CHAPTER 11

Refactoring

Refactoring capabilities are very important when working in the software industry. Anyone who has ever had to overhaul an existing code base has run into issues with changing code. One of the most common examples is moving classes between packages and having to manually edit the package statements at the top of each file. Another example is wanting to delete an element in code (such as a class member, an old utility method, and so on) and not knowing if code in your application still uses that element.

Manually performing these types of operations can be time-consuming and error-prone. In the days before advanced development environments, programmers used simpler tools like basic text editors, vi, or Emacs. While some of these tools allow you to search, match, and replace text, they are not Java-aware and thus produce incorrect results.

With the advanced capabilities available in IDE tools like NetBeans, developers have tool sets for refactoring code. With access to parsed source files and near real-time syntax validation, NetBeans can intelligently allow a developer to alter source code.

In this chapter, we'll review the NetBeans refactoring options.

Using NetBeans Refactoring Options

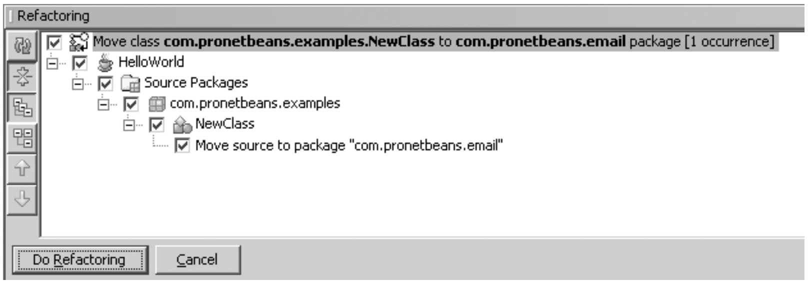
NetBeans provides many refactoring options on its Refactor menu:

- Move Class
- Rename
- Safely Delete
- Use Supertype Where Possible
- Move Inner to Outer Level
- Encapsulate Fields
- Pull Up
- Push Down
- Convert Anonymous to Inner
- Extract Method

- Extract Interface
- Extract Superclass
- Change Method Parameters
- Query and Refactor

When you execute a refactoring operation, a dialog box appears with options for the corresponding refactoring. All of them include a Preview All Changes check box, which is selected by default.

When it comes to refactoring, no tool is perfect, so I recommend always previewing changes before applying them. As shown in Figure 11-1, the preview window allows you to review each and every change that will be made to your code before it is applied.



Move class **com.pronetbeans.examples.NewClass** to **com.pronetbeans.email** package [1 occurrence]
HelloWorld
Source Packages
com.pronetbeans.examples
NewClass
Move source to package "com.pronetbeans.email"

Figure 11-1. *Previewing changes for the Move Class refactoring*

The icons along the left side of the preview window let you work with the preview as follows:

- The top icon refreshes the refactoring changes listed in the window.
- The second icon collapses or expands the tree hierarchy of the changes. This can be very useful when the list of changes is long.
- The third icon displays the logical view of the refactoring actions that will be performed.
- The fourth icon displays the physical view of the refactoring actions that will be performed.
- The last two icons let you navigate up and down to each change.

As you navigate up and down the changes in the preview window, the source file opens in the Source Editor and highlights the line to be changed. This lets you examine each change if you are concerned about the validity of the refactoring. You can click the Do Refactoring button to apply the changes, or click Cancel if you don't want the changes to be made.

Now let's look at how each of the refactoring options works.

Move Class Refactoring

Moving a Java class from one package to another seems like a simple task at first glance. A developer can manually copy and paste a source file into the new directory, and then edit the package statement at the top of the file. However, if other classes import or reference that class, then the developer must also search through and modify those files as well.

In NetBeans, the Move Class refactoring does exactly as the name implies. It allows you to move a Java class to a different project, different package hierarchy, or between source and test packages. It also corrects the references to the moved class that exist in other classes.

To use the Move Class option, select a class, and then choose Refactor ➤ Move Class or use the keyboard shortcut Alt+Shift+V. You will see the Move Class dialog box, as shown in Figure 11-2. In the Move Class dialog box, you can choose to move the class to a different project, location, or package.

Tip If you move one or more classes to the wrong package and apply the changes, don't panic. Most refactorings can be undone in NetBeans. From the main Refactoring menu just select the Undo option.

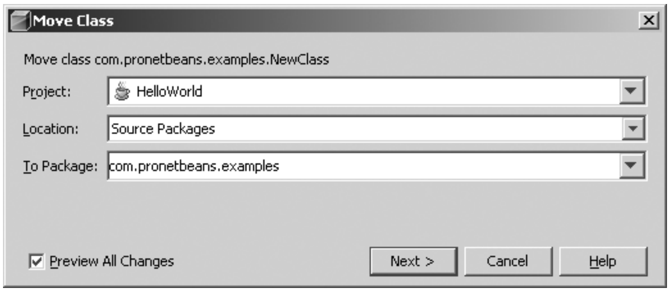

Move class com.pronetbeans.examples.NewClass
Project: HelloWorld
Location: Source Packages
To Package: com.pronetbeans.examples
☑ Preview All Changes

Figure 11-2. *The Move Class dialog box*

You can also activate the Move Class refactoring by dragging-and-dropping a class in the Projects window into a different location. The only difference in using the refactoring in this manner is that an additional option appears in the Move Class dialog box: Move Without Refactoring. If this option is checked, NetBeans moves the class without scanning additional classes to correct references to the moved class. You might want to use this option if you need to move a class out of a package temporarily, and move it back later. For example, while testing a package or running some analysis tool against a package, you may want to quickly exclude a class under development.

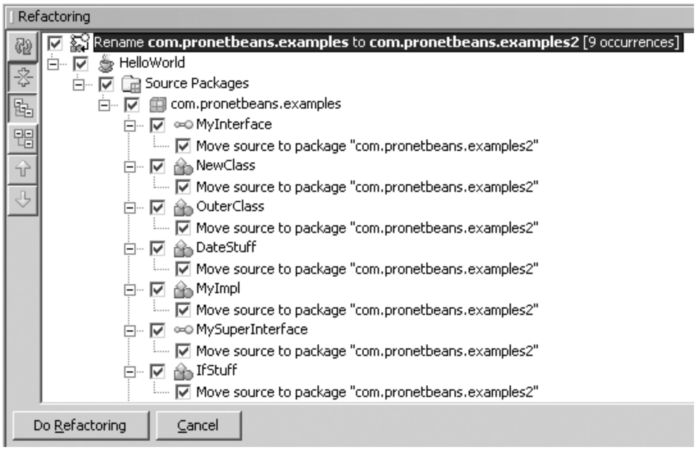
Rename Refactoring

The Rename refactoring can be used for two main purposes:

Renaming Java classes: Using the Rename refactoring will not only change the name of the class, but also any constructors, internal usages, and references to the renamed class by other classes. If you need to rename a Java class, this is definitely the way to do it.

