# Designing beans for visual programming

Tutorial Contents

What a concept
Introduction
Designing the Less bean
Writing the method code

**What a concept...**

Who are you writing beans for? If you're like most bean developers you probably think your customer is someone just like you, a Java programmer. You design your beans with yourself in mind, adding features that would help you as a programmer. Seems reasonable.

But what if you could reach a wider audience? Programmers are only a small slice of the computer-user population. If any computer user could pick up your bean and wire it into an application *without programming*, imagine the increased potential for selling your bean, and for Java programming in general.

**Remember before there were spreadsheets?**

Well most of us can anyway... Before there were spreadsheets if you wanted a computer to perform a calculation on a set of data (say, hike your prices by 10%), you got a programmer to write a COBOL program to process your request. Then someone came up with the idea of a spreadsheet, which hides all the complexity from the user, and suddenly anyone could make the computer do that sort of application. Most people don't consider working with spreadsheets to be programming, yet they are providing all the instructions the computer needs to do what they want it to do, at least for that sort of application.

Of course, spreadsheets don't do everything that users want. In fact, few programs ever exactly match the end users' specifications, because they're designed for a wide audience. So what would happen if we let users design their own apps? If they had the right components and could easily assemble them, they might be able to build these apps themselves.

**Beans...they're not just for programmers any more**

Beans (when used with the right visual builders) have the potential to change both how applications get created, and who creates them. Visual builder tools could tap into a larger customer base, but they need your help. To lure nonprogrammers to the world of Java and visual programming, we need more beans that anyone can pick up and use. And to design those beans, you need to consider a new and different type of user, an application *assembler* who shies away from writing code.

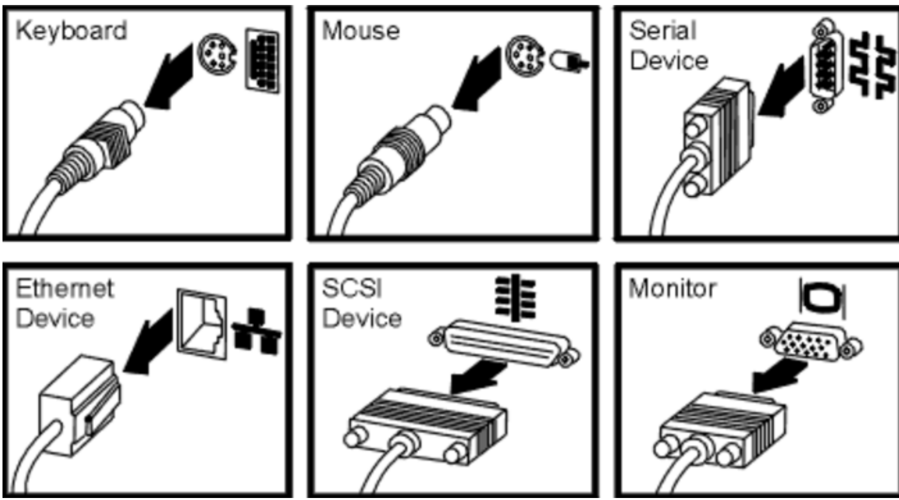
Beans have this great potential because they incorporate two important advances in application development:

*Object-oriented programming* - Objects encapsulate complex functions, making it easier to reuse code. Nonprogrammers will really get interested in your components if they contain complex function but manage to hide that complexity.

*Visual programming* - Connections between objects are displayed and manipulated visually, with lines between components to represent component interactions.

Most people can follow instructions for assembling visual components - wiring keyboard to a computer, connecting up a stereo, or assembling a child's toy. These instructions invariably use diagrams - visual representations of the assembly - rather than textual instructions. So why give computers textual instructions?

Many people can assemble physical components without instructions, because they can see how the parts fit together. Well-designed components only fit together in ways that work (the plug for the keyboard doesn't fit the socket for the printer), or they provide visual clues for assembly (icons on the plug and socket, or matching colors).




**What makes Java special?**

Does all this sound like Visual Basic? While Visual Basic did let you connect your GUI objects to code, its objects fell short of what's needed for Web programming, and they didn't really encapsulate complex functions. Beans have far greater potential: because they're written in Java they work across the Internet and run on any platform with a JVM. But most importantly, the component model for beans was *specifically designed* with visual builders in mind.

Beans also let you add *customizers*, a GUI that helps your bean consumer make your beans do what they want them to do. Think of a customizer as a wizard for a bean. You can provide all sorts of complex options to your users, but in a simple way. Customizers take property editors a step further, letting you change groups of properties with a single option.

**But you have to *code* Java...**

Still resisting this line of reasoning? You're probably imagining why nonprogrammers can't use beans:

*You can only do interesting stuff in methods, not with beans.* Not if your beans are designed right. Take a look at some of the complex, yet easy to use beans on alphaWorks and VisualAge Developer Domain:

- JMF Player
- Gauges
- SmartMarkers
- VisualScheduler
- XML Beans

If your bean successfully encapsulates a particular task that a lot of people want to do, you can provide the methods to do the most common functions. And the programmers can always add their own methods.

*You can't really make beans interact well without code.* Again the magic is in the design (and the visual builders). You have to put some thought into how these beans will need to connect with other beans, and do lots of usability testing. You also might need some helper beans that replace some coding techniques, like a Step bean that lets you specify the order in which events get fired.

*Using the visual interface is too limiting.* It doesn't have to be. Since beans were originally intended to be used in visual builders, the JavaBeans specification gives you a good start on how to expose all your bean's function visually. The JavaBeans Guidelines pick up where the JavaBeans spec leaves off, giving you more tips on how to design your bean. And this tutorial will teach you step-by-step how to implement these guidelines.

*The tools aren't good enough.* Many of the tools around today are designed mainly for programmers. They provide lots of extra function that nonprogrammers don't need. They're improving but we need better visual tools such as debuggers, and ways to visually encapsulate function -- visual subroutines to help manage the clutter on the workspace. But some current visual composition editors are intuitive enough for nonprogrammers to use -- have you checked out VisualAge for Java lately? The companion tutorial to this one, Visual Programming with Beans, shows how easy it is for nonprogrammers to develop programs in VisualAge for Java, without writing code.

**Who are these people anyway?**

The first thing you need is an introduction to your new audience. They could be your nontechnical manager who wants a tool to track your project, but doesn't want to spend money or programming time to get it. Or your Uncle Fred who wants to connect his computer to his barn's thermostat so it wakes him up when it's too cold for his cows. Or even a COBOL programmer who doesn't know Java but wants to put a Web front-end on her legacy application. Anyone who doesn't know how to code Java.

Take a look at Visual Programming with Beans. You'll get a feel for the type of person who might be lured into Java. Then come back here to learn more about coding for them.

**How do I know what they want?**

The thing is, nonprogrammers know what they want their computer to do, but they don't especially know what beans they want, or how they should perform. There's two keys to making good beans, and in fact they will make your beans more appealing to programmers and nonprogrammers alike:

- Encapsulate some complex, really useful function that can be used in many applications. The world has enough UI widgets for now, in all sorts of styles. Beans need to combine lower-level functions so that the bean really hides the complexity from the user. Programmers are good at providing complex function to users, but not so good at hiding the complexity. You must carefully analyze what most users will do with your bean, and cater to them.
- Make sure the bean is easy to use in visual builders. This is easier said than done, especially with the wide range of builders on the market. It includes everything from providing good short names and BeanInfo classes, to more complex issues like serialization and customizers. But that's what this tutorial is for, to teach you how to make your beans easier to use.

The real promise of beans is not "write once, run anywhere", but "write once, reuse everywhere".

**Ready to learn how?**

Continue to the next page to learn how to develop beans for nonprogrammers. We've taken the Less bean used in the Visual Programming with beans tutorial as an example of a well-behaved bean. We'll walk you through how we built this bean in VisualAge for Java. We've picked a simple bean to start with, to make it easier to break down the tasks involved in developing a bean. In future lessons we'll develop more complex beans, and show you how to design them with the visual programmer in mind.

