager.getConnection("jdbc:derby:;shutdown=true");
      } catch (SQLException ex) {}
   }
}
```

Do not forget to make the `org.apache.derby` package containing the `Database` class public. To enable the `Playlist` module to access the database, we specify a dependency to the `Java DB` module. As you already know, the nodes for a view have to be provided by the class `NodeContainer`. Knowing this, it would be best to just extend this class so it reads the content of the playlist from the database for itself and can store it when the application closes. To do so, we add the methods `load()` and `update()` to the `NodeContainer` class (see Listing 18-23). The `load()` method will perform a query to read all entries for a particular playlist. When the `getNodeDelegate()` method is used, each entry will result in an `Mp3DataObject` that delivers its corresponding node.

Listing 18-23. *The load() method reads all playlist entries from the database and adds them to the container.*

```
package com.hboeck.mp3manager.playlist;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.apache.derby.Database;
public final class NodeContainer extends Index.ArrayChildren {
   ...
   public void load(String id) {
      try {
         String sql="SELECT filename FROM playlist WHERE id = ?";
         PreparedStatement stmt = Database.getConnection().prepareStatement(sql);
         stmt.setString(1, id);
         ResultSet rs = stmt.executeQuery();
         while (rs.next()) {
            try {
               add(Mp3DataObject.find(rs.getString(1)).getNodeDelegate());
            } catch(Exception e) {}
         }
         rs.close();
         stmt.close();
```

```
        } catch(SQLException e) {
            LOG.severe(e.toString());
        }
    }
```

To store the playlist, we use the `update()` method (see Listing 18-24). First, remove all entries of this specific playlist to avoid lost entries. Then use the `getNodes()` method to obtain all nodes of this container and store the path of the related MP3 file for each node.

Listing 18-24. *The update() method stores the container's entries in the database.*

```
    public void update(String id) {
        try {
            String sql = "DELETE FROM playlist WHERE id = ?";
            PreparedStatement stmt = Database.getConnection().prepareStatement(sql);
            stmt.setString(1, id);
            stmt.execute();
            stmt.close();
            sql="INSERT INTO playlist (id, filename) VALUES (?, ?)";
            stmt = Database.getConnection().prepareStatement(sql);
            for(Node n : getNodes()) {
                stmt.setString(1, id);
                stmt.setString(2, n.getLookup().lookup(Mp3DataObject.class).
                    getPrimaryFile().getPath());
                stmt.execute();
            }
            stmt.close();
        } catch(Exception e) {
            LOG.severe(e.toString());
        }
    }
}
```

You might be wondering how these methods are being called. The answer can be found if we open the `Playlist` class. Its base class, `TopComponent`, defines a method named `componentOpened()`, called while opening the window. Here, we determine the unique ID of the `TopComponent`, while in a subsequent step we call the container's `load()` method with this ID (see Listing 18-25). The `writeExternal()` method of the superclass is used to store data while the application is closing. We will override this method and invoke the `update()` method with the ID we stored as a private variable. Finally, we must invoke the `writeExternal()` method of the superclass; otherwise, the `TopComponent` will not be stored.

Listing 18-25. *The Playlist class is responsible for loading and storing the container's content.*

```
public class Playlist extends TopComponent implements ExplorerManager.Provider {
    ...
    private String id;
    ...
    @Override
```