Renaming entire package structures: This can be useful if a programmer named a package incorrectly or misspelled a word that appears in the package structure. Rather than having to manually make the corrections, the Rename option can correct the errors all at once across the entire project.

To rename a class or package, select it and choose Refactor ➤ Rename. Enter the new name in the Rename Refactoring dialog box. Figure 11-3 shows an example of the pending changes in the preview window for a Rename operation. In the example, the `com.pronetbeans.examples` package is being renamed to `com.pronetbeans.examples2`.


Refactoring

```
Rename com.pronetbeans.examples to com.pronetbeans.examples2 [9 occurrences]
  HelloWorld
    Source Packages
      com.pronetbeans.examples
        MyInterface
          Move source to package "com.pronetbeans.examples2"
        NewClass
          Move source to package "com.pronetbeans.examples2"
        OuterClass
          Move source to package "com.pronetbeans.examples2"
        DateStuff
          Move source to package "com.pronetbeans.examples2"
        MyImpl
          Move source to package "com.pronetbeans.examples2"
        MySuperInterface
          Move source to package "com.pronetbeans.examples2"
        IfStuff
          Move source to package "com.pronetbeans.examples2"
```

Figure 11-3. *The preview window for the Rename refactoring*

Safely Delete Refactoring

During the software development process, programmers frequently revisit previously written code. During that time, they review what was written and decide what can and cannot be cleaned up and removed. One common mistake is removing a class member variable that you think is not used, only to discover that it does indeed appear in your code, and now your class does not compile.

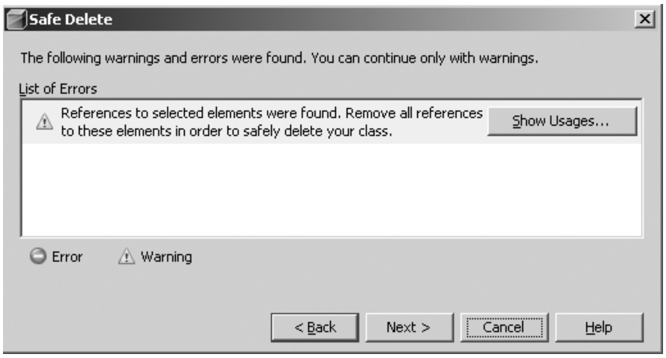
Using the Safely Delete refactoring, you can identify each usage of a class, method, or field in code before deleting it. This functionality can be invaluable, especially if you are removing a class method that may be used by multiple classes. For example, consider the following code fragment, which is a sample method that declares several method local variables and performs some nonsense operations.

```
public void calc() {

    int y = 2;
    int x = 0;
    int z = 0;

    z = x + y;

    if(z>3) {
        System.out.println("Z was greater than 3");
    }
    else if(y==2){
        System.out.println("x = "  + x);
    }
}
```

During a review of this class, you decide to delete the variable x. You could visually scan the class to see if the x variable is being used anywhere. In this example, it is pretty easy to find x being output in the System.out.println statement. However, if this method were 100 lines long and contained multiple nested statements, spotting x would be much more difficult.

To execute the Safely Delete refactoring, highlight the variable you want to delete (x in the example) and select Safely Delete. In the Safely Delete dialog box, the Search in Comments check box makes sure that the element is also deleted in any Javadoc comments in which it may appear. The only other option is the standard Preview All Changes check box, allowing you to review each change before it is made.

If an element is not used anywhere in your code, it is safe to delete. However, if the element you are attempting to delete is used somewhere in your code, some additional steps may be necessary. After clicking the Next button in the initial Safely Delete dialog box, a list of errors and warnings will appear, as shown in Figure 11-4. As long as there are only warnings displayed, you can proceed with the refactoring.


Safe Delete

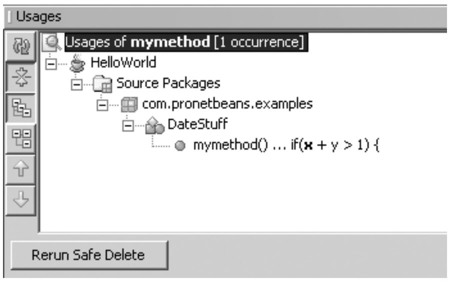
The following warnings and errors were found. You can continue only with warnings.

List of Errors

⚠ References to selected elements were found. Remove all references to these elements in order to safely delete your class.

Figure 11-4. *List of errors and warnings for the Safely Delete refactoring*

If you see errors in the list, you'll need to do a bit of work. The Show Usages button is key to resolving any sections in your code that reference the variable being deleted. Click the Show Usages button to open the Usages window, as shown in Figure 11-5.


Usages

```
Usages of mymethod [1 occurrence]
HelloWorld
  Source Packages
    com.pronetbeans.examples
      DateStuff
        mymethod() ... if(x + y > 1) {
```

Figure 11-5. *Viewing usages of the element to delete*

The Usages window displays each usage of the element you are trying to delete. Click a usage in the window, and the exact line in the source code will open in the Source Editor window. After navigating to each usage and manually correcting the code to not use the variable being deleted, you can click the Rerun Safe Delete button.

The Safely Delete refactoring may seem like a waste of time in certain circumstances. For instance, you may not need to use it if you are deleting a local variable in a method that is five or ten lines long. It is most useful if you have a class member variable or method that is used across numerous classes. The Safely Delete option allows you to review each usage and make sure you do not delete the element until there are no more references to it.

Use Supertype Where Possible Refactoring

The Use Supertype Where Possible refactoring converts usage of a subclass to a superclass. Suppose you have the following code in a source file:

```
ArrayList myarray = new ArrayList();
```

If you want to convert it to use a specific superclass, double-click or highlight the object type `ArrayList` and select Refactor ➤ Use Supertype Where Possible. You'll see the Use Supertype dialog box, which allows you to select a superclass or interface, as shown in Figure 11-6.

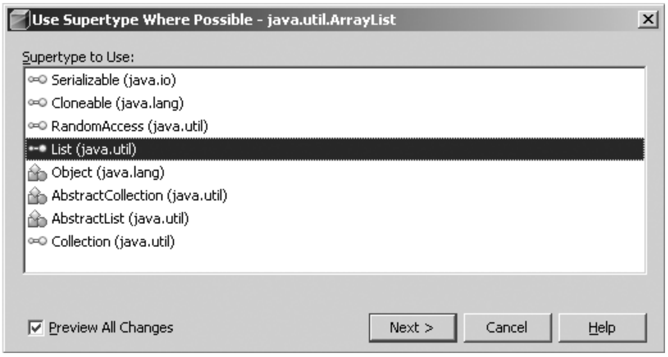


Figure 11-6. *The Use Supertype Where Possible dialog box for java.util.ArrayList*

Obviously, this is a ridiculously simple example, but it demonstrates the core functionality. This method can also be used in conjunction with the Extract Superclass refactoring, described later in this chapter.