< Previous Table of Contents Next >

---

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Copyright Trademarks

# Introduction to bean development

**Bean design tutorials - Table of Contents**

Tutorial Contents

What a concept
Introduction
Designing the Less bean
Writing the method code

***On this page:***

What is a bean?
Naming your bean
Choosing what features your bean will have

### What is a bean?

If you've read the Visual Programming with Beans tutorial (we hope you have) you've probably noticed that the application assembler views beans as reusable software components that can be wired together to produce a working application. In your job, however, you will not only have to retain that view but also learn to view beans from the technical, low-level point of view which is the one used and explained in this tutorial.

Technically speaking, a bean is a Java class that follows certain guidelines (Sun's JavaBeans specification) and is designed with application assembly use and reuse in mind. If you think that your software will likely be used with other reusable components and manipulated visually in Integrated Development Environments, you probably want to make it a bean.

Although not all beans have to implement all these features, typically a bean provides:

- **Events**: let beans communicate
- **Properties and their Customization**: let developers modify the appearance and behavior of beans
- **Introspection**: lets builder tools analyze bean features
- **Persistence**: lets developers save and retrieve customized beans (all beans have to implement either Serializable or Externalizable interfaces)

There are two major categories of beans: visual and non-visual. Visual beans have to extend java.awt.Component and can be presented visually at run time. Non-visual beans don't face such a restriction and are usually only presented visually at design time. There can be beans that are visual or non-visual depending on the situation, but that is uncommon.

### Naming your bean

Descriptive and appropriate naming is even more important when developing beans than when doing usual Java application development, because you're writing for a much larger audience.

Bean names should be unique (starting with your company's internet domain in inverse order as in com.ibm. or com.lotus. for packages is usually a good idea) and descriptive (avoid inclusion of internal names and information, acronyms or other information that is meaningless or irrelevant to customers).

Good object-oriented design requires a bean to serve one purpose, serve it well and be named clearly to indicate that purpose.

### Choosing what features your bean will have

As you know, only one connection type deals with methods - the Event-to-Method connection. Properties are therefore more flexible than methods and you should sometimes use them even when a method seems more logical (for example, you're often better off implementing void addComponent ( Component ) as a write-only property, not a method).

If you were, for example, developing a Less bean (that performs the logical less comparison operation) you would probably want it to have two read-write arguments representing the two values to be compared, and a read-only result property of type boolean.

So let's develop that Less bean.

---

< Previous Table of Contents Next >

# Designing the Less bean

**Tutorial Contents**
What a concept
Introduction
Designing the Less bean
Writing the method code

***On this page:***
Overview of the Less bean
Creating a Less class

***Time Required:*** 1 hour

***Files Required:*** vptut1cd.zip

### Overview of the Less bean

In this tutorial, we will create the Less bean in VisualAge for Java. You can download vptut1cd.zip, which contains this tutorial and the final source code, and VisualAge for Java Entry Edition 3.0. The Less bean is a non-visual component that represents the less than ("<") comparison operation between numbers. To use this bean the application assembler sets the values of the numbers to compare (the *inputFirstArg* and *inputSecondArg* properties), and then gets a true or false answer in the *result* property. The arguments are type *double* (but VisualAge for Java will attempt to automatically convert other types as well).

### Introspection

We'll also create a LessBeanInfo class that will provide information on properties, events, and methods.

A BeanInfo file is a Java class that implements the BeanInfo interface and has a name that is a concatenation of the name of the bean it describes, and the string "BeanInfo" (for example, for the "Less" bean, the BeanInfo file would be called "LessBeanInfo").

You must provide a BeanInfo file if you don't follow the bean naming rules laid out in Sun's JavaBean Specification. These rules allow bean tools to gather information about your bean, using introspection.

Although it is a good idea to follow the naming conventions, you should provide BeanInfo files anyway (to distinguish between "preferred" and "expert" features and provide property descriptions).

As you will see later in this tutorial, VisualAge for Java will automatically generate BeanInfo files for you if you select that option, however, if you wish to write BeanInfo files manually you can find the necessary information in Sun's JavaBean Specification.

### Customization

An application assembler will be able to change the following properties of the Less Bean: *inputFirstArg* and *inputSecondArg*. The read-only *result* property will not be customizable.

### Property Summary

| Name | Datatype | Constrained \|Bound\|Expert | Description | Default value | Access type |
|---|---|---|---|---|---|
| inputFirstArg | double | unbound | the first argument of the comparison | 0.0 | r/w |
| inputSecondArg | double | unbound | the second argument of the comparison | 0.0 | r/w |
| result | boolean | bound | the result of the comparison | false | r |

### Method Summary

| Instantiation | Less() |
|---|---|
| accessing properties | getInputFirstArg()<br>getInputSecondArg()<br>getResult()<br>setInputFirstArg (double)<br>setInputSecondArg(double) |

| | |
|---|---|
| adding/removing listeners | addPropertyChangeListener(PropertyChangeListener)<br>removePropertyChangeListener(PropertyChangeListener) |
| executing bean function | computeFunction() |

**Event Summary**

PropertyChangeEvent

---

**Creating the Less class**

**Initial preparation**

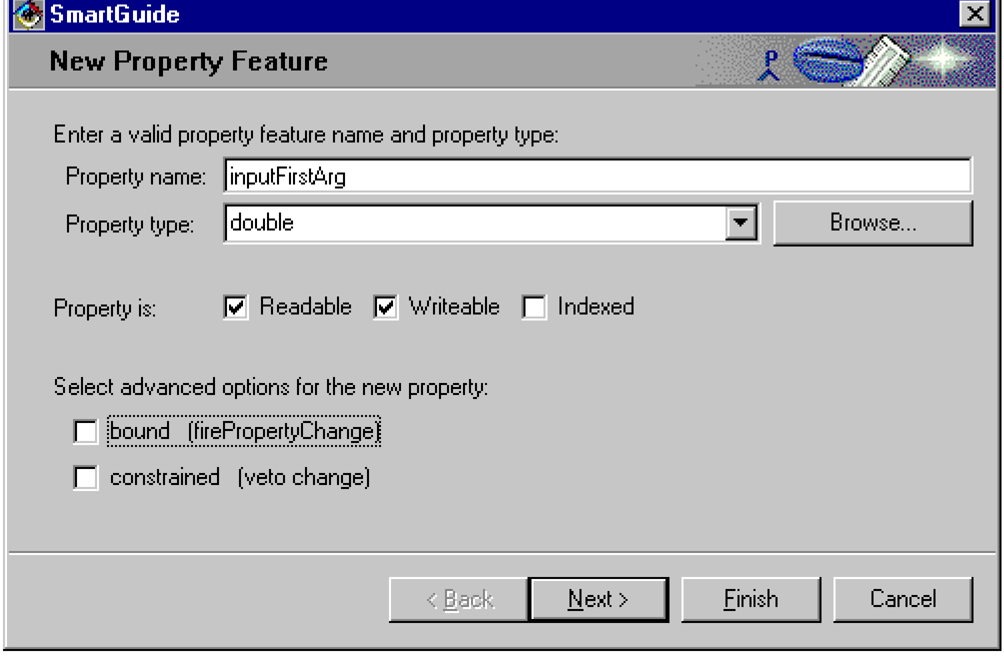
1. Add a project called "LessProject".
2. Add a package called "lesspackage".
3. Add a class called "Less" with the superclass java.lang.Object (leave "Browse the class when finished" and "Compose the class visually" check boxes unchecked). Click **Finish**. You can see the newly created class in the Workbench.

**Adding properties to the bean**

1. In the Workbench, double-click the Less class. The class/interface browser appears.
2. Your bean needs 3 properties: two properties of type *double* to hold the two arguments for comparison and a *boolean* result property. The easiest way to add properties is to define them in the BeanInfo page of the class/interface browser (click the **BeanInfo** tab to get there).

