Java Tutorials

Updated for Java SE 8

ORACLE

Java Tutorials

The Java Tutorials are a one stop shop for learning the Java language. The tutorials are practical guides for programmers who want to use the Java programming language to create applets and applications. They also serve as a reference for the experienced Java programmer.

Groups of related lessons are organized into "trails"; for example, the "Getting Started" trail, or the "Learning the Java Language" trail. This ebook contains a single trail. You can download other trails via the link on the Java Tutorials home page.

To read an ebook file that you have downloaded, transfer the document to your device.

We would love to hear what you think of this ebook, including any problems that you find. Please send us your feedback.

Legal Notices

Copyright 1995, 2014, Oracle Corporation and/or its affiliates (Oracle). All rights reserved.

This tutorial is a guide to developing applications for the Java Platform, Standard Edition and contains documentation (Tutorial) and sample code. The sample code made available with this Tutorial is licensed separately to you by Oracle under the Berkeley license. If you download any such sample code, you agree to the terms of the Berkeley license.

This Tutorial is provided to you by Oracle under the following license terms containing restrictions on use and disclosure and is protected by intellectual property laws. Oracle grants to you a limited, non-exclusive license to use this Tutorial for information purposes only, as an aid to learning about the Java SE platform. Except as expressly permitted in these license terms, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means this Tutorial. Reverse engineering, disassembly, or decompilation of this Tutorial is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If the Tutorial is licensed on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This Tutorial is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this Tutorial in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use.

THE TUTORIAL IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND. ORACLE FURTHER DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT.

IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF ORACLE HAS BEEN

ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ORACLES ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).

No Technical Support

Oracles technical support organization will not provide technical support, phone support, or updates to you.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

The sample code and Tutorial may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Supported Platforms

The `mobi` file format works best on the following devices:

- Kindle Fire
- Kindle DX

The `ePub` file format works best on the following devices:

- iPad
- Nook
- Other eReaders that support the `ePub` format.

For best results when viewing preformatted code blocks, adjust the landscape/portrait orientation and font size of your device to enable the maximum possible viewing area.

Trail: JavaBeans(TM)

JavaBeans makes it easy to reuse software components. Developers can use software components written by others without having to understand their inner workings.

To understand why software components are useful, think of a worker assembling a car. Instead of building a radio from scratch, for example, she simply obtains a radio and hooks it up with the rest of the car.

This trail describes JavaBeans using the following lessons:

 **Quick Start** provides a speedy introduction to JavaBeans by showing how to build applications with NetBeans.

 **Writing JavaBeans Components** describes the coding patterns used for bean properties, methods, and events. It also outlines the use of a `BeanInfo` to customize the development experience in a builder tool.

 **Advanced JavaBeans Topics** covers bean persistence, long term persistence, and customization.

Trail: JavaBeans(TM): Table of Contents

Lesson: Quick Start

This lesson describes how to use NetBeans to build a simple application. With a good tool like NetBeans, you can assemble JavaBeans components into an application without having to write any code.

The first three pages of this lesson show how to create a simple application using graphic beans that are part of the Java platform. The last page demonstrates how easy it is to incorporate a third-party bean into your application.

- Creating a Project describes the steps for setting up a new project in NetBeans.
- A Button is a Bean shows how to add a bean to the application's user interface and describes properties and events.
- Wiring the Application covers using NetBeans to respond to bean events in your application.
- Using a Third-Party Bean show how easy it is to add a new bean to the palette and use it in your application.

Note: See online version of topics in this ebook to download complete source code.

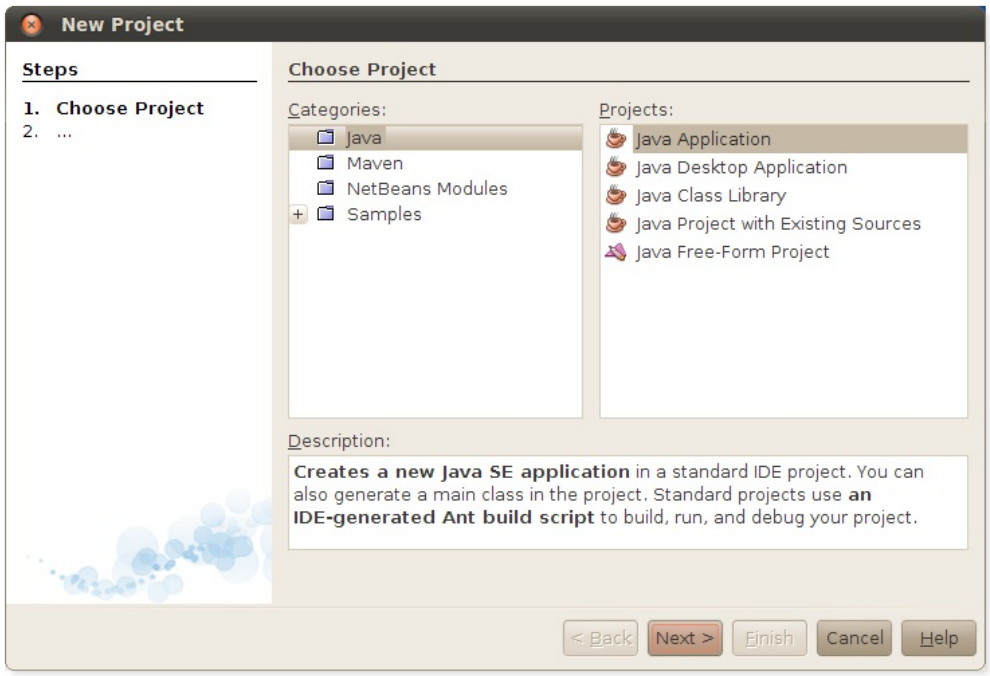
Creating a Project

The easiest way to learn about JavaBeans is to start using them. To begin, download and install the latest version of NetBeans. This tutorial describes how to use NetBeans version 7.0.

NetBeans is a *bean builder tool*, which means it recognizes JavaBeans components (beans) and enables you to snap components together into an application with ease.

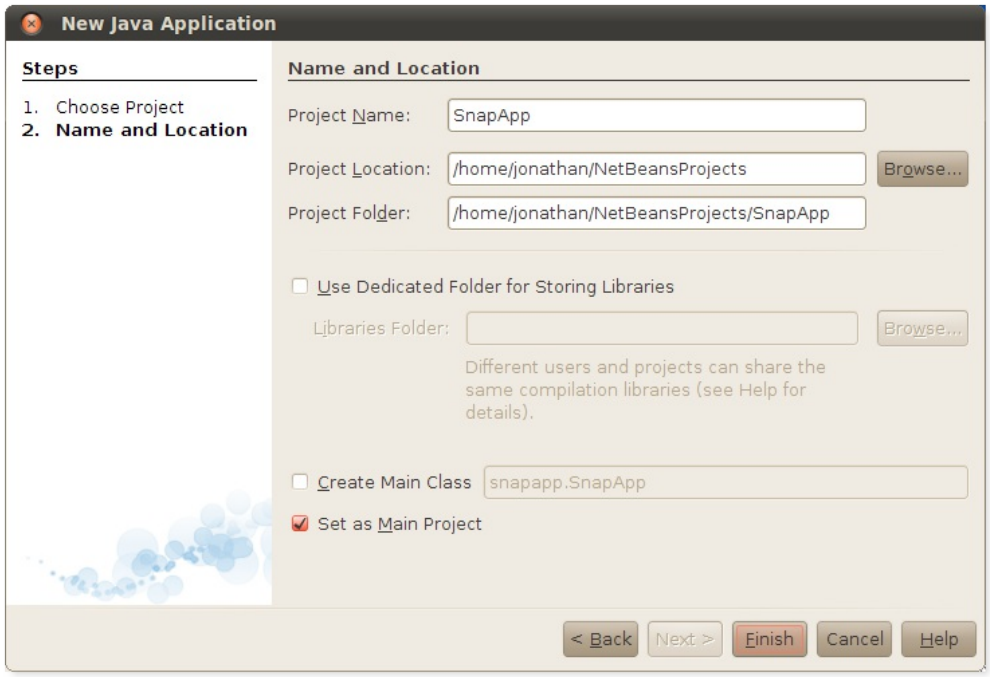
A Button is a Bean

Start NetBeans. Choose **File > New Project...** from the menu.



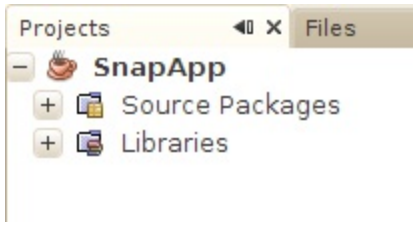
Click for full image

Select **Java** from the **Categories** list and select **Java Application** from the **Projects** list. Click **Next >**.