Move Inner to Outer Level Refactoring

The Move Inner to Outer Level refactoring converts an inner class to a separate external class declared in its own file. Suppose you have the following code, in which the `InnerClass` class is declared inside the `OuterClass` class.

```
public class OuterClass {
    public class InnerClass {
        public void execute() {
            System.out.println("execute...");
        }
    }
}
```

To move the InnerClass class to its own source file, highlight the class name and select Refactor ➤ Move Inner to Outer Level. In the Move Inner to Outer Level dialog box, you can specify a new name for the class that is being moved, as shown in Figure 11-7. This can be convenient, especially since inner classes are often named to make sense within the context of the containing outer class. Optionally, you can select to declare a field for the current outer class and enter a name for that field.

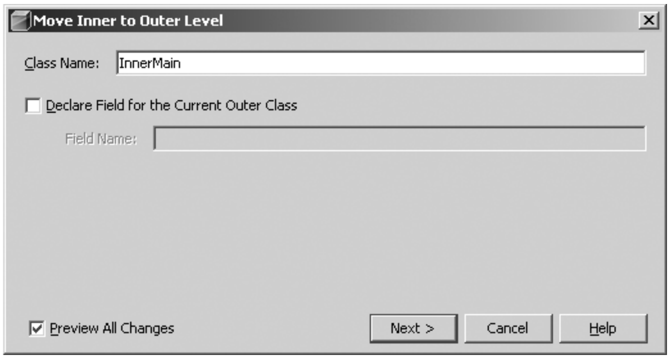


Figure 11-7. *The Move Inner to Outer Level dialog box*

If you apply the refactoring using the default settings, when you click the Next button, the code that results is as follows:

```
public class InnerClass {
    public void execute() {
        System.out.println("execute…");
    }
}
```

The InnerClass code is moved to its own individual source file of the same name in the same package as OuterClass.

If you select the Declare Field for the Current Outer Class option and name a variable, the refactored code looks like this:

```
public class InnerClass {

    private final OutClass myvar;

    public void execute(OuterClass newvar) {
        this.newvar = newvar;

        System.out.println("execute...");
    }
}
```

This option can be useful when separating the classes, especially if the InnerClass class used the members or methods of the OuterClass class.

Encapsulate Fields Refactoring

When writing applications, it is often useful to represent objects in the real world as classes with attributes. For example, you may choose to represent the fields for an employee as an Employee class with first name and last name public members:

```
public class Employee {
    public String FirstName;
    public String LastName;
}
```

Of course, you might also include address, phone number, organizational, and personal fields in the class.

Such an Employee class is quick and easy to work with, such as in the following code:

```
public class NewHire {
    public static void main(String[] args) {
        Employee newemp = new Employee();
        newemp.FirstName = args[0];
        newemp.LastName = args[1];
        saveEmployee(newemp);
    }
}
```

In the NewHire class, an instance of Employee is instantiated and the FirstName and LastName fields are set from the arguments passed on the command line. (Obviously, there are a lot of problems with the code in the NewHire class, such as no parameter or error checking, but here we are just focusing on the topic of encapsulation.)

As a programmer, you should be starting to realize this approach has some negative design features. For example, suppose your client has requested that the employee name be stored in the database with initial capital letters, such as John Smith. However, in the application, the values need to be processed in uppercase. You could rewrite the entire application to add the usage of String.toUpperCase() anywhere the Employee.FirstName and Employee.LastName fields are output or processed throughout the entire code base. You could also encapsulate the fields.

Encapsulation involves controlling access to a class member variable using getter and setter methods. The class member variable is set to private, so that no code outside the class can interact with it. The getter and setter methods are usually given a public accessor, so that any code can retrieve or set the value of the member variable.

In the following code, the Employee class has been modified to use getters and setters for the FirstName and LastName member variables.

```
 public class Employee {
    private String FirstName;
    private String LastName;

    public void setFirstName(String FirstName) {
        this.FirstName = FirstName;
    }
    public String getFirstName() {
        return this.FirstName;
    }

    public void setLastName(String LastName) {
        this.LastName = LastName;
    }

    public String getLastName() {
        return this.LastName;
    }
}
```

You can also modify the code in the `NewHire` class to interact with the updated `Employee` class. The `NewHire` class must now use the getter and setter methods.

```
public class NewHire {
    public static void main(String[] args) {

        Employee newemp = new Employee();
        newemp.setFirstName(args[0]);
        newemp.setLastName(args[1]);

        saveEmployee(newemp);
    }
}
```

Using this type of design, you are in a better position to modify the code to handle special conditions. In the example, the code in the `Employee` class can be modified to convert the member variables to uppercase when they are set using `Employee.setFirstName` and `Employee.setLastName`.

```
public class Employee {
    private String FirstName;
    private String LastName;

    public void setFirstName(String FirstName) {
        if(FirstName!=null) {
            this.FirstName = FirstName.toUpperCase();
        } else {
            this.FirstName = null;
        }
    }

    public String getFirstName() {
        return this.FirstName;
    }
    public void setLastName(String LastName) {
        if(LastName!=null) {
            this.LastName = LastName.toUpperCase();
        } else {
            this.LastName = null;
        }
    }

    public String getLastName() {
        return this.LastName;
    }
}
```

■Note It is usually preferable to perform any data conversion, checking, or modification in the setter method for a member variable, rather than in the getter method. If the data conversion is implemented in the getter, each time the data is retrieved, the data conversion will take place, thus slightly reducing performance.

Generally, it is a common best practice to never have a public member of a class for which you write other code to set or get the value. Arguably, the only exception to the rule is using static constants.

Now that you have read a quick review of a key object-oriented concept, we can discuss how NetBeans can assist in encapsulation. (I apologize to those of you groaning about now, but this is one of the most frequent mistakes I see programmers make, so it deserves some review.)