**Creating the inputFirstArg property**

1. Right-click on the Features pane (top left pane) and select **New Property Feature** from the pop-up menu.
2. Enter the "inputFirstArg" in the Property name entry field.
3. Select "double" as Property type.
4. Leave check boxes **Readable** and **Writeable** checked.
5. Uncheck the **bound** check box.
   **Bound vs. Unbound Properties:** You should carefully evaluate whether to make a property bound or unbound. Unbound properties are more efficient than bound ones, so you should only make a property bound when another bean might need to know when this property changes. In general input variables don't need to be bound, because the value of this property doesn't change by itself (or the end user), it only gets changed by the underlying application.



6. Click **Next**. In the next page you write the description and display name for this property.

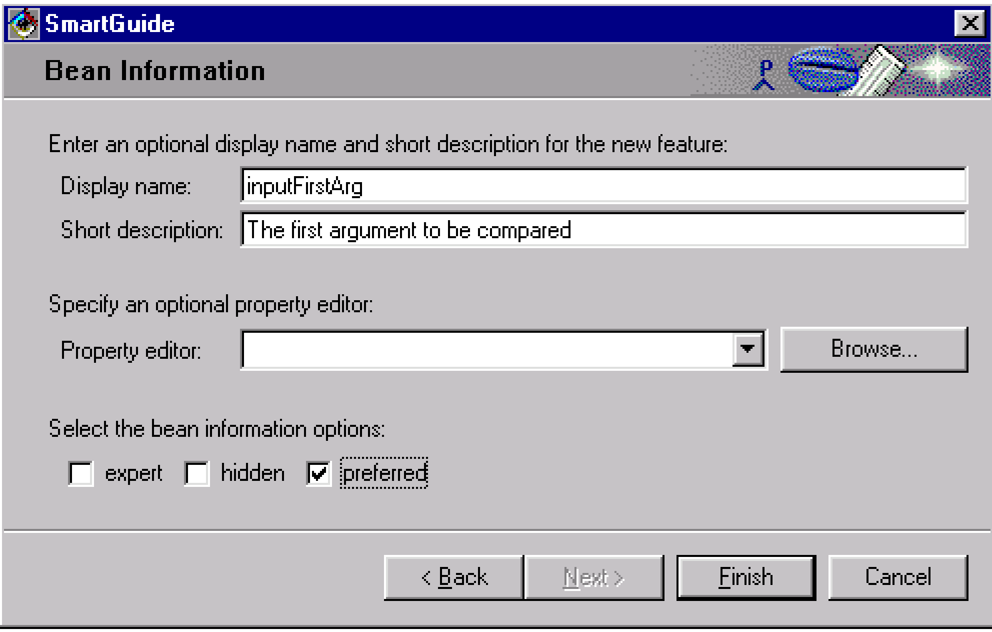
**Writing descriptions for properties**

The VCE uses the display name in the connections pop-up (or the property name if the display name isn't specified). It shows the descriptions in the Property List or while making connections. So you can see it's important to provide intuitive and descriptive names for both of these fields.

1. In the second Bean Information page of the SmartGuide write "inputFirstArg" as a display name.
2. Enter "The first argument to be compared" as the short description.
3. Make the property preferred. To do this check *preferred* check box in the SmartGuide window.

**Making properties preferred features**

As you will see, all properties created on this page (*result*, *inputFirstArg*, *inputSecondArg*) are necessary for wiring. Therefore it is better to make them preferred - this will show them in the property list while connecting beans instead of showing them only in the Connectable features window.

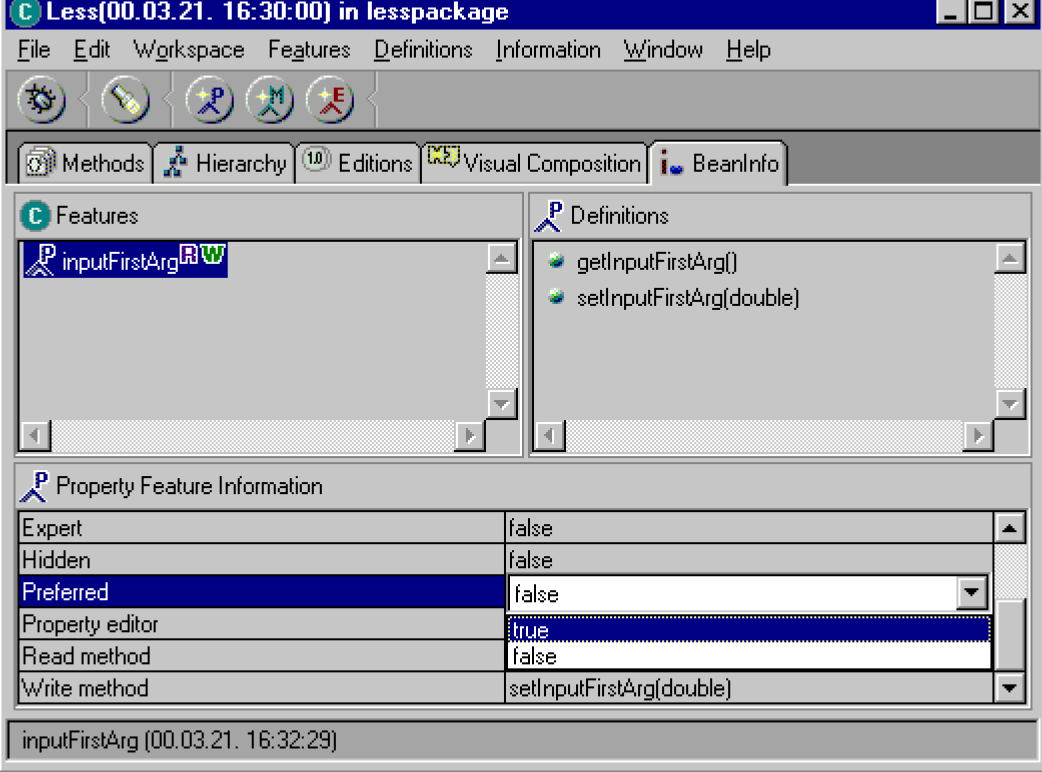



4. Click **Finish**.

***VisualAge for Java 2.0 users: Making properties preferred features***

If you are VisualAge for Java 2.0 user, you won't see the "preferred" check box in the SmartGuide window. To make property preferred:

1. Select the *inputFirstArg* property from the Features pane.
2. In the Property Feature Information pane at the bottom select the Preferred item from the list.
3. Choose *true* as the value, as shown in this image:



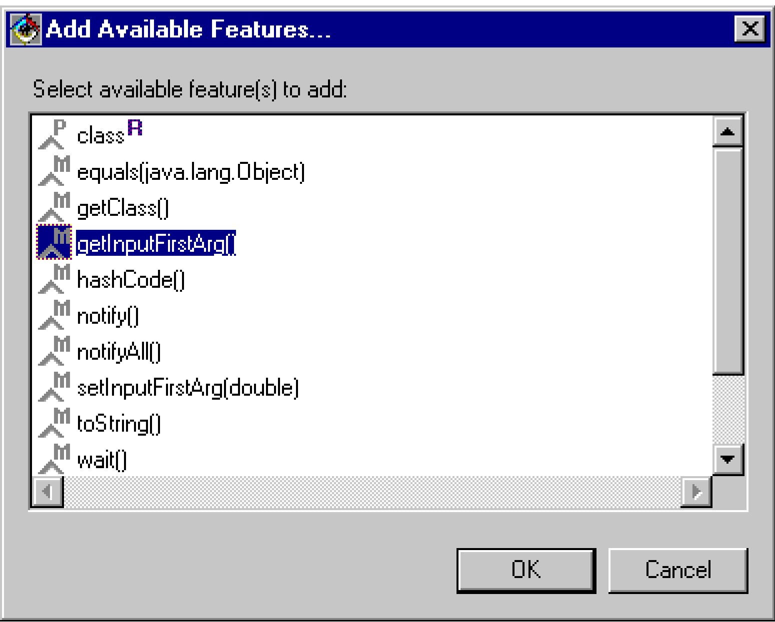

4. Answer **Yes** in the Property Feature Information dialog that appears.

You will need to repeat this for the *inputSecondArg* and *result* properties.