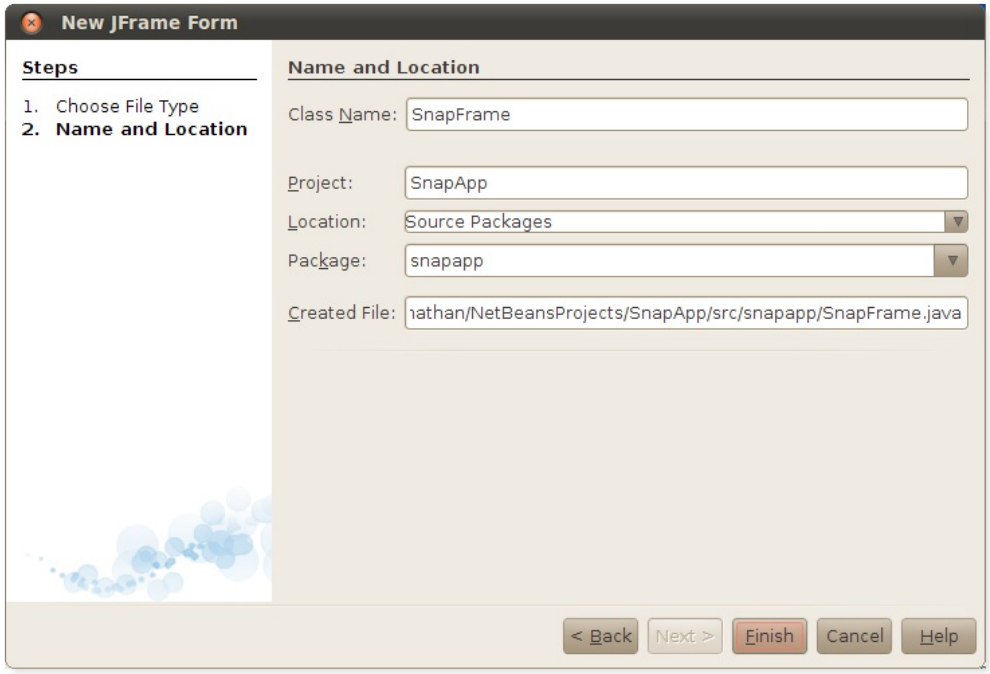


Click for full image

Enter **SnapApp** as the application name. Uncheck **Create Main Class** and click **Finish**. NetBeans creates the new project and you can see it in NetBeans' **Projects** pane:



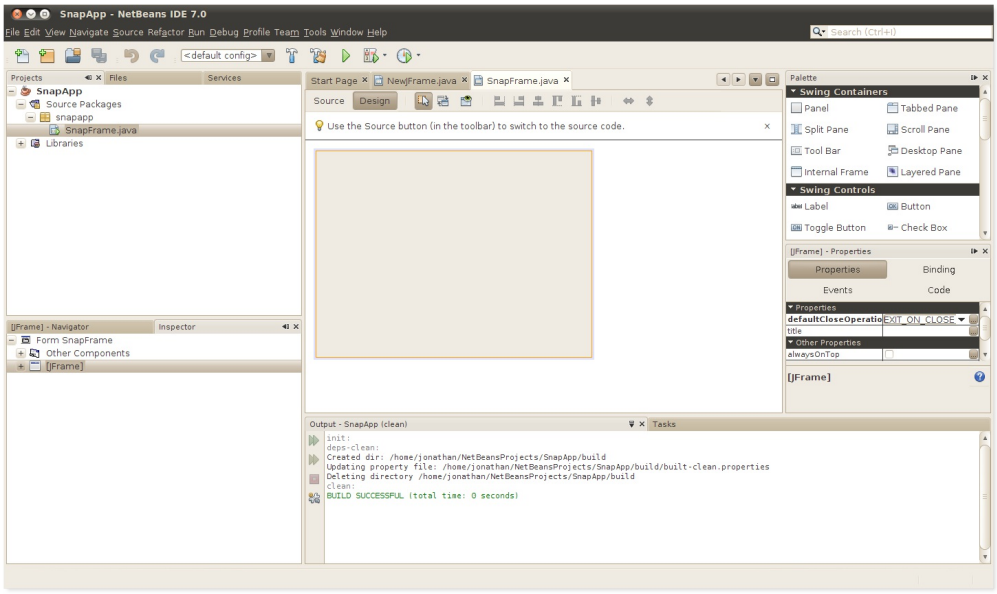
Control-click on the **SnapApp** project and choose **New > JFrame Form...** from the popup menu.



Click for full image

Fill in **SnapFrame** for the class name and **snapapp** as the package. Click **Finish**. NetBeans creates the

new class and shows its visual designer:

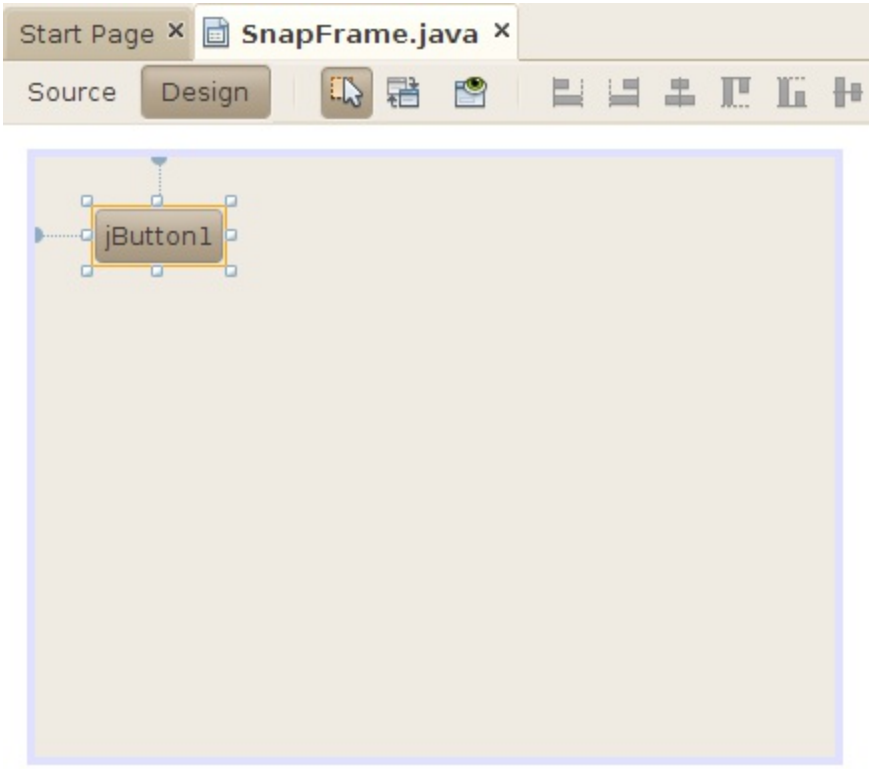


Click for full image

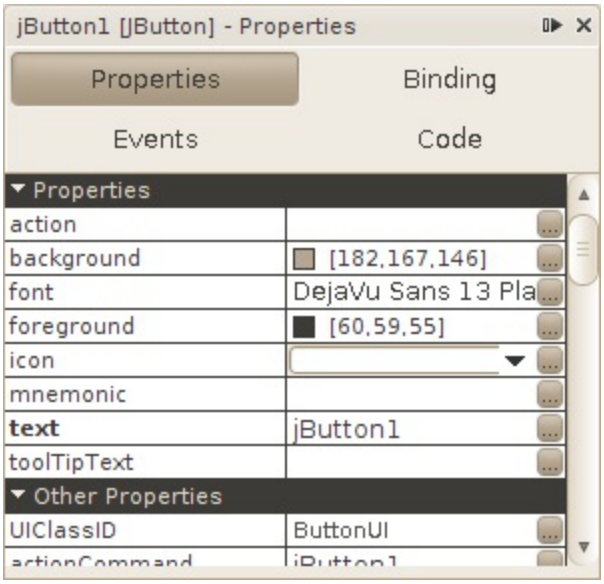
In the **Projects** pane on the left, you can see the newly created `SnapFrame` class. In the center of the screen is the NetBeans visual designer. On the right side is the **Palette**, which contains all the components you can add to the frame in the visual designer.

A Button is a Bean

Take a closer look at the **Palette**. All of the components listed are beans. The components are grouped by function. Scroll to find the **Swing Controls** group, then click on **Button** and drag it over into the visual designer. The button is a bean!



Under the palette on the right side of NetBeans is an inspector pane that you can use to examine and manipulate the button. Try closing the output window at the bottom to give the inspector pane more space.



Properties

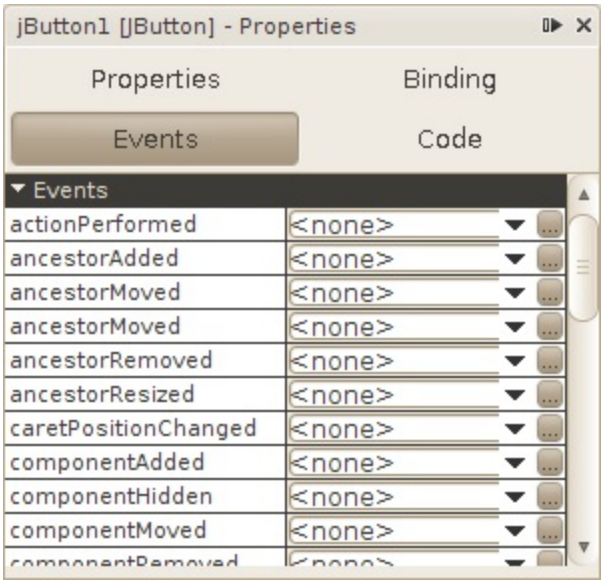
The properties of a bean are the things you can change that affect its appearance or internal state. For the button in this example, the properties include the foreground color, the font, and the text that

appears on the button. The properties are shown in two groups. **Properties** lists the most frequently used properties, while **Other Properties** shows less commonly used properties.