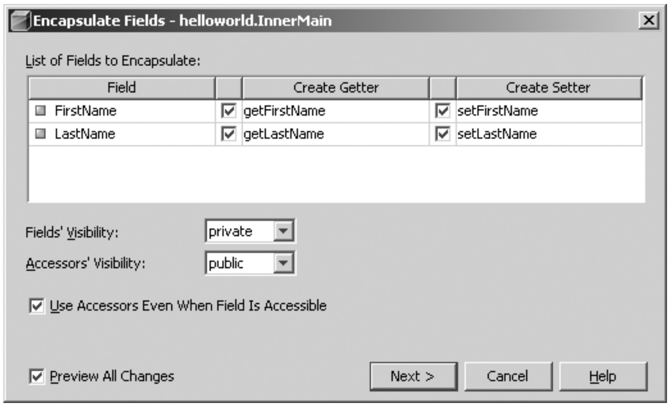
The Encapsulate Fields refactoring in NetBeans allows you to easily implement the design paradigm of encapsulation. It helps you to generate getter and setter methods for the members of a class to enforce good design.

Suppose you have the simple `Employee` class shown at the beginning of this section:

```
public class Employee {
    public String FirstName;
    public String LastName;
}
```

If you highlight the name of the class and select Refactor ➤ Encapsulate Fields, the Encapsulate Fields dialog box will list all the class fields, all selected by default. If you highlight specific class fields and select the Encapsulate Fields option, the dialog box will still display the entire list of fields in the class, but only the field or fields you highlighted will be selected by default. For example, if you highlighted the `FirstName` and `LastName` fields (the entire line for each field), the dialog box will list both the fields, as shown in Figure 11-8.

You can disable or enable creation of the getter and setter methods using the check boxes next to each one. In this dialog box, you can also manually alter the names of the getter and setter methods. The Fields' Visibility and Accessors' Visibility drop-down lists allow you to set the access level to the original fields (should be private) and to the getters and setters (should be public), respectively.


Encapsulate Fields - helloworld.InnerMain

List of Fields to Encapsulate:

Field		Create Getter		Create Setter
FirstName	✓	getFirstName	✓	setFirstName
LastName	✓	getLastName	✓	setLastName

Fields' Visibility: private

Accessors' Visibility: public

✓ Use Accessors Even When Field Is Accessible

✓ Preview All Changes

Figure 11-8. *The Encapsulate Fields dialog box*

In my opinion, the Use Accessors Even When Field is Accessible option should always remain checked. Then the refactoring procedure attempts to correct code in other classes that use the class member variables and convert it to use the accessors (getters and setters). The only time you might want to disable this option is when you set the Fields' Visibility option to anything other than private. The refactoring will then perform the Encapsulate Fields operation, but will not convert code to use the accessors.

Once the overall refactoring is complete, the `Employee` class should look like this:

```
public class Employee {
    private String FirstName;
    private String LastName;

    public String getFirstName()
    {
        return FirstName;
    }

    public void setFirstName(String FirstName)
    {
        this.FirstName = FirstName;
    }

    public String getLastName()
    {
        return LastName;
    }

    public void setLastName(String LastName)
    {
        this.LastName = LastName;
    }
}
```

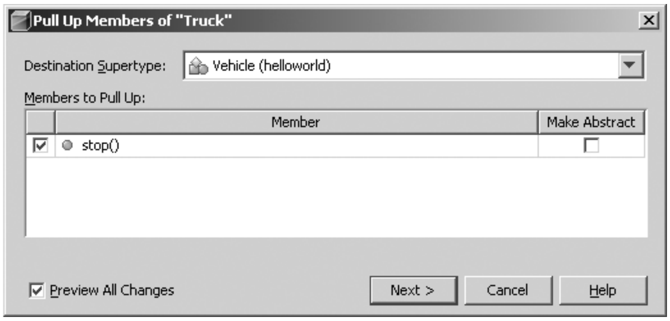
Pull Up Refactoring

The Pull Up refactoring is useful when dealing with classes and superclasses. It allows you to move class members and methods from a subclass up into the superclass.

For example, suppose you have a `Vehicle` class and a `Truck` class that extends `Vehicle`:

```
public class Vehicle
{
    public void start()
    {
        // start the vehicle
    }
}

public class Truck extends Vehicle
{
    public void stop()
    {
        // stop the vehicle
    }
}
```

If you want to move the `stop()` method from the `Truck` subclass to the `Vehicle` superclass, select the `stop()` method and select Refactor ➤ Pull Up. In the Pull Up dialog box, select the destination supertype, the exact list of members to pull up, and whether or not to make them abstract, as shown in Figure 11-9.


Pull Up Members of "Truck"

Destination Supertype: Vehicle (helloworld)

Members to Pull Up:

	Member	Make Abstract
☑	● stop()	☐

☑ Preview All Changes

Figure 11-9. *The Pull Up dialog box*

Once the refactoring changes have been applied, the `Truck` and `Vehicle` classes look like this:

```
public class Vehicle
{
    public void start()
    {
        // start the vehicle
    }
    public void stop()
    {
        // stop the vehicle
    }
}

public class Truck extends Vehicle
{
}
```

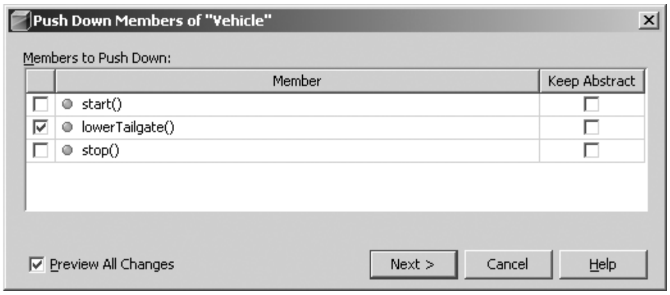
Push Down Refactoring

The Push Down refactoring is exactly the opposite of the Pull Up refactoring. It pushes an inner class, field, or method in a superclass down into a subclass. For example, suppose that you added a `lowerTailgate()` method to the `Vehicle` class shown in the previous example:

```
public class Vehicle
{
    public void start()
    {
        // start the vehicle
    }

    public void stop()
    {
        // stop the vehicle
    }

    public void lowerTailgate()
    {
        // lower tailgate of vehicle
    }
}

public class Truck extends Vehicle
{
}
```

However, since many vehicles (such as cars, planes, and boats) do not have tailgates, you want to push the `lowerTailgate()` method down to the `Truck` subclass.

Select the `lowerTailgate()` method and choose Refactor ➤ Push Down. In the Push Down dialog box, select which class members you want to push down into the subclass, as shown in Figure 11-10. You can also choose whether you would like to keep them abstract if they already are abstract.



	Member	Keep Abstract
☐	start()	☐
☑	lowerTailgate()	☐
☐	stop()	☐

Figure 11-10. *The Push Down dialog box*

After you have applied the code changes, you can view the result. As expected, the `lowerTailgate()` method will now be in the `Truck` subclass.