```
    public void componentOpened() {
        id = WindowManager.getDefault().findTopComponentID(this);
        LOG.info("Load playlist with ID: " + id);
        container.load(id);
    }
    @Override
    public void writeExternal(ObjectOutput oo) throws IOException {
        LOG.info("Save playlist with ID: " + id);
        container.update(id);
        super.writeExternal(oo);
    }
}
```

Summary

In this chapter, we created a bigger example to apply a lot of the concepts you learned about in the previous chapters. Firstly, we defined a modular application structure based on NetBeans modules. We created a module, enabling our example application to play MP3 files. The module contains the JMF classes, as well as the MP3 plugin.

To handle MP3 files within the NetBeans Platform, we created an MP3 file type, as explained in Chapter 7. Next, we included an ID3 library in our NetBeans Platform application, implementing an MP3 player service module. In this module, we also implemented a small GUI for the player.

Next, we created playlist functionality. To that end, we created our own node view and node container. To implement drag-and-drop functionality from the Media Library window, we extended our `Node` class.

Finally, this chapter demonstrated how easy it is to incrementally build a modular NetBeans Platform application. The extensibility of such an application has also been highlighted.

APPENDIX

Important NetBeans Extension Points and Configuration DTDs

This Appendix outlines the most important extension points and configuration DTDs of the NetBeans Platform. The extension points are outlined first, in Table A-1. Following that, Listings A-1 through A-6 describe the DTDs for some important configuration files, including modes, toolbars, and `TopComponent` group configurations.

Table A-1. *Important NetBeans Extension Points*

Extension Point	Usage
`Actions`	Registers all actions used throughout the application. In other words, this extension point creates the central action pool, the content of which can be referenced from other classes. See Chapter 4.
`Menu`	Registers all the entries in the application menus. An application menu is built from the folders and files in this extension point. See Chapter 5.
`Navigator/Panels`	Registers all the available Navigator panels by MIME type. See Chapter 9.
`OptionsDialog`	Registers all panels that extend the Options window. Do so by registering the `OptionsCategory` implementation in the form of `.instance` entries. See Chapter 9.
`Services`	Registers service providers that are available via the standard Lookup. Alternatively, use `META-INF/services` or the filesystem. See Chapter 9.
`Services/AutoupdateType`	Registers update center configurations. See Chapter 12.
`Services/JavaHelp`	Registers JavaHelp helpsets, which are then combined with all helpsets from the other modules, resulting in a single JavaHelp system for the end user. See Chapter 9.
`Services/MIMEResolver`	Registers specific data types to a `DataLoader`, which in turn is a factory for its `DataObject`. See Chapter 7.
`Shortcuts`	Registers keyboard shortcuts for an action. Provides a central overview of all existing shortcuts. See Chapter 3.

Table A-1. *Important NetBeans Extension Points (Continued)*

Extension Point	Usage
TaskList/Groups	Registers task groups that are shown in the NetBeans Platform's task list. See Chapter 15.
TaskList/Scanners	Registers custom scanner implementations that provide tasks for the NetBeans Platform's task list. See Chapter 15.
TaskList/Scopes	Registers custom scopes for task list searches.
Toolbars	Registers new toolbars and their actions. You can also add actions to preexisting toolbars via this extension point. See Chapter 5.
WarmUp	Registers instances of the Runnable class, which are executed automatically and asynchronously as applications start. See Chapter 17.
Windows2/Components	Registers module TopComponents. See Chapter 9.
Windows2/Groups	Registers groups of related TopComponents that should behave in concert with each other. See Chapter 9.
Windows2/Modes	Registers custom modes—i.e., areas within the application where TopComponents can be displayed. See Chapter 9.

Listing A-1. *Mode definitions*

```
<!-- //NetBeans//DTD Mode Properties 2.2//EN -->
<!ELEMENT mode (
   module?,
   name,
   kind,
   state,
   constraints?,
   (bounds | relative-bounds)?,
   frame?,
   active-tc?,
   empty-behavior?,
   slidingSide?,
   slideInSize*) >
<!ATTLIST mode
   version CDATA #REQUIRED >

<!ELEMENT module EMPTY >
<!ATTLIST module
   name CDATA #REQUIRED
   spec CDATA #IMPLIED >

<!ELEMENT name EMPTY >
<!ATTLIST name
   unique CDATA #REQUIRED >

<!ELEMENT kind EMPTY >
<!ATTLIST kind
```

```
   type (editor | view | sliding) #REQUIRED >

<!ELEMENT slidingSide EMPTY >
<!ATTLIST slidingSide
   side (left | right | bottom) #REQUIRED >

<!ELEMENT slideInSize EMPTY >
<!ATTLIST slideInSize
   tc-id CDATA #REQUIRED
   size  CDATA #REQUIRED >

<!ELEMENT state EMPTY >
<!ATTLIST state
   type (joined | separated) #REQUIRED >

<!-- This entry is used when a window is moved out of the application
        via the „Undock" function. -->
<!ELEMENT bounds EMPTY >
<!ATTLIST bounds
   x      CDATA #REQUIRED
   y      CDATA #REQUIRED
   width  CDATA #REQUIRED
   height CDATA #REQUIRED >

<!ELEMENT relative-bounds EMPTY >
<!ATTLIST relative-bounds
   x      CDATA #REQUIRED
   y      CDATA #REQUIRED
   width  CDATA #REQUIRED
   height CDATA #REQUIRED >

<!-- The current status of the windows. The value is represented by an integer.
        Settable values can be found in the java.awt.Frame class.
        Default: Frame.NORMAL (0) -->
<!ELEMENT frame EMPTY >
<!ATTLIST frame
   state CDATA #IMPLIED >

<!ELEMENT constraints (path*) >
<!ATTLIST constraints >

<!ELEMENT path EMPTY >
<!ATTLIST path
   orientation (horizontal | vertical) #REQUIRED
   number      CDATA                   #REQUIRED
   weight      CDATA                   #IMPLIED >

<!ELEMENT active-tc EMPTY >
<!ATTLIST active-tc
   id CDATA #IMPLIED > // ID of the active TopComponent
```