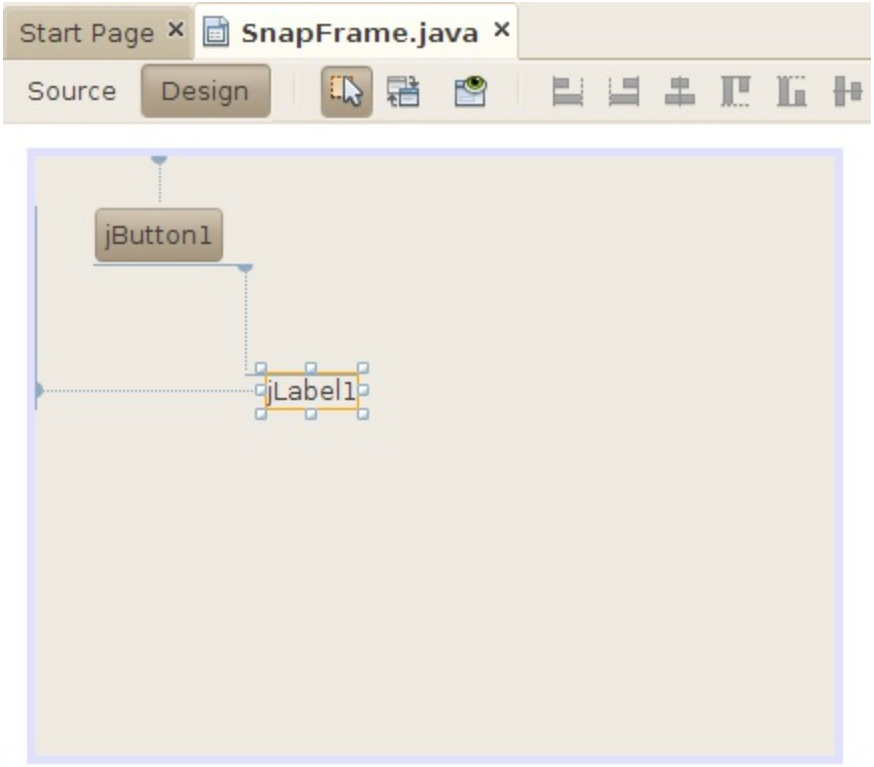
Go ahead and edit the button's properties. For some properties, you can type values directly into the table. For others, click on the **...** button to edit the value. For example, click on **...** to the right of the **foreground** property. A color chooser dialog pops up and you can choose a new color for the foreground text on the button. Try some other properties to see what happens. Notice you are not writing any code.

Events

Beans can also fire events. Click on the **Events** button in the bean properties pane. You'll see a list of every event that the button is capable of firing.



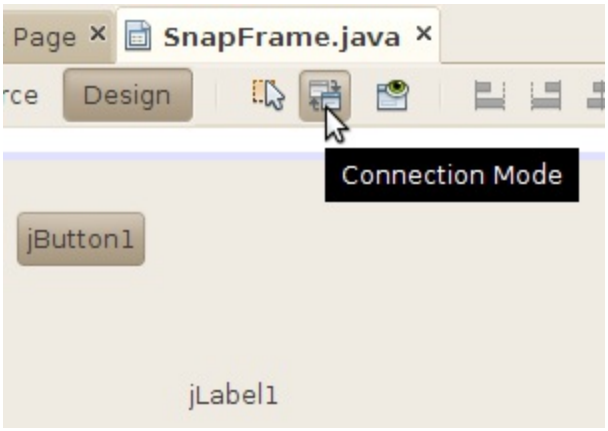
You can use NetBeans to hook up beans using their events and properties. To see how this works, drag a **Label** out of the palette into the visual designer for `SnapFrame`.



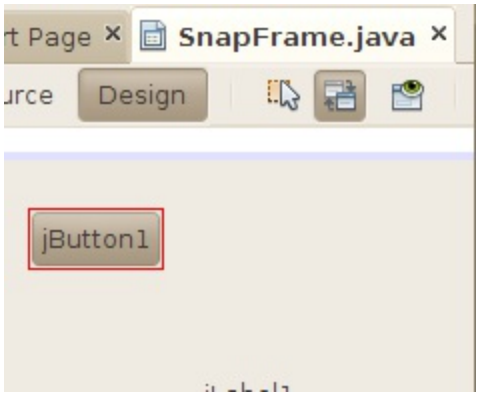
Edit the label's properties until it looks just perfect.

Wiring the Application

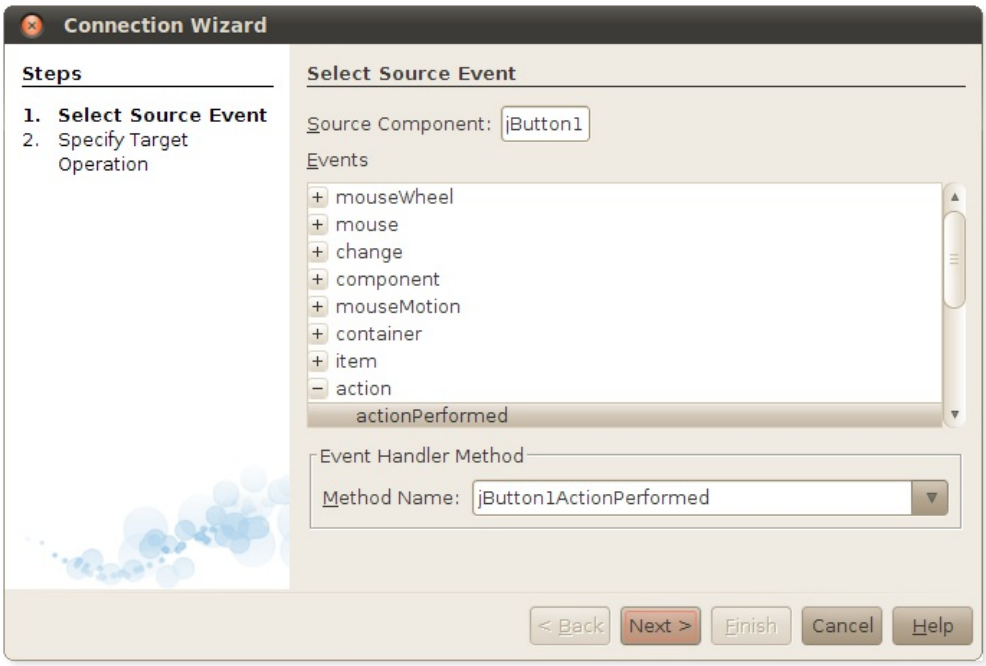
To wire the button and the label together, click on the **Connection Mode** button in the visual designer toolbar.



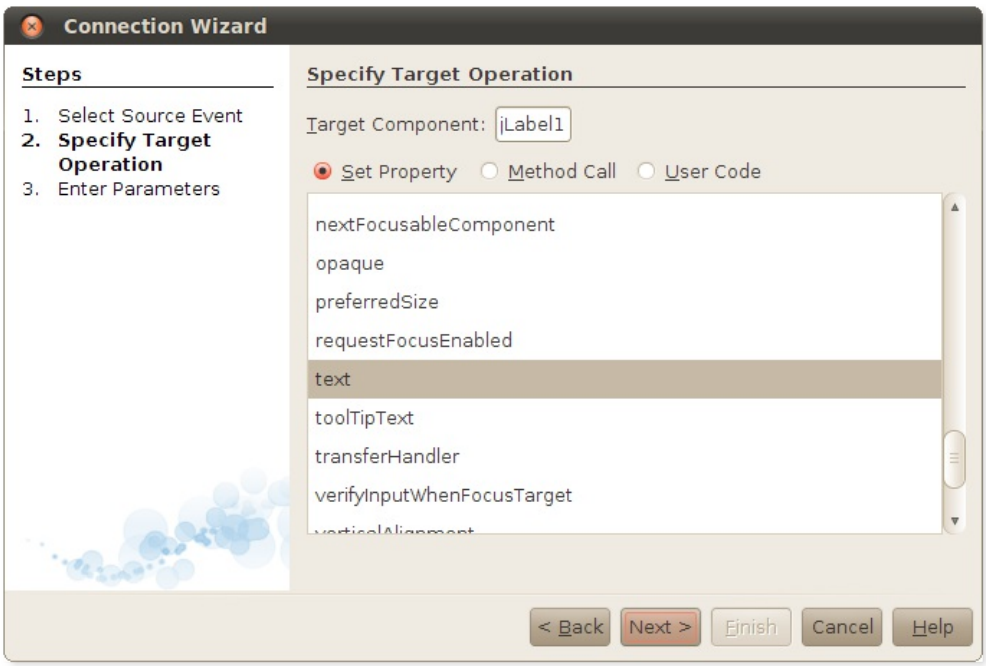
Click on the button in the `SnapFrame` form. NetBeans outlines the button in red to show that it is the component that will be generating an event.



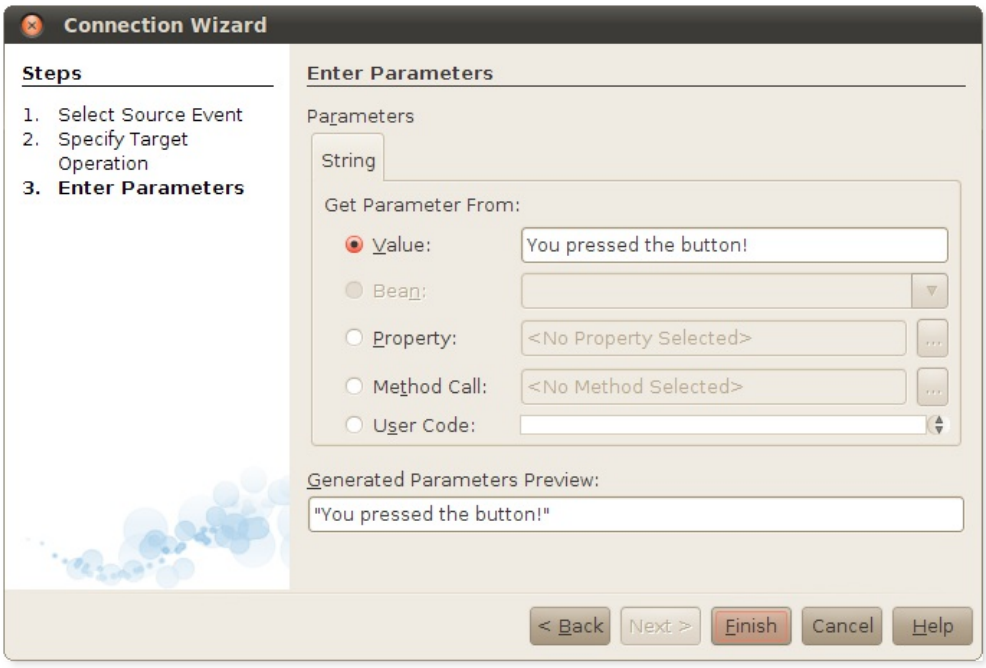
Click on the label. NetBeans' **Connection Wizard** pops up. First you will choose the event you wish to respond to. For the button, this is the **action** event. Click on the + next to **action** and select **actionPerformed**. Click **Next >**.