**Adding features to the BeanInfo**

The *inputFirstArg* property appears in the Features page. Two methods - *getInputFirstArg()* and *setInputFirstArg(double)* are shown in the Definitions page. To make these methods available in the VCE you must add them to the Features page; otherwise these methods would not be available for wiring.

1. Right-click on the Features pane (top left pane) and select **Add Available Features...** from the pop-up menu.
2. The list of available features appears.
3. Select the *getInputFirstArg()*.




4. Click **OK**.
5. In the same way add the *setInputFirstArg(double)* method to BeanInfo.
6. The two methods are added to the Features pane.

**Adding more features to the bean**

1. Create the *inputSecondArg* property in the same way (set the description for it as "The second argument to be compared").
2. Make this property preferred.
3. Create the *result* property (set the description for it as "The result of the comparison"). Select the "boolean" as the type for this property.
4. Make it bound - this helps the usability of this bean by allowing other beans to perform some functionality every time the value of this property changes.
5. Leave **Readable** and **Writeable** checked. Although we will want the *result* property to be read-only in the end, we're leaving **Writeable** checked so that VisualAge for Java generates a *setResult(boolean)* method that fires *PropertyChangeEvent* (for bound properties). Make the *result* property preferred as well.
6. Two more properties (the *inputSecondArg* and *result*) have been added to the Features pane. Add *getInputSecondArg(), setInputSecondArg(double)* and *getResult()* methods to the Features pane using **Add Available Features...** from the pop-up menu - this will allow access to your properties from builder tools also using method connections.

**Creating the method for the bean**

Now you must create the method that compares two arguments and set the result of the comparison to the *result* property. Let's define the computeFunction() method.

1. Right-click on the Features pane and select **New Method Feature** from the pop-up menu.
2. Type "computeFunction" as the Method name, leave the Return type as *void* and parameter count as 0.
3. Click **Next**. You can enter the description and display name of this method in the same way as you did for properties. In the Bean Information second page of the Smart Guide write the "computeFunction" as display name and "Check if the first argument is less than second" as short description.
4. Make *computeFunction()* method a preferred feature.
5. Click **Finish**.

Now you've created a stub for your method; next we'll check out how the bean looks in VisualAge for Java, and then write the method code.

---

< Previous Table of Contents Next >

# Writing the method code

***On this page:***

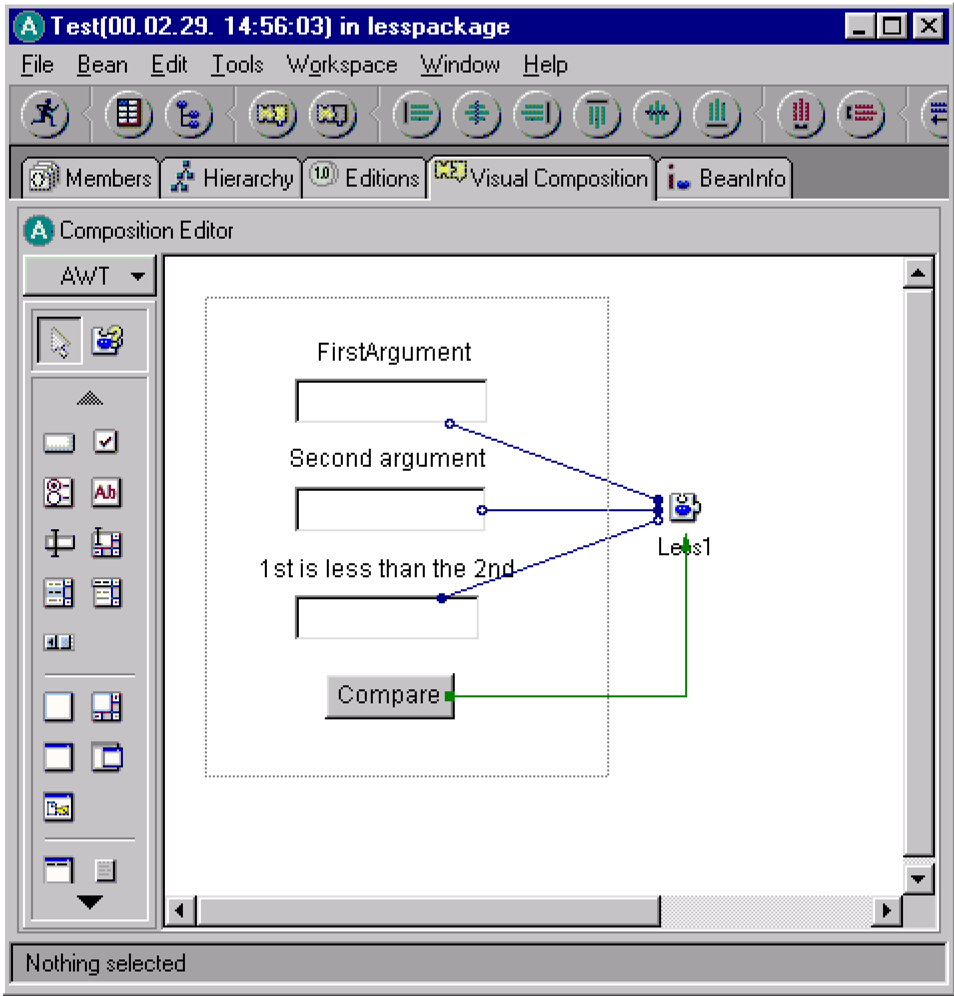

[Making a simple application with the Less bean](#)
[Initial preparation and placement of the Less bean](#)
[Placing necessary beans](#)
[Making connections](#)
[Implementing the computeFunction() method](#)
[Making the result property read-only](#)
[Documenting your bean](#)
[Conclusion](#)



**Tutorial Contents**
What a concept
Introduction
Designing the Less bean
Writing the method code


### Making a simple application with the Less bean

To see how our Less bean works let's make the following application:




This application consists of 3 TextFields - two for arguments to be compared and one for the result of the comparison. When you click the button, it invokes the *computeFunction()* method of the Less bean. Every time the result of comparison is changed, the value of the *result* property of the Less bean is assigned to the third TextField.

### Initial preparation and placement of the Less bean

1. Create an applet called "Test" in the lesspackage package.
2. Open this applet in the VCE
3. From the palette select the icon. In the Choose Bean window add the Less bean from the lesspackage to the applet with the help of the **Browse** button or by typing in the full name of the bean - "lesspackage.Less".

### Placing necessary beans

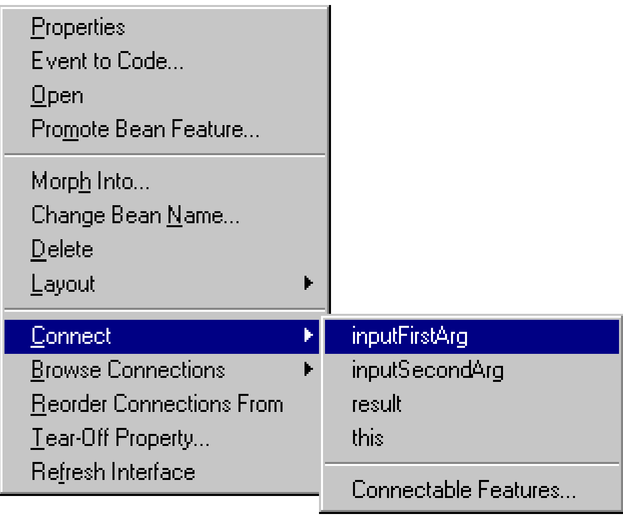
1. Put 3 TextFields from the AWT palette on the design surface.

2. Put 3 AWT labels and change the *label* property for them as shown in the previous image.
3. Put one AWT button, and change its *label* property to "Compare".
4. Arrange and resize these beans as shown in the image above.

**Making connections**

1. Connect TextField1.text to Less.inputFirstArg (set the textValueChanged event as the source event).
   The textValueChanged event gets fired when you input something into a TextField. To set the source event for the connection, double-click on the connection (this will bring up the Properties window) and choose the appropriate event from the list. Our newly created connection provides the Less bean with its first argument.
2. Connect TextField2.text to Less.inputSecondArg (set the textValueChanged event as the source event).
3. Connect Button1.actionPerformed to Less.computeFunction().
4. Connect Less.result to TextField3.text.