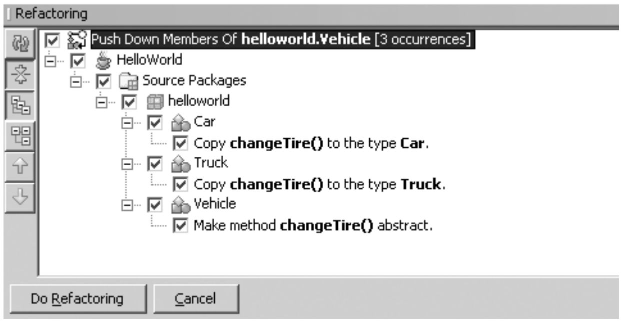
```
public class Truck extends Vehicle
{
    public void lowerTailgate()
    {
        // do something
    }
}
```

If the superclass has multiple subclasses (which is usually the case), you could still perform a Push Down refactoring of a method from a particular class. For example, if you had a `Car` subclass that extended `Vehicle`, you could still push down a method from the `Vehicle` class. Suppose the `Truck`, `Car`, and `Vehicle` classes were defined as follows:

```
public class Vehicle
{
    public void changeTire()
    {
        // general method for changing tire
    }
}

public class Car extends Vehicle
{
   // car class
}

public class Truck extends Vehicle
{
  // truck class
}
```

The `Truck` class represents a large tractor-trailer. Changing a tire for this type of vehicle will most likely involve a different procedure than for a car. Thus, you might want to have the `changeTire()` method in the `Car` and `Truck` classes override the one in the `Vehicle` superclass. The `changeTire()` method in the `Vehicle` class should also be left as abstract (even though some vehicles, like boats, do not have tires that need changing).

In the Push Down dialog box, you need to select the check box to keep the `changeTire()` method abstract in the `Vehicle` class. Preview the changes to make sure the code is modified as you expect. In Figure 11-11, notice the third suggested operation is altering `Vehicle.changeTire()` to make it abstract. If the Keep Abstract option is not selected during the refactoring operation, then the line in the preview window would say "Remove changeTire() element." You could prevent it from being removed from the `Vehicle` class by unselecting the check box next to this option.



- [✓] Push Down Members Of **helloworld.Vehicle** [3 occurrences]
  - [✓] HelloWorld
    - [✓] Source Packages
      - [✓] helloworld
        - [✓] Car
          - [✓] Copy **changeTire()** to the type **Car**.
        - [✓] Truck
          - [✓] Copy **changeTire()** to the type **Truck**.
        - [✓] Vehicle
          - [✓] Make method **changeTire()** abstract.

Figure 11-11. *Push Down refactoring with one superclass and two subclasses*

Convert Anonymous to Inner Refactoring

The Convert Anonymous to Inner refactoring is used to separate an anonymous inner class into an actual inner class. There are several varieties of anonymous inner classes:

- Inner class for defining and instantiating an instance of an unnamed subclass
- Inner class for defining and instantiating an anonymous implementation of an interface
- Anonymous inner class defined as an argument to a method

For this section, we will focus on the first type: unnamed subclasses. Suppose you have the following code:

```
public class Item {
    public void assemble() {
        System.out.println("Item.assemble");
    }
}
```

```
public class Factory {
    public void makeStandardItem(int type) {
        if(type==0) {
            // make extremely unusual item .01% of the time
            Item myitem = new Item() {
                public void assemble() {
                    System.out.println("anonymous Item.assemble");
                }
            };
            myitem.assemble();
        } else {
            // make standard item 99.9% of the time
            Item myitem = new Item();
            myitem.assemble();
        }
    }
}
```

The code declares a class `Item` with a method named `assemble()`. The `Factory` class defines a variable `myitem` of type `Item` and instantiates an anonymous subclass of `Item` that overrides the `assemble()` method.

Why would you bother using an anonymous inner class instead of a normal inner or outer class? In this example, if the one-off case where the anonymous inner class is used were the only area it is needed, you might not want to create a separate class. However, if you find that you need the code in the anonymous subclass in multiple areas, you might want to convert it to an inner class.

To convert the code to an inner class, click anywhere inside the anonymous class or highlight the name of the `Item` class constructor in the following section of the code:

```
Item myitem = new Item() {
    public void assemble() {
        System.out.println("anonymous Item.assemble");
    }
};
```

Then select Refactor ➤ Convert Anonymous Class to Inner. In the Convert Anonymous Class to Inner dialog box, you'll see the default class name of `NewClass`, as shown in Figure 11-12. You can set the name of the new inner class that will be created, the access level, and whether it should be declared static. If the constructor for the anonymous class has any parameters, the dialog box will also list them.

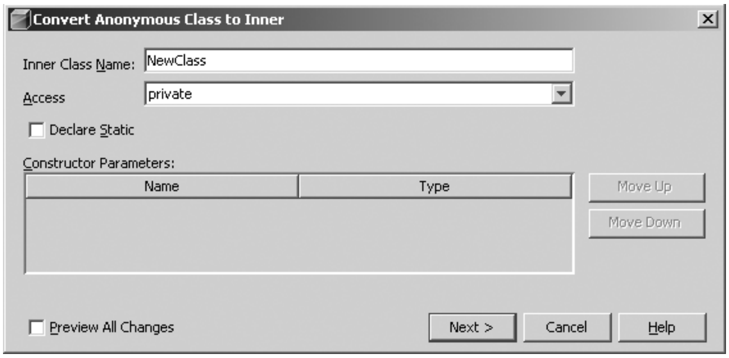


Figure 11-12. *The Convert Anonymous Class to Inner dialog box*

Suppose you named the new inner class StrangeItem. The refactored code would look like this:

```
private class StrangeItem extends Item {

    public void assemble() {
        System.out.println("anonymous Item.assemble");
    }
}
```

This class would be declared inside the Factory class, since that is where the original anonymous inner class resides.

In the following code, notice that the creation of the anonymous inner class has been altered to create an instance of the new inner class.

```
public void makeStandardItem(int type) {
    if(type==0) {
        // make extremely unusual item .01% of the time
        Item myitem = new StrangeItem();
        myitem.assemble();
    } else {
        // make standard item 99.9% of the time
        Item myitem = new Item();
        myitem.assemble();
    }
}
```

The purpose of using this refactoring is to make your code more reusable and modular. Extracting the anonymous inner class into its own inner class helps improve many aspects of your code. It makes no sense to redefine the same anonymous inner class in multiple places in the Factory class, and the Convert Anonymous to Inner refactoring can help correct the situation.

Extract Method Refactoring

As you review code in a project, you may notice that certain sections of code, even small ones, contain similar looking blocks of code. These blocks of code can be extracted out into a separate method that can then be called. Separating out blocks of code makes your code more readable, more reusable, and easier to maintain.