Now you get to choose what happens when the button fires its **action** event. The **Connection Wizard** lists all the properites in the label bean. Select **text** in the list and click **Next**.



In the final screen of the **Connection Wizard**, fill in the value you wish to set for the **text** property. Click on **Value**, then type **You pressed the button!** or something just as eloquent. Click **Finish**.



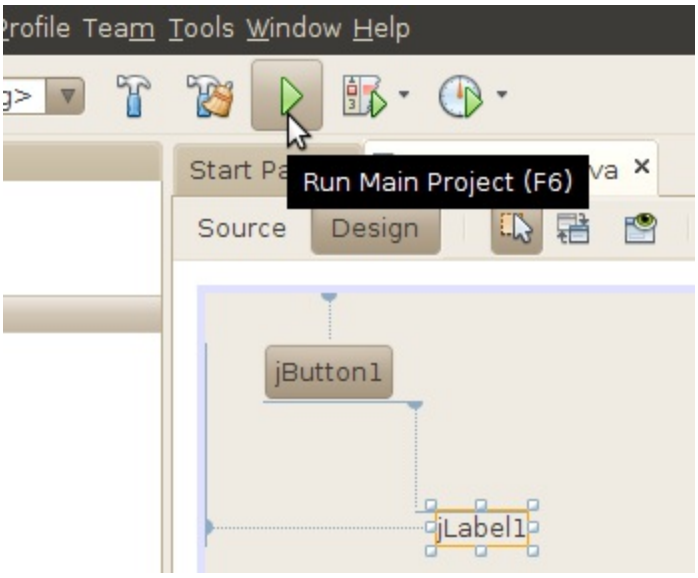
NetBeans wires the components together and shows you its handiwork in the source code editor.

```
    @SuppressWarnings("unchecked")
    Generated Code

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        jLabel1.setText("You pressed the button!");
    }

    /**
```

Click on the **Design** button in the source code toolbar to return to the UI designer. Click **Run Main Project** or press **F6** to build and run your project.



NetBeans builds and runs the project. It asks you to identify the main class, which is `SnapFrame`. When the application window pops up, click on the button. You'll see your immortal prose in the label.

Notice that you did not write any code. This is the real power of JavaBeans with a good builder tool like NetBeans, you can quickly wire together components to create a running application.

Using a Third-Party Bean

Almost any code can be packaged as a bean. The beans you have seen so far are all visual beans, but beans can provide functionality without having a visible component.

The power of JavaBeans is that you can use software components without having to write them or understand their implementation.

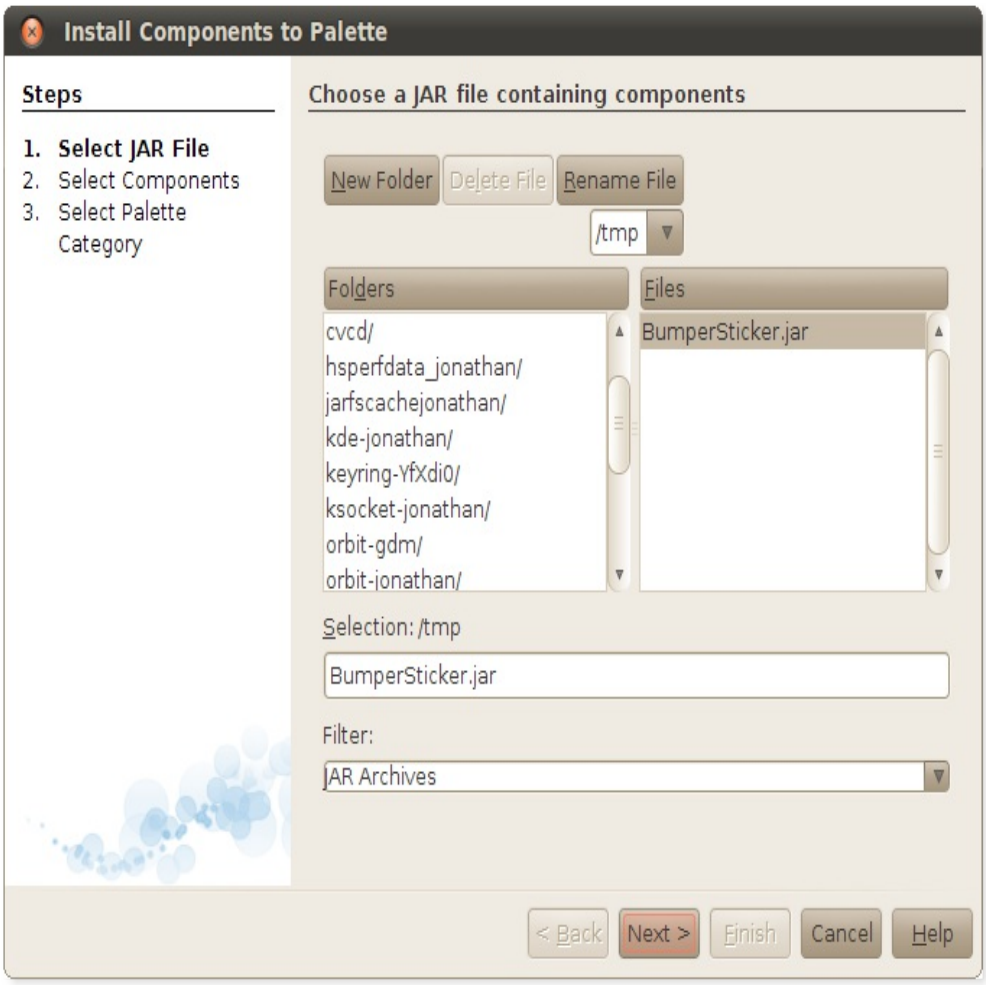
This page describes how you can add a JavaBean to your application and take advantage of its functionality.

Adding a Bean to the NetBeans Palette

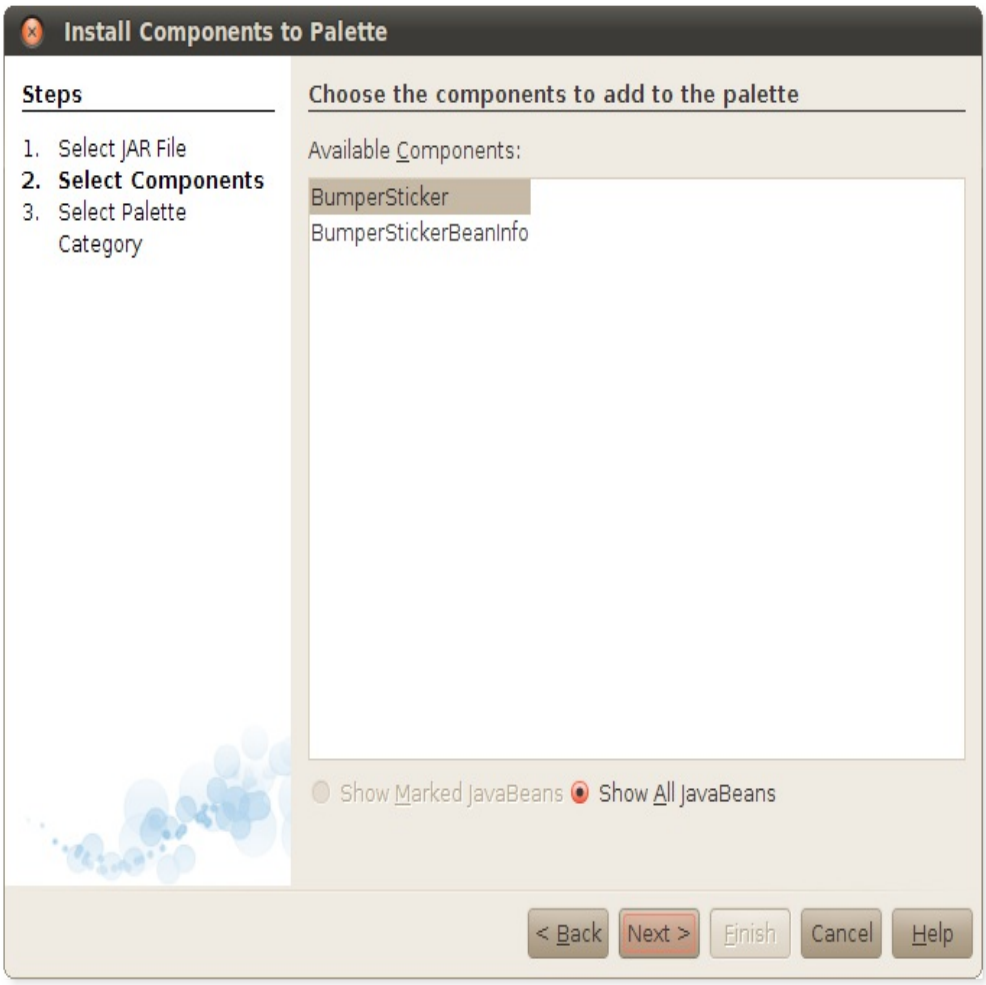
Download `an example JavaBean component, BumperSticker`. Beans are distributed as JAR files. Save the file somewhere on your computer. `BumperSticker` is graphic component and exposes one method, `go()`, that kicks off an animation.