As you might have noticed making connections, only preferred properties appear in the pop-up list:

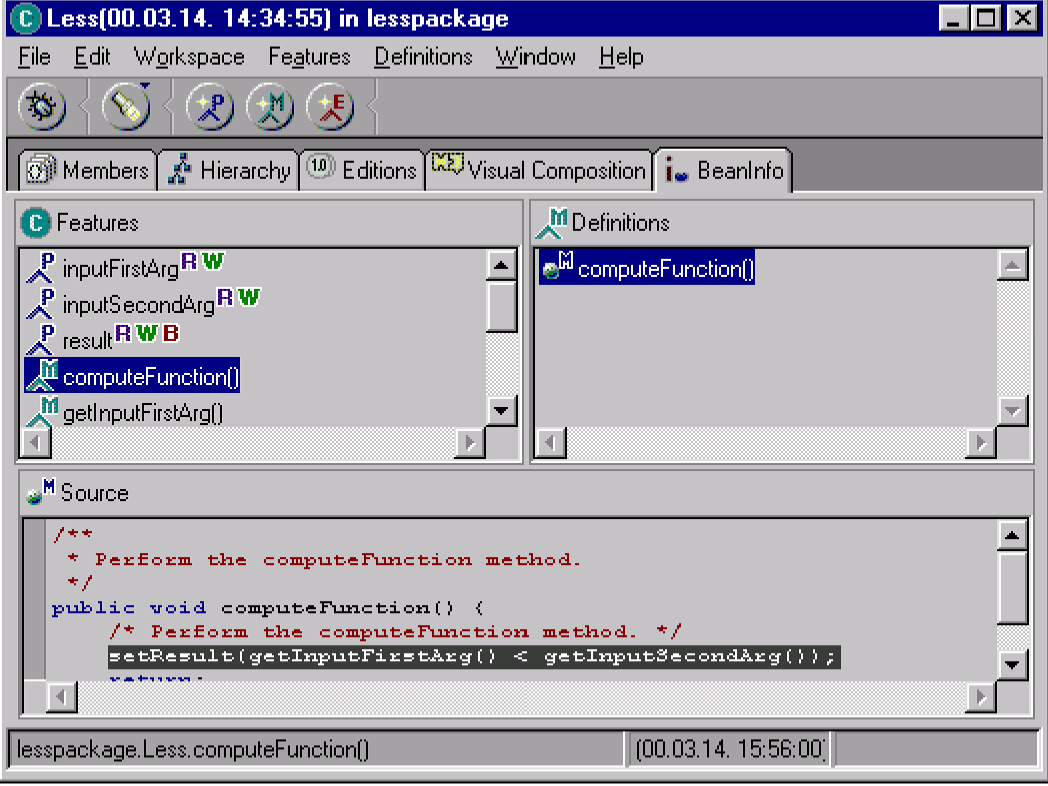



Run the applet. Type numbers in the TextFields, and click the Compare button. The third text field is not being set because we haven't written the method *computeFunction()* yet.

**Implementing the *computeFunction()* method**

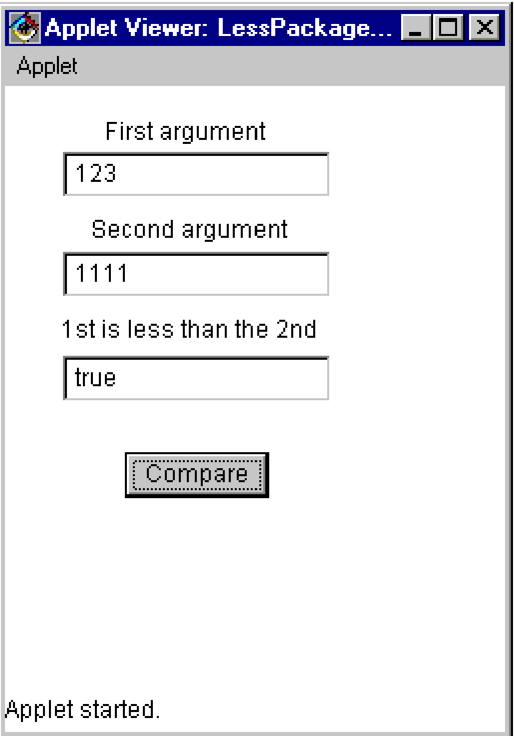
1. To write the code for *computeFunction()* method, open Less bean in **BeanInfo** tab pane. Select *computeFunction()* in Features top left pane, then select *computeFunction()* in the Definitions top right pane.
2. At this moment the *computeFunction()* method in the bottom Source pane consists of the following:
   - First comment
   - The method definition and an opening brace
   - Another comment
   - The return statement and a closing brace
3. Add the following code after the second comment:

```
setResult(getInputFirstArg()<getInputSecondArg());
```




This code sets the result of the comparison between the two arguments to the *result* property. Save your changes.

4. Now you can check the test applet again. It works just as expected now!



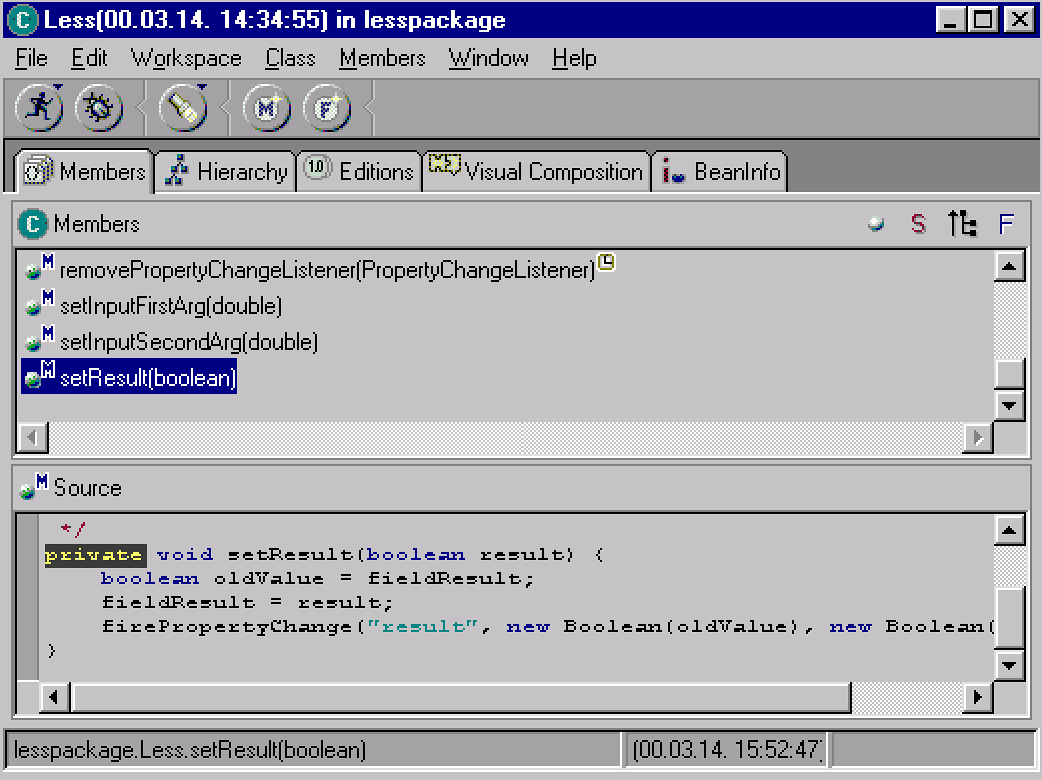

**Making the *result* property read-only**

As you can see now, the Less bean has 3 properties shown in the Property List at design time. We want to make the *result* property read-only because there is no need to leave it writeable - the value of this property changes according to the result of comparison and changing it in any other way does not make any sense (and the ability to do so would therefore be bad design). For example, a property-to-property connection in which the *result* property takes part will change the *result* every time the other property changes (if it is bound).