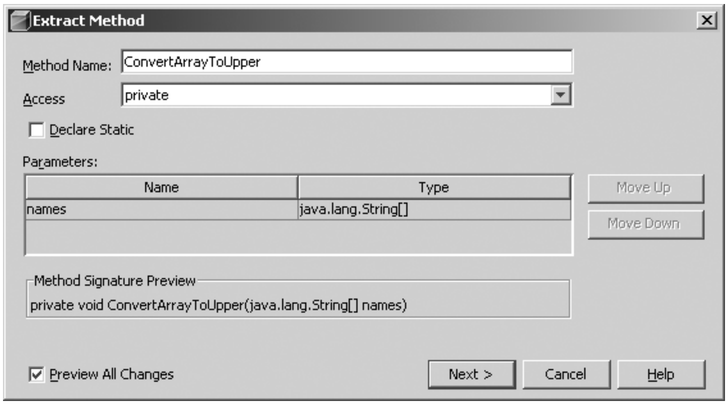
As a simple example, suppose you have the following code:

```
public void processArray(String[] names)
{
    for(int i=0;i < names.length; i++)
    {
        names[i] = names[i].toUpperCase();
    }
    // rest of method here
}
```

This block of code contains a loop that iterates through a `String` array and converts each `String` to uppercase. You might want to put this code into a separate method. The Extract Method refactoring can do this for you.

To activate the refactoring, highlight the code you want to convert to a method and select Refactor ➤ Extract Method. In this example, highlight the entire `for` loop in the `processArray(String[])` method.

In the Extract Method dialog box, you can set the name of the new method, the access level, and whether it should be declared static, as shown in Figure 11-13. The refactoring is even smart enough to assume that a `String` array should be passed into the method and lists it as a parameter for the new method.



Method Name: ConvertArrayToUpper
Access: private
☐ Declare Static
Parameters:

Name	Type
names	java.lang.String[]

Method Signature Preview
private void ConvertArrayToUpper(java.lang.String[] names)

☑ Preview All Changes

Figure 11-13. *The Extract Method dialog box*

After applying the refactoring, the resulting code has the loop split out:

```
public void processArray(String[] names)
{
    ConvertArrayToUpper(names);
    // other method code here
}

private void ConvertArrayToUpper(final String[] names)
{
    for(int i=0;i < names.length; i++)
    {
        names[i] = names[i].toUpperCase();
    }
}
```

You can see that not only has the selected code been extracted out into a separate method, but it was also replaced with the correct call to the new method with the correct parameter.

Extract Interface Refactoring

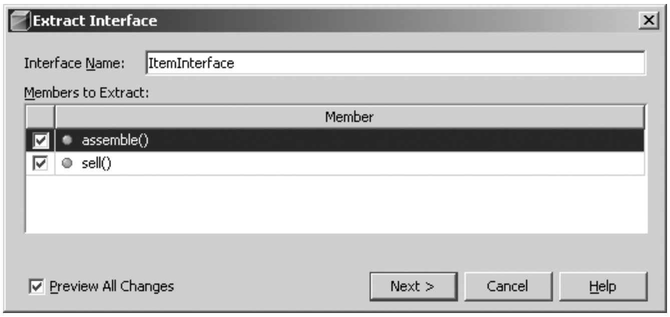
The Extract Interface refactoring allows you to select public non-static methods and move them into an interface. This can be useful as you attempt to make your code more reusable and easier to maintain.

For example, suppose that you want to extract two public non-static methods in the following `Item` class into an interface.

```
public class Item {
    public void assemble() {
        System.out.println("Item.assemble");
    }

    public void sell() {
        System.out.println("sell me");
    }
}
```

You can activate the refactoring by highlighting the class in the Projects window (or by simply having the class open in the Source Editor) and selecting Refactor ➤ Extract Interface.

As shown in Figure 11-14, the options for the Extract Interface refactoring are quite straightforward. You can specify the name of the new interface that will be created. You can also select exactly which methods you want to include in the interface.


Extract Interface

Interface Name: ItemInterface

Members to Extract:

	Member
✓	assemble()
✓	sell()

✓ Preview All Changes

Figure 11-14. *The Extract Interface dialog box*

After applying the refactoring, the code for the interface looks like this:

```
public interface ItemInterface {
    void assemble();
    void sell();
}

public class Item implements ItemInterface {
    public void assemble() {
        System.out.println("Item.assemble");
    }

    public void sell() {
        System.out.println("sell me");
    }
}
```

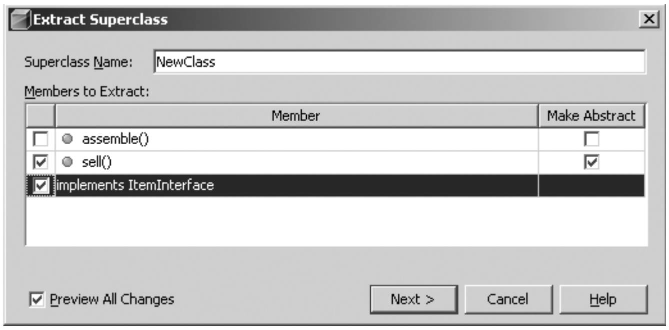
The original `Item` class has been modified to implement the `ItemInterface`.

Extract Superclass Refactoring

The Extract Superclass refactoring is nearly identical to the Extract Interface refactoring. The only difference is that Extract Superclass pulls methods into a newly created superclass and extends the refactored class. Using the refactored code from the previous section as an example, you might want to modify the `Item` class to have a superclass.

```
public class Item implements ItemInterface {
    public void assemble() {
        System.out.println("Item.assemble");
    }

    public void sell() {
        System.out.println("sell me");
    }
}
```

Starting with the `Item` class selected, select Refactor ➤ Extract Superclass. As shown in Figure 11-15, the Extract Superclass dialog box allows you to set the name of the new superclass that will be created. You can select which members you wish to extract and place them in the superclass. Since the `Item` class implements the `ItemInterface`, you can decide if you want to extract the `implements` clause into the superclass. You can also select whether or not the methods that are extracted are made abstract in the superclass. Selecting this option inserts abstract methods into the superclass and leaves the concrete implementations in the `Item` subclass.