To add `BumperSticker` to the NetBeans palette, choose **Tools > Palette > Swing/AWT Components** from the NetBeans menu.

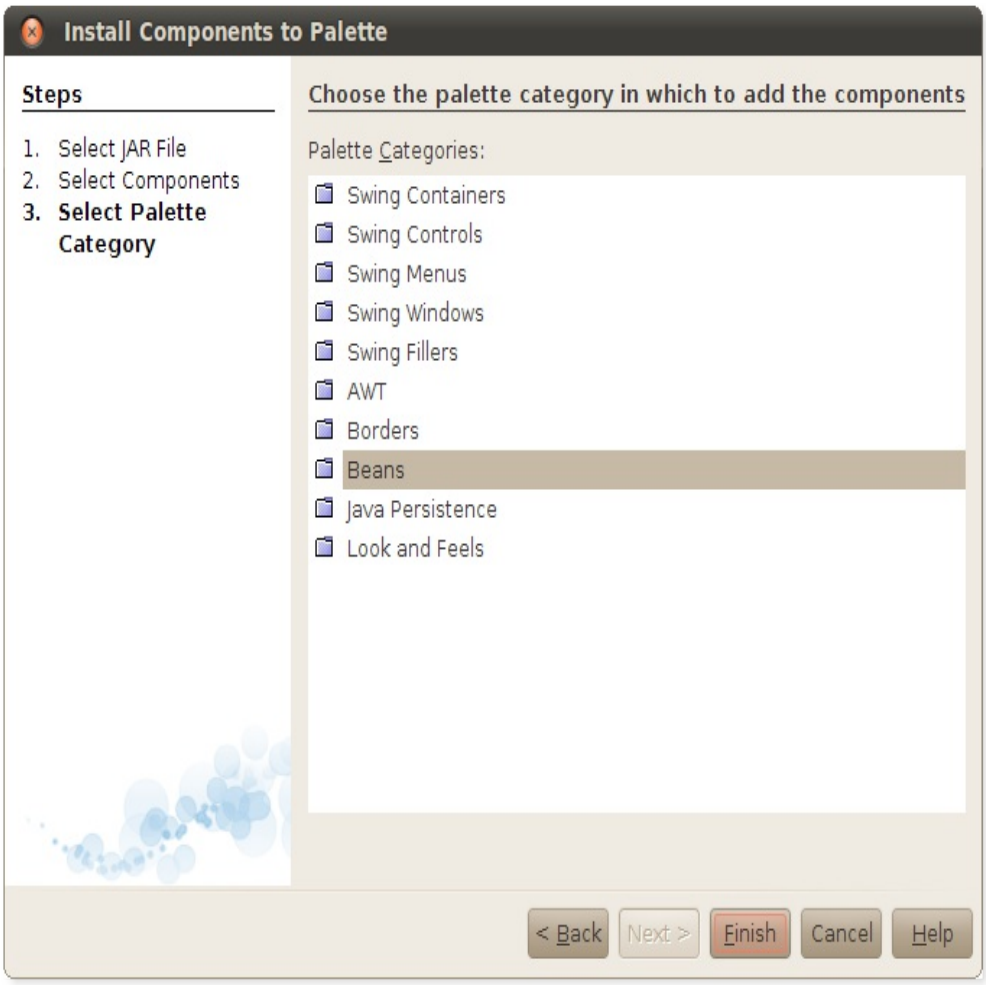
Click on the **Add from JAR...** button. NetBeans asks you to locate the JAR file that contains the beans you wish to add to the palette. Locate the file you just downloaded and click **Next**.



NetBeans shows a list of the classes in the JAR file. Choose the ones you wish you add to the palette. In this case, select **BumperSticker** and click **Next**.



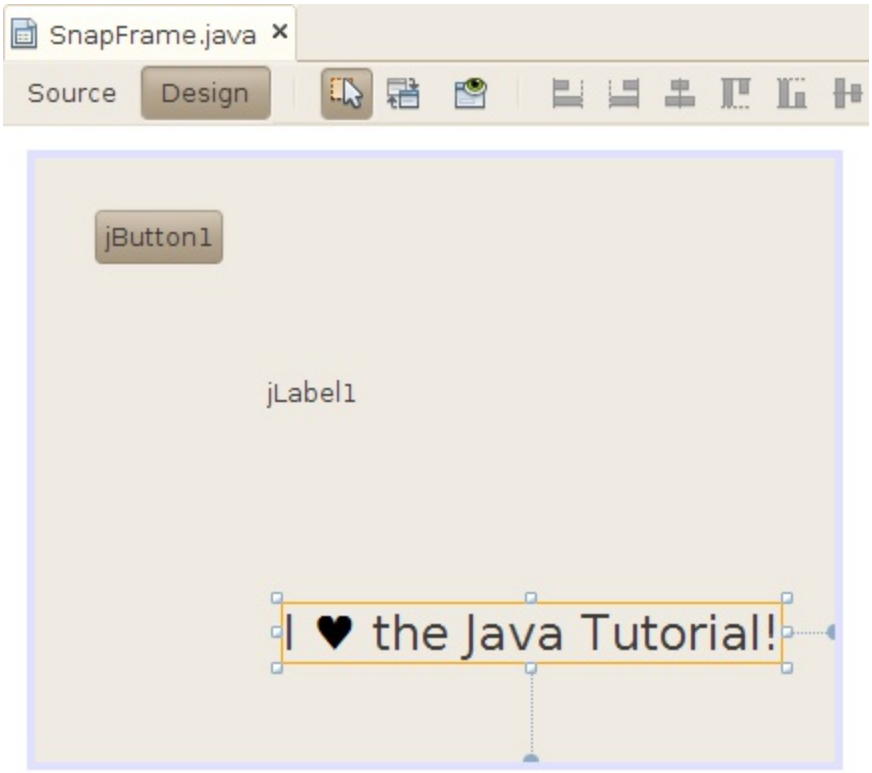
Finally, NetBeans needs to know which section of the palette will receive the new beans. Choose **Beans** and click **Finish**.



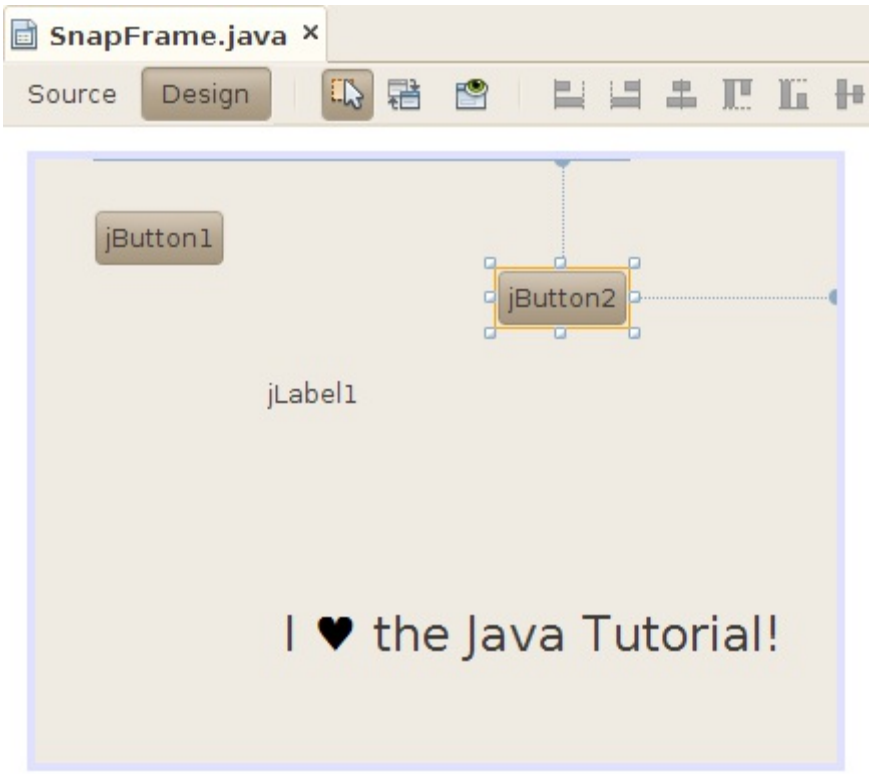
Click **Close** to make the **Palette Manager** window go away. Now take a look in the palette. `BumperSticker` is there in the **Beans** section.

Using Your New JavaBean

Go ahead and drag `BumperSticker` out of the palette and into your form.



You can work with the `BumperSticker` instance just as you would work with any other bean. To see this in action, drag another button out into the form. This button will kick off the `BumperSticker`'s animation.



Wire the button to the `BumperSticker` bean, just as you already wired the first button to the text field.

1. Begin by clicking on the **Connection Mode** button.
2. Click on the second button. NetBeans gives it a red outline.
3. Click on the `BumperSticker` component. The **Connection Wizard** pops up.

4. Click on the + next to **action** and select **actionPerformed**. Click **Next >**.
5. Select **Method Call**, then select **go()** from the list. Click **Finish**.

If you're unsure of any of the steps, review Wiring the Application. The process here is very similar.

Run the application again. When you click on the second button, the `BumperSticker` component animates the color of the heart.

Again, notice how you have produced a functioning application without writing any code.

Lesson: Writing JavaBeans Components

Writing JavaBeans components is surprisingly easy. You don't need a special tool and you don't have to implement any interfaces. Writing beans is simply a matter of following certain coding conventions. All you have to do is make your class *look* like a bean tools that *use* beans will be able to recognize and use your bean.

However, NetBeans provides some features that make it easier to write beans. In addition, the Java SE API includes some support classes to help implement common tasks.

The code examples in this lesson are based on a simple graphic component called `FaceBean`.