To make the property read-only, simply change the public modifier of the method *setResult(boolean)* to private (as

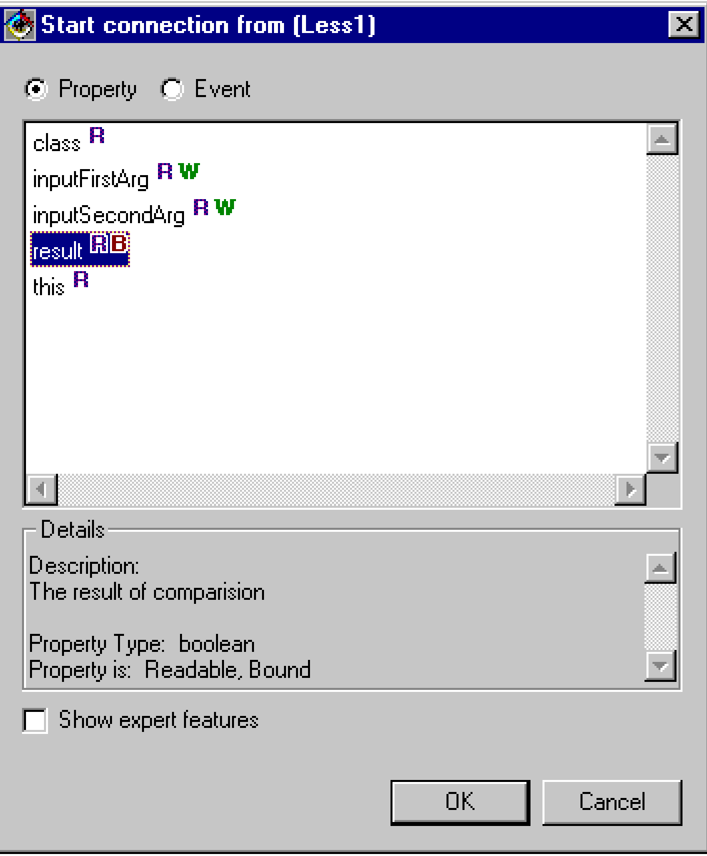
you remember, only public methods can be used to export properties):

1. Double-click the bean. The class/interface browser appears.
2. Choose the Members pane.
3. Select the *setResult(boolean)* method in this pane.
4. Change the string "public" in the class definition in the Source pane to "private" and save changes:




Choose the BeanInfo pane and select Generate BeanInfo class from the pop-up menu. This will apply our changes to the BeanInfo.

Check what the Property List looks like now - the *result* property no longer appears there. Right-click on the bean, select **Connect...** from the pop-up menu and choose the **Connectable Features**. As you can see, the *result* property is shown as a read-only property:




If you want to compare your wiring with ours, import vptut1cd.dat with the final Less bean and Test application. We've provided two versions of vptut1cd.dat. One is located in the vaj3_repository directory for Visual Age for Java 3.0 users, while the other one is located in the vaj2_repository directory for Visual Age for Java 2.0 users. You can choose the .dat file that's appropriate for your edition. Although VisualAge for Java editions are upwardly compatible, and DAT files made in edition 2.0 work fine in edition 3.0, the wiring composition sometimes become unreadable. Besides the generated code may be slightly different between both VisualAge for Java versions. So we've given you two .dat file versions.

**Documenting your bean**

When developing beans good documentation is very important. You should usually start developing the documentation as soon as the requirements and design for your bean are clear, this will help you notice the problem points earlier (if you cannot describe it well chances are that your customers will have a hard time understanding it).

Feature descriptions in BeanInfo files help your customers find out what each feature does. JavaDoc comments in code help your customers extend your beans and will help you (or other people from your company) maintain your bean as it evolves.

Supplying help files lets you describe your beans in more detail as well as let you provide tutorials to using your bean suites. As beans are cross-platform, HTML is the recommended cross-platform format to use at the moment.

As you may have noticed, most of the beans included in our Visual Programming Tutorial for application assemblers have a special property called "aboutThisBean". This property presents help information that application assemblers can browse directly from the visual composition editor to see how to wire a bean into their applications. In a future installment, we'll show how you can add help information to your beans using the AboutBeanEditor.

**Conclusion**

We hope that in this tutorial we've shown you that developing components is as easy as developing applications, and intuitively leads to better, more modular code. If you follow these design principles you really can reach a broader audience of visual programmers. In further installments we'll try to teach you more component development design patterns and answer any questions you might have.

**Express yourself:** **Check out these sites:**

We want your ideas on what you'd like to see in future installments. Need some help with a particular bean or got a question on this tutorial? Let us hear from you at:

stephp@us.ibm.com

IBM's Java site
Javasoft
alphaWorks
SanFrancisco

< Previous Table of Contents Next >

# User-generated events

**Bean design tutorials - Table of Contents**

**Tutorial Contents**
User-generated events
Creating a visual bean
Adding a MouseListener
Testing your visual bean
Final Applet

***This page covers:***
Introduction to events
Adding an event listener interface to the Less bean
Testing the improved Less bean

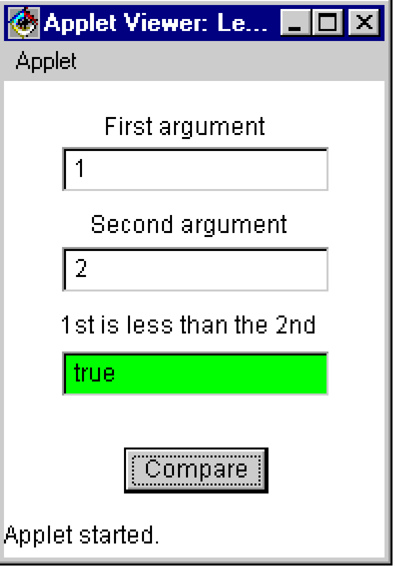
***Prerequisite Lesson:*** Creating beans in VisualAge for Java

***Time Required:*** 3 hours

***Files Required:*** The vptut2cd.zip file provides this tutorial and the source code of the final application that you will build in one zip file. To unzip the file, you need a zip utility, like WinZip or PKZIP.

**Introduction to events**

We want now to improve our Test application and make it more interactive. Why not change the color of the result TextField, to reflect whether the result is true or false? We will need to add a new feature to our Less bean, to inform other beans about its *conditionChange* event.



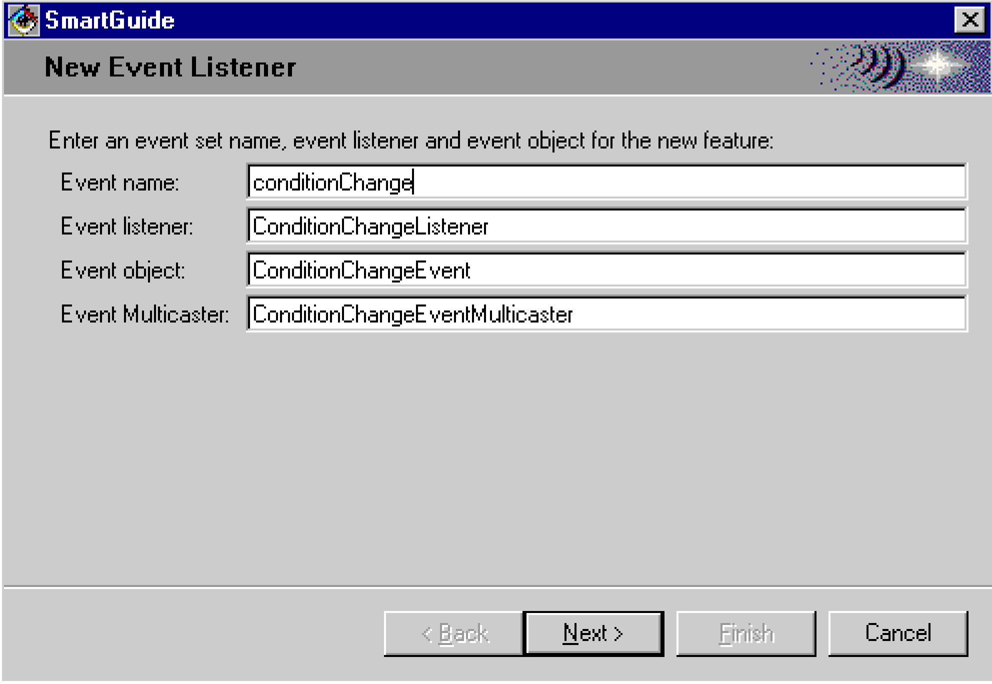

Events generally indicate that a component in a program has reached a certain state or that an externally generated action has taken place. The Java event model defines *event sources* and *event listeners*. Listeners are responsible for registering their interest in an event with the source of the event. When the event occurs, the source informs all listeners, which can then act on the event according to their own needs.