Superclass Name: NewClass

Members to Extract:

	Member	Make Abstract
☐	assemble()	☐
✓	sell()	✓
✓	implements ItemInterface	

✓ Preview All Changes

Figure 11-15. *The Extract Superclass dialog box*

For this example, select all the members for extraction. Then select the Make Abstract field only for the `Item.sell()` method. Preview the changes and apply the refactoring. The following code will be generated:

```
public interface ItemInterface {
    void assemble();
    void sell();
}

public abstract class ItemSuperclass implements ItemInterface {

    public void assemble() {
        System.out.println("Item.assemble");
    }

    public abstract void sell();
}

public class Item extends ItemSuperclass {

    public void sell() {
        System.out.println("sell me");
    }
}
```

Now you have an `Item` class with a concrete implementation of the `sell()` method. It extends the `ItemSuperclass`. `ItemSuperclass implements ItemInterface`, and contains an abstract sell() method and a concrete implementation of the assemble() method. ItemInterface contains the definitions of the assemble() and sell() methods.

Using refactoring options like Extract Method, Extract Interface, and Extract Superclass, you can attempt to structure your code to take full advantage of good design principles. Ideally, for new code projects, you would design classes correctly and wouldn't need refactoring. However, many programmers take over projects that have been implemented poorly and need refactoring.

Change Method Parameters Refactoring

The Change Method Parameters refactoring is one of the most useful options in NetBeans. I have made extensive use of it on projects that I have inherited from other developers.

In the old days of development, changing a method signature was time-consuming. You would need to modify the method and then search through all your code to make sure all the references to it were updated. No sooner would you finish that task then you would decide to change the data types on the arguments or rearrange their ordering in the method. The Change Method Parameters refactoring can reduce time spent on such operations.

Suppose you had the following code:

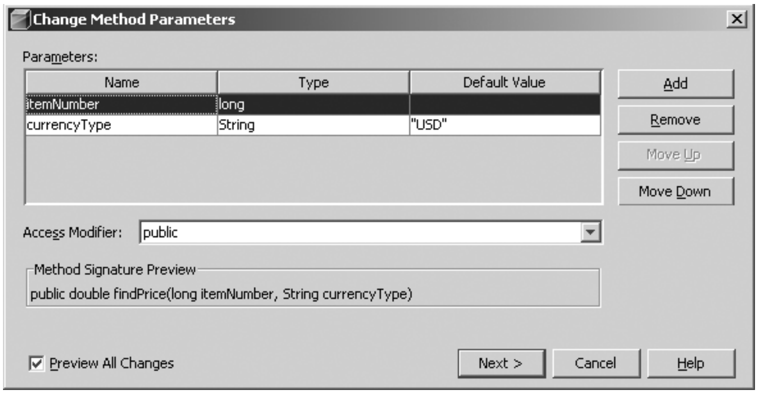
```
public class Item extends ItemSuperclass {

    public void sell() {
        System.out.println("sell me");

        System.out.println("Price(12345) : " + findPrice(12345));
    }

    public double findPrice(long itemNumber) {

        double price = 0.00;
        // look up itemNumber in database and set price variable
        return price;
    }
}
```

The Item class contains a findPrice(long) method. The method accepts an item number, looks it up in a database, and returns a price to the calling sell() method. If your client decided he wants to also be able to return the price and the currency in which it is specified, you would need to alter the findPrice(long) method.

Assume you need to add a String argument to the findPrice(long) method that allows you to specify the type of currency. Highlight the name of the method and select Refactor ➤ Change Method Parameters. In the Change Method Parameters dialog box, you can add and remove parameters to the method. You can also change the order of the parameters and specify the method's new access level.

■ **Tip** You don't have to actually alter the parameters of a method to reorder them. You can use the Change Method Parameters refactoring just to reorder parameters—a task I find myself doing often when I am developing code.

To add the new parameter, click the Add button. A new line appears in the parameters grid. Change the name, type, and default value fields, as shown in Figure 11-16. Then click the Next button, preview the changes, and apply the refactoring.


Change Method Parameters

Parameters:

Name	Type	Default Value
itemNumber	long	
currencyType	String	"USD"

Access Modifier: public

Method Signature Preview

public double findPrice(long itemNumber, String currencyType)

Figure 11-16. *The Change Method Parameters dialog box*

Your refactored code will look like this:

```
public class Item extends ItemSuperclass {

    public void sell() {
        System.out.println("sell me");

        System.out.println("Price(12345) : " + findPrice(12345, "USD"));
    }

    public double findPrice(long itemNumber, String currencyType) {

        double price = 0.00;
        // look up itemNumber in database and set price variable
        return price;
    }
}
```

Notice that the `findPrice(long)` method has been altered to include the new parameter. The `sell()` method has also been altered to call the modified method and pass it the default value of `"USD"`, which was specified during the refactoring operation.

Refactoring Keyboard Shortcuts

At the time of this writing, NetBeans provides only three refactoring keyboard shortcuts, as listed in Table 11-1.

Table 11-1. *Refactoring Keyboard Shortcuts*

Option	Shortcut
Move Class	Alt+Shift+V
Rename	Alt+Shift+R
Extract Method	Ctrl+Shift+M

■ **Tip** You can add your own shortcut for each refactoring option by selecting Tools ➤ Options ➤ Keymap ➤ Refactor. Make sure to explore the existing key mappings to get an idea of what is already used. NetBeans will prevent duplicates from being added.

Refactoring with Jackpot

The Jackpot project was announced by Sun at the JavaOne 2006 conference. For NetBeans 5.0 and 5.5, it is a separate module that needs to be installed. As of NetBeans 6.0, the Jackpot refactoring engine is more tightly integrated with the IDE.

Jackpot provides an impressive set of refactoring capabilities. It is not just a refactoring engine, but provides type-aware modeling, querying, and transformation of Java source files. Jackpot queries can run against large bases of Java source and intelligently manipulate elements, all while maintaining the original code formatting. While this sounds simple, it is actually quite an advanced set of capabilities that makes Jackpot unique. It does not simply use a parsed model of the data, but integrates tightly with the information provided by the Java compiler `javac`.

Adding a Jackpot Refactoring Query

The main way to interface with Jackpot is to write queries in the Jackpot rule language. Written by James Gosling, this rule language is similar to Java's regular expression library, but is type- and Java semantic-aware. Here is the general format of a Jackpot query:

```
pattern expression   =>   replacement expression :: guard/filter expression;
```

The pattern expression is the element or expression you want to attempt to replace. It should be no surprise that the replacement expression is what you want to replace it with. The guard expression is used to match the pattern expression and provide type-aware filtering of matched results. If the guard expression is satisfied, the pattern is substituted with the replacement expression.

You can also use meta-variables to provide wildcard matching in your query expression. A meta-variable is simply a regular Java identifier preceded by a $ character.

Another element of a Jackpot query is a meta-list. A meta-list is essentially the same as a meta-variable, but a meta-list starts and ends with the $ character. Meta-lists also allow more than one expression to be matched.