FaceBean source code only:

`FaceBean.java`

Entire NetBeans project including FaceBean source code:

`FaceBean.zip`

A bean is a Java class with method names that follow the JavaBeans guidelines. A bean builder tool uses *introspection* to examine the bean class. Based on this inspection, the bean builder tool can figure out the bean's properties, methods, and events.

The following sections describe the JavaBeans guidelines for properties, methods, and events. Finally, a section on `BeanInfo` shows how you can customize the developer's experience with your bean.

Note: See online version of topics in this ebook to download complete source code.

Properties

To define a property in a bean class, supply public getter and setter methods. For example, the following methods define an `int` property called `mouthWidth`:

```
public class FaceBean {
    private int mMouthWidth = 90;

    public int getMouthWidth() {
        return mMouthWidth;
    }

    public void setMouthWidth(int mw) {
        mMouthWidth = mw;
    }
}
```

A builder tool like NetBeans recognizes the method names and shows the `mouthWidth` property in its list of properties. It also recognizes the type, `int`, and provides an appropriate editor so the property can be manipulated at design time.

This example shows a property than can be read and written. Other combinations are also possible. A read-only property, for example, has a getter method but no setter. A write-only property has a setter method only.

A special case for `boolean` properties allows the accessor method to be defined using `is` instead of `get`. For example, the accessor for a `boolean` property `running` could be as follows:

```
public boolean isRunning() {
    // ...
}
```

Various specializations of basic properties are available and described in the following sections.

Indexed Properties

An *indexed* property is an array instead of a single value. In this case, the bean class provides a method for getting and setting the entire array. Here is an example for an `int[]` property called `testGrades`:

```
public int[] getTestGrades() {
    return mTestGrades;
}

public void setTestGrades(int[] tg) {
    mTestGrades = tg;
}
```

For indexed properties, the bean class also provides methods for getting and setting a specific element of the array.

```
public int getTestGrades(int index) {
    return mTestGrades[index];
}

public void setTestGrades(int index, int grade) {
    mTestGrades[index] = grade;
}
```

Bound Properties

A *bound* property notifies listeners when its value changes. This has two implications:

1. The bean class includes `addPropertyChangeListener()` and `removePropertyChangeListener()` methods for managing the bean's listeners.
2. When a bound property is changed, the bean sends a `PropertyChangeEvent` to its registered listeners.

`PropertyChangeEvent` and `PropertyChangeListener` live in the `java.beans` package.

The `java.beans` package also includes a class, `PropertyChangeSupport`, that takes care of most of the work of bound properties. This handy class keeps track of property listeners and includes a convenience method that fires property change events to all registered listeners.

The following example shows how you could make the `mouthWidth` property a bound property using `PropertyChangeSupport`. The necessary additions for the bound property are shown in bold.

```
import java.beans.*;

public class FaceBean {
    private int mMouthWidth = 90;
    private PropertyChangeSupport mPcs =
        new PropertyChangeSupport(this);

    public int getMouthWidth() {
        return mMouthWidth;
    }

    public void setMouthWidth(int mw) {
        int oldMouthWidth = mMouthWidth;
        mMouthWidth = mw;
        mPcs.firePropertyChange("mouthWidth",
                                oldMouthWidth, mw);
    }

    public void
    addPropertyChangeListener(PropertyChangeListener listener) {
        mPcs.addPropertyChangeListener(listener);
    }

    public void
    removePropertyChangeListener(PropertyChangeListener listener) {
        mPcs.removePropertyChangeListener(listener);
    }
}
```

Bound properties can be tied directly to other bean properties using a builder tool like NetBeans. You could, for example, take the `value` property of a slider component and bind it to the `mouthWidth` property shown in the example. NetBeans allows you to do this without writing any code.

Constrained Properties

A *constrained* property is a special kind of bound property. For a constrained property, the bean keeps track of a set of *veto* listeners. When a constrained property is about to change, the listeners are consulted about the change. Any one of the listeners has a chance to veto the change, in which case the property remains unchanged.

The veto listeners are separate from the property change listeners. Fortunately, the `java.beans` package includes a `VetoableChangeSupport` class that greatly simplifies constrained properties.

Changes to the `mouthWidth` example are shown in bold:

```
import java.beans.*;

public class FaceBean {
    private int mMouthWidth = 90;
    private PropertyChangeSupport mPcs =
        new PropertyChangeSupport(this);
    private VetoableChangeSupport mVcs =
        new VetoableChangeSupport(this);

    public int getMouthWidth() {
        return mMouthWidth;
    }

    public void
    setMouthWidth(int mw) throws PropertyVetoException {
        int oldMouthWidth = mMouthWidth;
        mVcs.fireVetoableChange("mouthWidth",
                                oldMouthWidth, mw);
        mMouthWidth = mw;
        mPcs.firePropertyChange("mouthWidth",
                                oldMouthWidth, mw);
    }

    public void
    addPropertyChangeListener(PropertyChangeListener listener) {
        mPcs.addPropertyChangeListener(listener);
    }

    public void
    removePropertyChangeListener(PropertyChangeListener listener) {
        mPcs.removePropertyChangeListener(listener);
    }

    public void
    addVetoableChangeListener(VetoableChangeListener listener) {
        mVcs.addVetoableChangeListener(listener);
    }

    public void
    removeVetoableChangeListener(VetoableChangeListener listener) {
        mVcs.removeVetoableChangeListener(listener);
    }
}
```

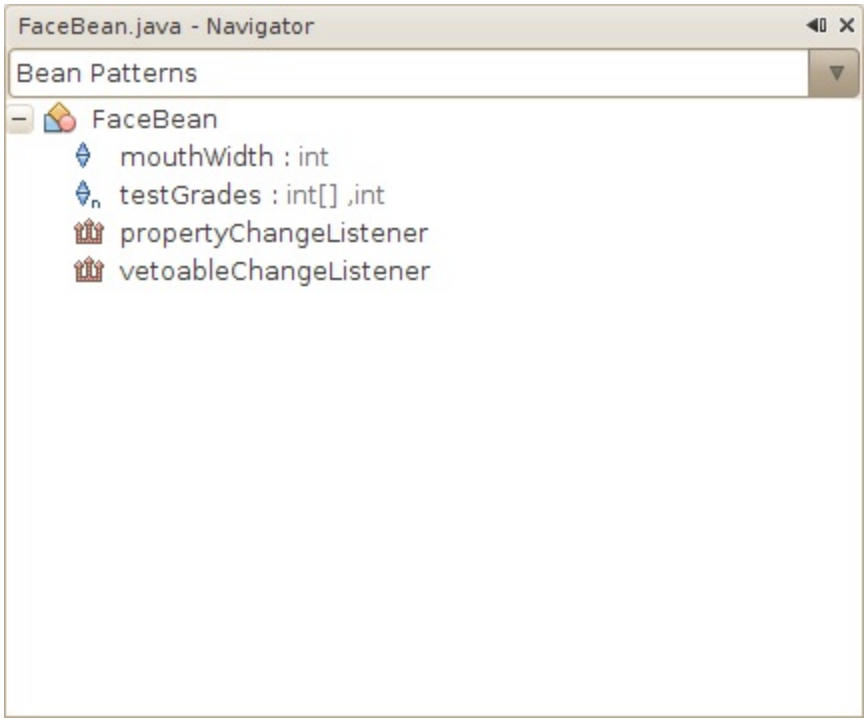
Development Support in NetBeans

The coding patterns for creating bean properties are straightforward, but sometimes it's hard to tell if you are getting everything correct. NetBeans has support for property patterns so you can immediately see results as you are writing code.

To take advantage of this feature, look at the **Navigator** pane, which is typically in the lower left corner of the NetBeans window. Normally, this pane is in **Members View** mode, which shows all the methods and fields defined in the current class.

Click on the combo box to switch to **Bean Patterns** view. You will see a list of the properties that NetBeans can infer from your method definitions. NetBeans updates this list as you type, making it a handy way to check your work.

In the following example, NetBeans has found the read-write `mouthWidth` property and the read-write indexed `testGrades` property. In addition, NetBeans has recognized that `FaceBean` allows registration of both `PropertyChangeListener`s and `VetoableChangeListener`s.



Methods

A bean's *methods* are the things it can do. Any public method that is not part of a property definition is a bean method. When you use a bean in the context of a builder tool like NetBeans, you can use a bean's methods as part of your application. For example, you could wire a button press to call one of your bean's methods.

Events

A bean class can fire off any type of event, including custom events. As with properties, events are identified by a specific pattern of method names.

```
public void add<Event>Listener(<Event>Listener a)
public void remove<Event>Listener(<Event>Listener a)
```

The listener type must be a descendant of `java.util.EventListener`.

For example, a Swing `JButton` is a bean that fires `action` events when the user clicks on it. `JButton` includes the following methods (actually inherited from `AbstractButton`), which are the bean pattern for an event:

```
public void addActionListener(ActionListener l);
public void removeActionListener(ActionListener l);
```

Bean events are recognized by builder tools and can be used in wiring components together. For example, you can wire a button's `action` event to make something happen, like invoking another bean's method.

Using a BeanInfo

Beans, especially graphic components, can have a dizzying number of properties. If your class inherits from `Component`, or `JComponent`, or other Swing classes, it will have over one hundred properties already. Although a builder tool like NetBeans makes it easy to edit bean properties, it can be hard to find the right properties to edit, especially for inexperienced programmers.