**Adding an events listener interface to the Less bean**

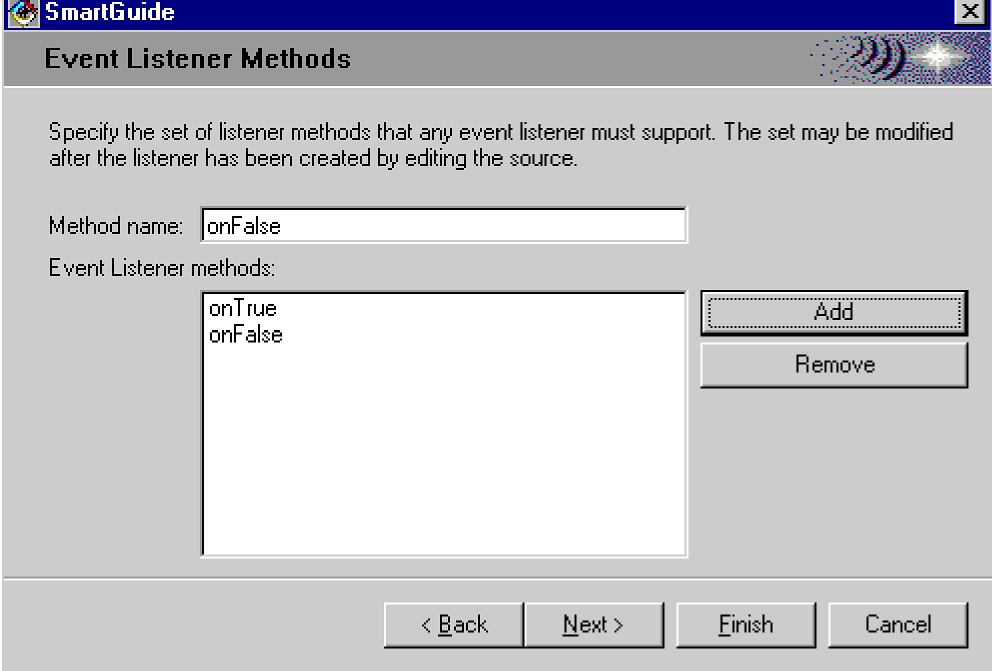
1. In the Workbench, double-click your Less class. The class/interface browser appears.
2. Your bean needs two event listener methods, one for True results and one for False results. The easiest way to add the listener interface is to define it in the BeanInfo page of the class/interface browser (click the **BeanInfo** tab to get there).

**Creating the ConditionChange Listener**

1. Right-click on the Features pane (top left pane) and select **New Listener Interface** from the pop-up menu.
2. Enter the "conditionChange" in the Event name entry field.




3. Click the **Next** button
4. Add two event listener methods: *onTrue* and *onFalse*. Every class that listens for these events will implement these methods.




5. Click the **Next** button
6. In the Bean Information page of the SmartGuide write "conditionChange" as a display name. Enter "The condition change event" as the Short description.
7. Click the **Finish** button

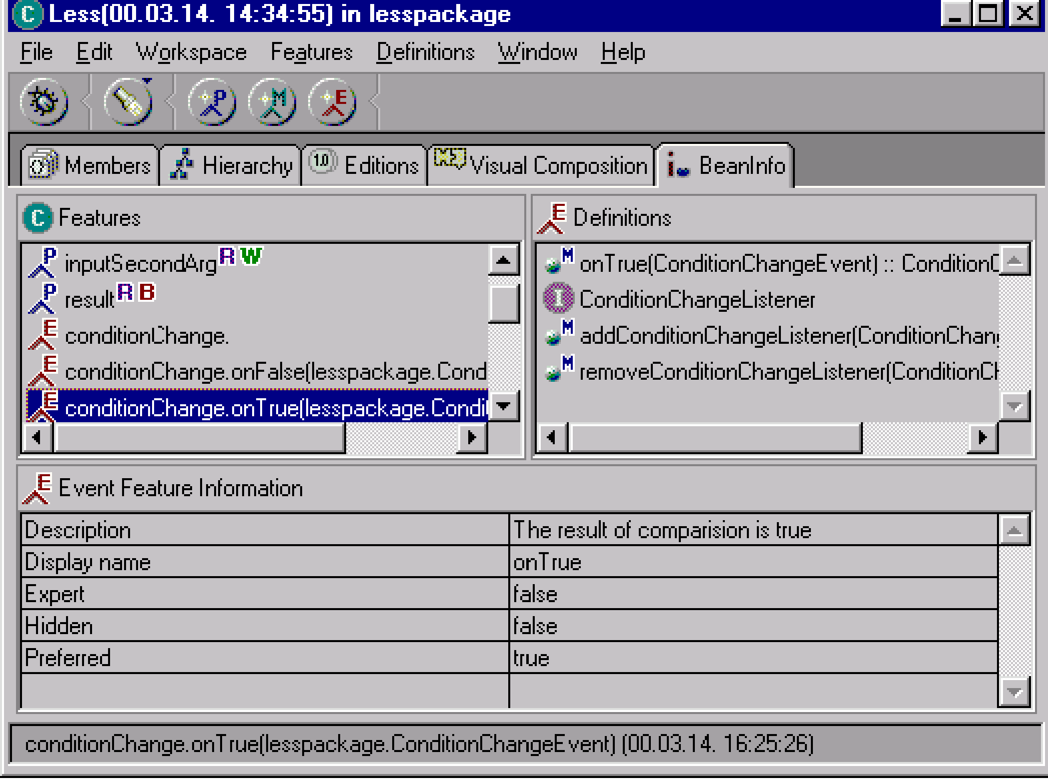
### Marking events preferred

1. In the Feature pane select the *conditionChange.onTrue(lesspackage.ConditionChangedEvent)* event and edit its Event Feature Information as follows (save the event when prompted):

| **Description:** | The result of comparison is True |
|---|---|
| **Display name:** | onTrue |
| **Preferred:** | true |

- Select the *conditionChange.onFalse(lesspackage.ConditionChangedEvent)* event and edit its Event Feature:

| **Description:** | The result of comparison is False |
|---|---|
| **Display name:** | onFalse |
| **Preferred:** | true |




Switch to the Members page and note the methods created:

- *addConditionChangeListener(ConditionChangeListener)*
- *removeConditionChangeListener(ConditionChangeListener)*
- *fireOnFalse(ConditionChangeEvent)*
- *fireOnTrue(ConditionChangeEvent)*

The first two methods are used by the listeners to add and remove themselves to the list of interested objects. The last methods are used in this bean to indicate whether the result is true or false.

**Adding the code**

Switch to the *setResult(boolean)* method and add the highlighted code (you can copy it from your browser and paste it into VisualAge).