```
<!-- If set to true permanently, the mode will continue to exist even when no
        TopComponent is docked within it -->
<!ELEMENT empty-behavior EMPTY >
<!ATTLIST empty-behavior
   permanent (true | false) #IMPLIED >
```

Listing A-2. *Ordering of TopComponents in modes*

```
<!-- //NetBeans//DTD Top Component in Mode Properties 2.2//EN -->
<!ELEMENT tc-ref (
   module?,
   tc-id,
   state,
   previousMode,
   docking-status?,
   slide-in-status?) >
<!ATTLIST tc-ref
   version CDATA #REQUIRED>

<!-- This optional element is used to kill the TopComponent
        when the specified module is deactivated -->
<!ELEMENT module EMPTY >
<!ATTLIST module
   name CDATA #REQUIRED  // Code name base of the module
   spec CDATA #IMPLIED > // Specification version of the module

<!ELEMENT tc-id EMPTY >
<!ATTLIST tc-id
   id CDATA #REQUIRED >  // Unique ID of the TopComponent

<!ELEMENT state EMPTY >
<!ATTLIST state
   opened (true | false) #REQUIRED >

<!-- This attribute is used by the sliding views to put the TopComponent back
        into its original state -->
<!ELEMENT previousMode EMPTY >
<!ATTLIST previousMode
   name  CDATA
   index CDATA #IMPLIED>

<!ELEMENT docking-status EMPTY >
<!ATTLIST docking-status
   maximized-mode (docked | slided) #IMPLIED
   default-mode   (docked | slided) #IMPLIED >

<!ELEMENT slide-in-status EMPTY >
<!ATTLIST slide-in-status
   maximized (true | false) #IMPLIED >
```

Listing A-3. *TopComponent group definitions*

```
<!-- //NetBeans//DTD Group Properties 2.0//EN -->
<!ELEMENT group (
   module?,
   name,
   state) >
<!ATTLIST group
   version CDATA #REQUIRED >

<!ELEMENT module EMPTY >
<!ATTLIST module
   name CDATA #REQUIRED
   spec CDATA #IMPLIED >

<!ELEMENT name EMPTY >
<!ATTLIST name
   unique CDATA #REQUIRED >

<!ELEMENT state EMPTY >
<!ATTLIST state
   opened (true | false) #REQUIRED >
```

Listing A-4. *Ordering of TopComponents in groups*

```
<!--//NetBeans//DTD Top Component in Group Properties 2.0//EN -->
<!ELEMENT tc-group (
   module?,
   tc-id,
   open-close-behavior) >
<!ATTLIST tc-group
   version CDATA #REQUIRED >

<!ELEMENT module EMPTY >
<!ATTLIST module
   name CDATA #REQUIRED
   spec CDATA #IMPLIED >

<!ELEMENT tc-id EMPTY >
<!ATTLIST tc-id
   id CDATA #REQUIRED > // unique ID of the TopComponent

<!ELEMENT open-close-behavior EMPTY >
<!ATTLIST open-close-behavior
   open       (true | false)   #REQUIRED
   close      (true | false)   #REQUIRED
   was-opened (true | false)   #IMPLIED >
```

Listing A-5. *Toolbar definition and configuration*

```
<!-- //NetBeans//DTD Toolbar Configuration //EN -->
<!ELEMENT Configuration (Row+) >
<!ELEMENT Row (Toolbar*) >
<!ELEMENT Toolbar EMPTY >
<!ATTLIST Toolbar
   name     CDATA          #REQUIRED
   position CDATA          #IMPLIED
   visible  (true | false) #IMPLIED >
```

Listing A-6. *Palette item definition*

```
<!-- //NetBeans//DTD Editor Palette Item 1.1//EN -->
<!ELEMENT editor_palette_item (
   (class|body),
   icon16,
   icon32,
   (description|inline-description)) >
<!ATTLIST editor_palette_item
   version CDATA #REQUIRED >

<!-- Name of the class that implements the
        org.openide.text.ActiveEditorDrop interface -->
<!ELEMENT class EMPTY>
<!ATTLIST class
   name CDATA #REQUIRED >

<!-- Textual description, which can also contain HTML tags -->
<!ELEMENT body (#PCDATA)>

<!ELEMENT icon16 EMPTY>
<!ATTLIST icon16
   urlvalue CDATA #REQUIRED >
<!ELEMENT icon32 EMPTY>
<!ATTLIST icon32
   urlvalue CDATA #REQUIRED >

<!ELEMENT description EMPTY>
<!ATTLIST description
   localizing-bundle CDATA #REQUIRED
   display-name-key  CDATA #REQUIRED
   tooltip-key       CDATA #REQUIRED >

<!ELEMENT inline-description (display-name, tooltip)>
<!ELEMENT display-name (#PCDATA)>
<!ELEMENT tooltip (#PCDATA)>
```

Index

O

P

Q

R

S

T

U

V

W

X

Z