Overview of `BeanInfo`

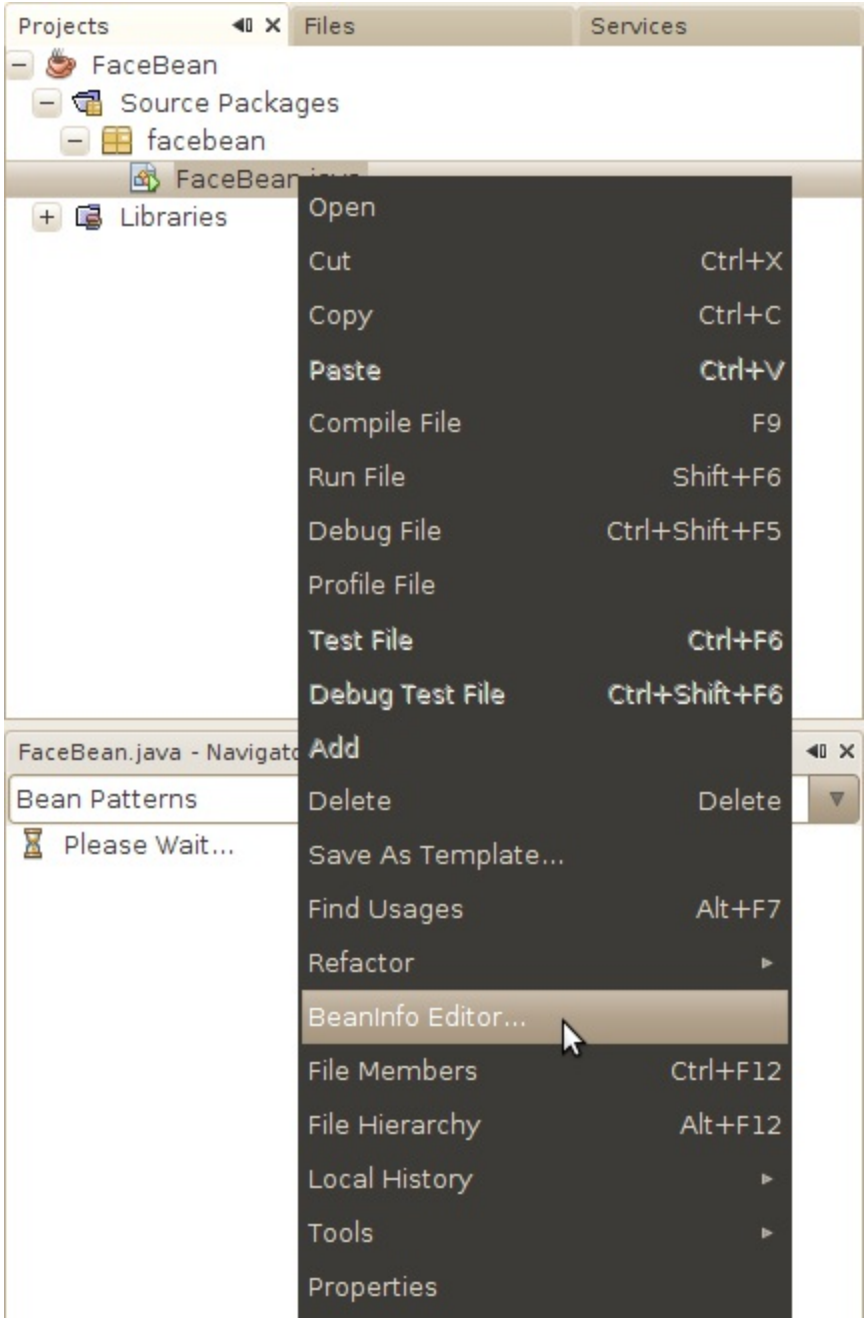
A `BeanInfo` is a class that changes how your bean appears in a builder tool. A builder tool can query the `BeanInfo` to find out which properties it should display first and which should be hidden.

The `BeanInfo` class for your bean should have the same name as the bean class, with `BeanInfo` appended. For example, the `FaceBean` class has a corresponding `FaceBeanBeanInfo` class that describes it.

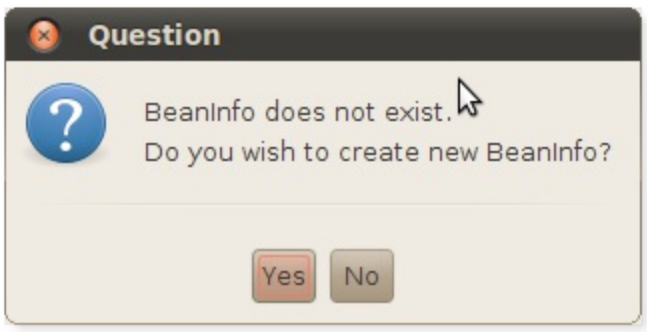
Although it is possible to implement a `BeanInfo` class "by hand," you will find it is much easier to use a tool like NetBeans to edit the `BeanInfo`.

Creating a `BeanInfo` in NetBeans

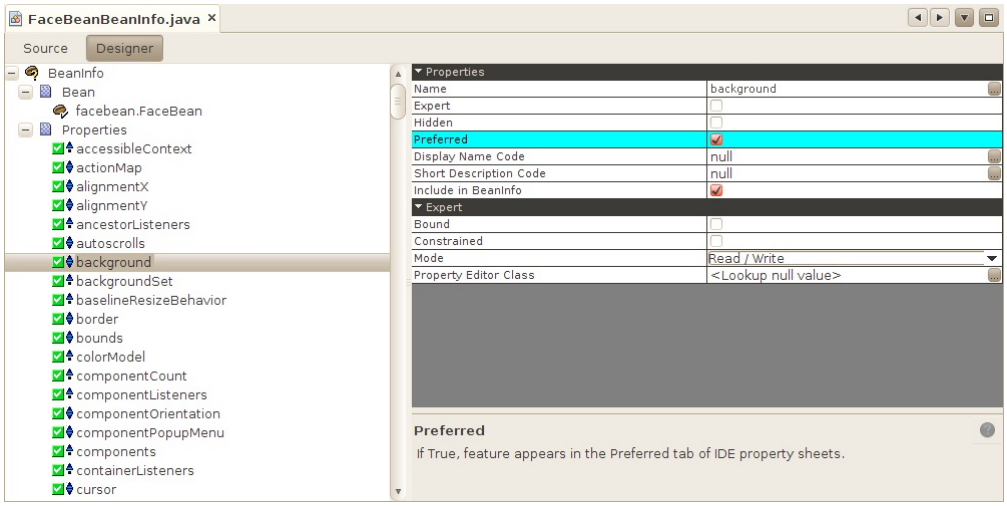
In the **Projects** pane, Control-click on the name of your bean class and choose **BeanInfo Editor...** from the popup menu.



NetBeans notices you don't have a `BeanInfo` and asks if you want to create one. Click **Yes**.



NetBeans creates a new class and drops you into the source code editor. Click on **Designer** to switch to a visual editor.



Click for full image

Select properties from the list in the left side of the visual editor, then edit its attributes in the right side. If you don't want a particular property to appear to a developer using a builder tool, click **Hidden**. To signal that a property should be shown before others, click **Preferred**. You can also indicate if a property is bound or constrained.

You can provide similar information for the bean's event sources and methods.

When a builder tool loads your bean class to add it to a palette, it will automatically find the corresponding `BeanInfo` and use it to decide how to present your bean to the developer.

Lesson: Advanced JavaBeans Topics

Getting started with JavaBeans development is easy, but beans have surprising depth. This lesson covers some more advanced topics, including how beans can be stored (persisted) and how you can supply custom editors for custom data types.

- Bean Persistence describes the mechanisms for saving and reconstituting a bean.
- Long Term Persistence covers representing a bean as XML.
- Bean Customization provides an overview of creating editor components for custom data types.

Note: See online version of topics in this ebook to download complete source code.

Bean Persistence

A bean has the property of persistence when its properties, fields, and state information are saved to and retrieved from storage. Component models provide a mechanism for persistence that enables the state of components to be stored in a non-volatile place for later retrieval.

The mechanism that makes persistence possible is called *serialization*. Object serialization means converting an object into a data stream and writing it to storage. Any applet, application, or tool that uses that bean can then "reconstitute" it by deserialization. The object is then restored to its original state.

For example, a Java application can serialize a Frame window on a Microsoft Windows machine, the serialized file can be sent with e-mail to a Solaris machine, and then a Java application can restore the Frame window to the exact state which existed on the Microsoft Windows machine.

Any applet, application, or tool that uses that bean can then "reconstitute" it by *deserialization*.

All beans must persist. To persist, your beans must support serialization by implementing either the `java.io.Serializable` (in the API reference documentation) interface, or the `java.io.Externalizable` (in the API reference documentation) interface. These interfaces offer you the choices of automatic serialization and customized serialization. If any class in a class's inheritance hierarchy implements `Serializable` or `Externalizable`, then that class is serializable.

Classes That Are Serializable

Any class is serializable as long as that class or a parent class implements the `java.io.Serializable` interface. Examples of serializable classes include `Component`, `String`, `Date`, `Vector`, and `Hashtable`. Thus, any subclass of the `Component` class, including `Applet`, can be serialized. Notable classes not supporting serialization include `Image`, `Thread`, `Socket`, and `InputStream`. Attempting to serialize objects of these types will result in an `NotSerializableException`.

The Java Object Serialization API automatically serializes most fields of a Serializable object to the storage stream. This includes primitive types, arrays,and strings. The API does not serialize or deserialize fields that are marked transient or static.

Controlling Serialization

You can control the level of serialization that your beans undergo. Three ways to control serilization are:

- Automatic serialization, implemented by the `Serializable` interface. The Java serialization software serializes the entire object, except transient and static fields.
- Customized serialization. Selectively exclude fields you do not want serialized by marking with the `transient` (or `static`) modifier.
- Customized file format, implemented by the `Externalizable` interface and its two methods.

Beans are written in a specific file format.