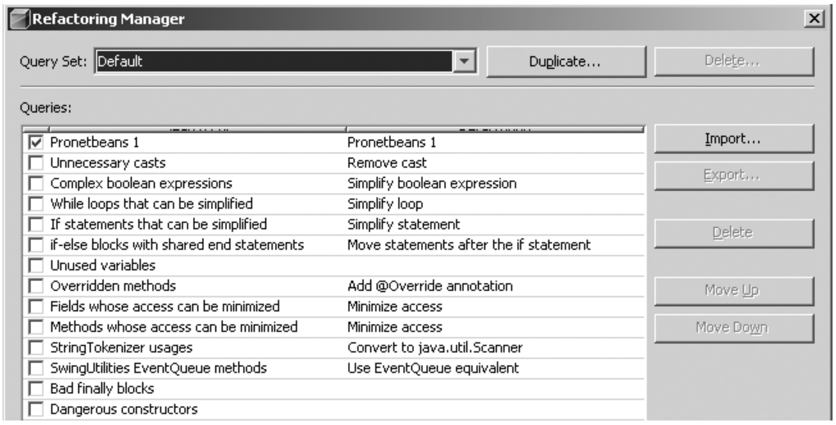
These topics can be a little confusing at first, but you can go online and read more about Jackpot at the official site, `http://jackpot.netbeans.org/`.

Let's start with a simple example. The following is an old piece of date formatting code I once found in an application:

```
Date objStartDate = new Date();
String sDate = objStartDate.toLocaleString();
System.out.println("Date is : " + sDate);
```

In this code fragment, the developer defined and instantiated an instance of the `Date` class. A `String` representation of the system local date was then retrieved and written to the standard output stream. The only problem with this code is that the `toLocaleString()` method is deprecated as of JDK 1.1. It has since been replaced by `DateFormat.format(Date)`.

If you were a developer who was tasked with updating this entire project so it did not use deprecated elements, you could start by adding a Jackpot query through the Refactoring Manager. The Refactoring Manager is the main user interface for working with Jackpot query sets, or groups of related queries, as shown in Figure 11-17. You can open it by selecting Tools ➤ Refactoring Manager.


Refactoring Manager

Query Set: Default

Queries:

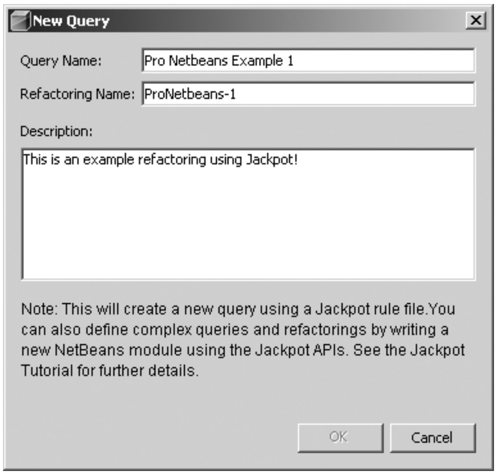
☑	Pronetbeans 1	Pronetbeans 1
☐	Unnecessary casts	Remove cast
☐	Complex boolean expressions	Simplify boolean expression
☐	While loops that can be simplified	Simplify loop
☐	If statements that can be simplified	Simplify statement
☐	if-else blocks with shared end statements	Move statements after the if statement
☐	Unused variables	
☐	Overridden methods	Add @Override annotation
☐	Fields whose access can be minimized	Minimize access
☐	Methods whose access can be minimized	Minimize access
☐	StringTokenizer usages	Convert to java.util.Scanner
☐	SwingUtilities EventQueue methods	Use EventQueue equivalent
☐	Bad finally blocks	
☐	Dangerous constructors	

Figure 11-17. *The Jackpot Refactoring Manager*

Initially, Jackpot has two query sets: Default and Effective Java Items. Select a query set to see its queries listed in the Refactoring Manager.

You can create your own set of queries by clicking the Duplicate button next to the query set name and assigning a name to the new query set. Then you can add, edit, and delete queries in whatever manner you wish.

To add a new query, click the New Query button. Fill in the query name, refactoring name, and description, as shown in Figure 11-18.



Query Name: Pro Netbeans Example 1

Refactoring Name: ProNetbeans-1

Description: This is an example refactoring using Jackpot!

Note: This will create a new query using a Jackpot rule file.You can also define complex queries and refactorings by writing a new NetBeans module using the Jackpot APIs. See the Jackpot Tutorial for further details.

Figure 11-18. *The Jackpot New Query dialog box*

Once you click the OK button, the query name is added to the list in the Refactoring Manager. To actually add the query rule, highlight the query name and click the Edit Query button. At this point, a new code window will open in the Source Editor. The name of the file open in the Source Editor is the name you assigned in the New Query dialog box (`Pro NetBeans Example 1` in the example in Figure 11-18), with the file extension of `.rules` (`Pro NetBeans Example 1.rules`).

Enter the query expression you would like the new query to contain, such as the following:

```
$object.toLocaleString() => java.text.DateFormat.format($object) :: ➥
$object instanceof java.util.Date;
```

This query rule replaces the deprecated method `Date.toLocaleString()` with a more correct way to format the `Date` object using `DateFormat.format(Date)`. The pattern expression to search for is `$object.toLocaleString`. If the `$object` is an instance of the `java.util.Date` class, the fragments that use the `toLocaleString()` method will be replaced with the `DateFormat.format(Date)` method.

After you've entered the query rule, save the file and close the code window. You now have a valid Jackpot query that can be run against your code.

Running a Jackpot Refactoring Query

To use Jackpot refactoring, with a Java project open and selected, choose Refactoring ➤ Query and Refactor. In the Query and Refactor dialog box, you can choose to apply an entire query set or just a single query to the project code. Select the query and click the Query button.

Jackpot then runs the refactoring query against your code. If no results are found, a status message appears in the lower-left status bar. If the query matched code in your project, you'll see a list of pending changes. None of the refactorings are applied until you click the Do Refactoring button.

Click the Do Refactoring button to apply the changes. Once the refactoring has been executed, you can review the source code. The example shown at the beginning of this section would now look like this:

```
Date objStartDate = new Date();
String sDate = DateFormat.format(objStartDate);
System.out.println("Date is : " + sDate);
```

This is a very simple example. You can use Jackpot queries to refactor poorly designed code and help apply good design principles to your project.

As you become more experienced with Jackpot, you will find it easier to write query rules to scan for bad sections of code. The benefits of this refactoring capability are well worth the time you spend learning the Jackpot rule language and writing queries.

Summary

In this chapter, you saw the wide variety of refactoring options available in the NetBeans IDE. You can use them to rework existing code or to make your new coding smoother.

Some of these refactorings will obviously be used more often than others, but you should become familiar with when and how to use each one. Using these refactoring options when working with large code bases can be a lifesaver.