Default Serialization: The Serializable Interface

The `Serializable` interface provides automatic serialization by using the Java Object Serialization tools. `Serializable` declares no methods; it acts as a marker, telling the Object Serialization tools that your bean class is serializable. Marking your class `Serializable` means you are telling the Java Virtual Machine (JVM) that you have made sure your class will work with default serialization. Here are some important points about working with the `Serializable` interface:

- Classes that implement `Serializable` must have an access to a *no-argument constructor* of supertype. This constructor will be called when an object is "reconstituted" from a `.ser` file.
- You don't need to implement `Serializable` in your class if it is already implemented in a superclass.
- All fields except static and transient fields are serialized. Use the `transient` modifier to specify fields you do not want serialized, and to specify classes that are not serializable.

Selective Serialization Using the transient Keyword

To exclude fields from serialization in a `Serializable` object mark the fields with the `transient` modifier.

```
transient int status;
```

Default serialization will not serialize `transient` and `static` fields.

Selective Serialization: writeObject and readObject

If your serializable class contains either of the following two methods (the signatures must be exact), then the default serialization will not take place.

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException;
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

You can control how more complex objects are serialized, by writing your own implementations of the `writeObject` and `readObject` methods. Implement `writeObject` when you need to exercise greater control over what gets serialized when you need to serialize objects that default serialization cannot handle, or when you need to add data to the serialization stream that is not an object data member. Implement `readObject` to reconstruct the data stream you wrote with `writeObject`.

The Externalizable Interface

Use the `Externalizable` interface when you need complete control over your bean's serialization (for example, when writing and reading a specific file format). To use the `Externalizable` interface

you need to implement two methods: `readExternal` and `writeExternal`. Classes that implement `Externalizable` must have a no-argument constructor.

Long Term Persistence

Long-term persistence is a model that enables beans to be saved in XML format.

Information on the XML format and on how to implement long-term persistence for non-beans can be found in XML Schema and Using XMLEncoder.

Encoder and Decoder

The `XMLEncoder` class is assigned to write output files for textual representation of `Serializable` objects. The following code fragment is an example of writing a Java bean and its properties in XML format:

```
XMLEncoder encoder = new XMLEncoder(
           new BufferedOutputStream(
           new FileOutputStream("Beanarchive.xml")));

encoder.writeObject(object);
encoder.close();
```

The `XMLDecoder` class reads an XML document that was created with XMLEncoder:

```
XMLDecoder decoder = new XMLDecoder(
    new BufferedInputStream(
    new FileInputStream("Beanarchive.xml")));

Object object = decoder.readObject();
decoder.close();
```

What's in XML?

An XML bean archive has its own specific syntax, which includes the following tags to represent each bean element:

- an XML preamble to describe a version of XML and type of encoding
- a **`<java>`** tag to embody all object elements of the bean
- an **`<object>`** tag to represent a set of method calls needed to reconstruct an object from its serialized form

  ```
  <object class="javax.swing.JButton" method="new">
      <string>Ok</string>
  </object>
  ```

  or statements

  ```
  <object class="javax.swing.JButton">
      <void method="setText">
          <string>Cancel</string>
              </void>
      </object>
  ```

- tags to define appropriate primitive types:

- **<boolean>**
- **<byte>**
- **<char>**
- **<short>**
- **<int>**
- **<long>**
- **<float>**
- **<double>**

```
<int>5555</int>
```

- a **<class>** tag to represent an instance of Class.

```
<class>java.swing.JFrame</class>
```

- an **<array>** tag to define an array

```
<array class="java.lang.String" length="5">
</array>
```

The following code represents an XML archive that will be generated for the `SimpleBean` component:

```
<?xml version="1.0" encoding="UTF-8" ?>
<java>
  <object class="javax.swing.JFrame">
    <void method="add">
      <object class="java.awt.BorderLayout" field="CENTER"/>
      <object class="SimpleBean"/>
    </void>
    <void property="defaultCloseOperation">
      <object class="javax.swing.WindowConstants" field="DISPOSE_ON_CLOSE"/>
    </void>
    <void method="pack"/>
    <void property="visible">
      <boolean>true</boolean>
    </void>
  </object>
</java>
```

Bean Customization

Customization provides a means for modifying the appearance and behavior of a bean within an application builder so it meets your specific needs. There are several levels of customization available for a bean developer to allow other developers to get maximum benefit from a bean's potential functionality.

The following links are useful for learning about property editors and customizers:

- `PropertyEditor` interface
- `PropertyEditorSupport` class
- `PropertyEditorManager` class
- `Customizer` interface
- `BeanInfo` interface

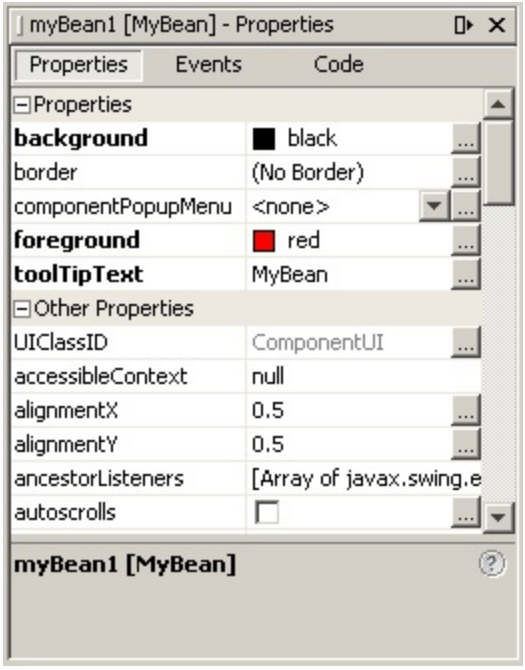
A bean's appearance and behavior can be customized at design time within beans-compliant builder tools. There are two ways to customize a bean:

- By using a *property editor*. Each bean property has its own property editor. The NetBeans GUI Builder usually displays a bean's property editors in the Properties window. The property editor that is associated with a particular property type edits that property type.
- By using *customizers*. Customizers give you complete GUI control over bean customization. Customizers are used where property editors are not practical or applicable. Unlike a property editor, which is associated with a property, a customizer is associated with a bean.

Property Editors

A property editor is a tool for customizing a particular property type. Property editors are activated in the Properties window. This window determines a property's type, searches for a relevant property editor, and displays the property's current value in a relevant way.

Property editors must implement the `PropertyEditor` interface, which provides methods to specify how a property should be displayed in a property sheet. The following figure represents the Properties window containing `myBean1` properties:



You begin the process of editing these properties by clicking the property entry. Clicking most of these entries will bring up separate panels. For example, to set up the `foreground` or `background` use selection boxes with choices of colors, or press the "..." button to work with a standard ColorEditor window. Clicking on the `toolTipText` property opens a StringEditor window.

The support class `PropertyEditorSupport` provides a default implementation of the `PropertyEditor` interface. By subclassing your property editor from `PropertyEditorSupport`, you can simply override the methods you need.

To display the current property value "sample" within the Properties window, you need to override `isPaintable` to return `true`. You then must override `paintValue` to paint the current property value in a rectangle in the property sheet. Here's how `ColorEditor` implements `paintValue`:

```
public void paintValue(java.awt.Graphics gfx, java.awt.Rectangle box) {
    Color oldColor = gfx.getColor();
    gfx.setColor(Color.black);
    gfx.drawRect(box.x, box.y, box.width-3, box.height-3);
    gfx.setColor(color);
    gfx.fillRect(box.x+1, box.y+1, box.width-4, box.height-4);
    gfx.setColor(oldColor);
}
```

To support the custom property editor, override two more methods. Override `supportsCustomEditor` to return true, and then override `getCustomEditor` to return a custom editor instance. `ColorEditor.getCustomEditor` returns `this`.

In addition, the `PropertyEditorSupport` class maintains a `PropertyChangeListener` list, and fires property change event notifications to those listeners when a bound property is changed.

How Property Editors are Associated with Properties

Property editors are discovered and associated with a given property in the following ways:

- Explicit association by way of a `BeanInfo` object. The editor of the title's property is set with the following line of code:

```
pd.setPropertyEditorClass(TitleEditor.class);
```

- Explicit registration by way of the `java.beans.PropertyEditorManager.registerEditor` method. This method takes two arguments: the bean class type, and the editor class to be associated with that type.
- Name search. If a class has no explicitly associated property editor, then the `PropertyEditorManager` searchs for that class's property editor in the following ways:
  - Appending "Editor" to the fully qualified class name. For example, for the `my.package.ComplexNumber` class, the property editor manager would search for the `my.package.ComplexNumberEditor` class.
  - Appending "Editor" to the class name and searching a class path.

Customizers

You have learned that builder tools provide support for you to create your own property editors. What other needs should visual builders meet for complex, industrial-strength beans? Often it is undesirable to have all the properties of a bean revealed on a single (sometimes huge) property sheet. What if one single root choice about the type of the bean rendered half the properties irrelevant? The JavaBeans specification provides for user-defined customizers, through which you can define a higher level of customization for bean properties than is available with property editors.

When you use a bean *Customizer*, you have complete control over how to configure or edit a bean. A Customizer is an application that specifically targets a bean's customization. Sometimes properties are insufficient for representing a bean's configurable attributes. Customizers are used where sophisticated instructions would be needed to change a bean, and where property editors are too primitive to achieve bean customization.

All customizers must:

- Extend `java.awt.Component` or one of its subclasses.
- Implement the `java.beans.Customizer` interface This means implementing methods to register `PropertyChangeListener` objects, and firing property change events at those listeners when a change to the target bean has occurred.
- Implement a default constructor.
- Associate the customizer with its target class via `BeanInfo.getBeanDescriptor`.

JavaBeans(TM): End of Trail

You have reached the end of the "JavaBeans(TM)" trail.

If you have comments or suggestions about this trail, use our feedback page to tell us about it.

 Creating a GUI With JFC/Swing: Here is where you learn how to produce graphical user interfaces. This trail includes information on using the Swing components, which are JavaBeans-compliant.