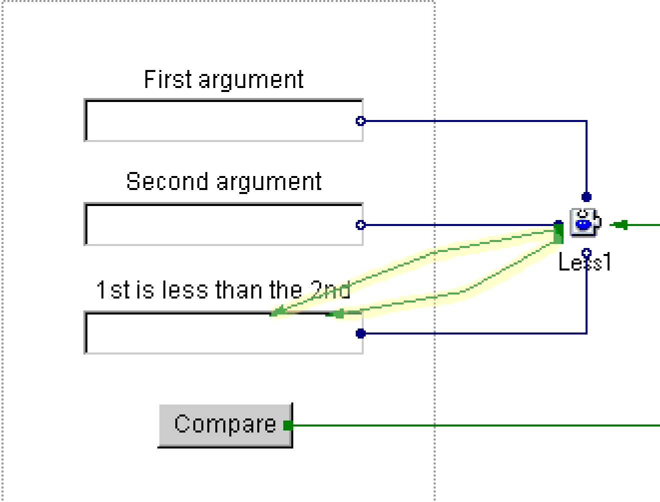
```
private void setResult(boolean result) {
      boolean oldValue = fieldResult;
      fieldResult = result;
      firePropertyChange("result", new Boolean(oldValue),
         new Boolean(result));
      if (result)
         fireOnTrue(new ConditionChangeEvent (this));
      else
        fireOnFalse(new ConditionChangeEvent (this));
      }
```

Save your changes.

**Testing the improved Less bean**

To see how the Event Listeners work let's make the following connections in our Test application:

1. Connect the Less1.onTrue to TextField3.background.
   The connection is dashed because you need to define what color you want to set the background to. Double-click the connection, press the **Set parameters** button, and set the value to green.
2. Connect the Less1.onFalse to TextField3.background. Set the parameter for this connection to red.




Save and test the program.

---

<Previous Table of Contents Next >

# Creating a visual bean

**Tutorial Contents**
User-generated events
Creating a visual bean
Adding a MouseListener
Testing your visual bean
Final Applet

**Bean design tutorials - Table of Contents**

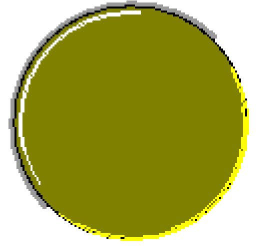
***On this page:***

What is Semaphore?
Creating the Semaphore class
Adding properties and methods to the semaphore class
Making the paint() method of the Semaphore
Changing the initialization color of the Semaphore

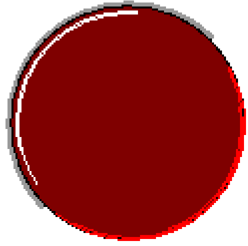
Now that you can develop your own functional beans let's create something visual.  In this section we will use many new techniques, both in visual and conventional programming.

### What is a semaphore?

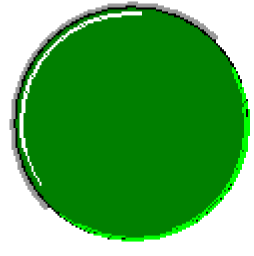
The image below shows three running Semaphores. In this section you will build the Semaphore bean to indicate the state of an object. A semaphore can change its color to signal visually that the property of an object changed. In our program we'll use yellow for the initial state of the semaphore, red for false, and green for true:



*Initial color at start up*



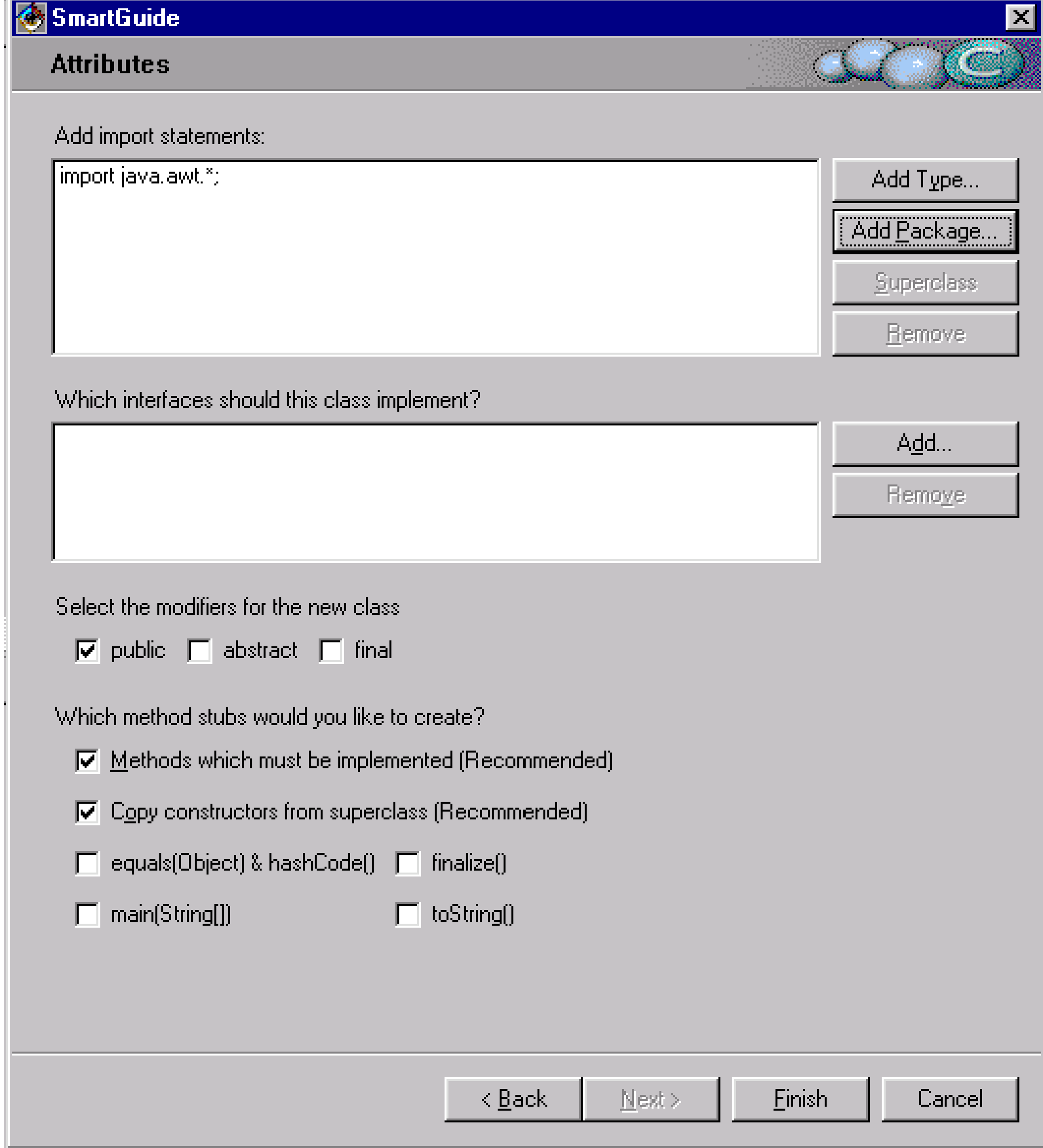
*False*



*True*

### Creating the Semaphore class

1. In the lesspackage, create a new class called Semaphore, subclassing java.awt.Component.
2. Click **Next>** to add the necessary import statements.
3. In the second SmartGuide window, click the **Add Package...** button.
4. Enter *java.awt* in the *Choose a package* field, and click **Add** and then **Close**.




5. Click **Finish**.
6. After the class is created, double-click on it.
7. Switch to the Hierarchy window.
8. The code of the Semaphore class should look like this:

```
import java.awt.*; /**
* Insert the type's description here.
* Creation date: (00.02.29. 17:08:21)
* @author:
*/

public class Semaphore extends Component {
}
```

**Adding properties and methods to the Semaphore class**

We need one property and two methods of our visual component to be available to the user: the *color* property and its *setColor()* and *getColor()* methods.

1. Switch to the BeanInfo window and add the *color* property to your class.
2. Select "*java.awt.Color*" as Property type. Leave check boxes *Readable* and *writeable* checked. Uncheck the *bound* check box.
3. Click **Next**. Set the display name to color and the description for it to "The color of the Semaphore".
4. Make the color property preferred - this will show it in the property list while connecting beans instead of showing it only in the Connectable features window.
5. You don't need to modify the getColor() method; it should look like this:

```
public java.awt.Color getColor()
{
return fieldColor;
}
```

6. Change the setColor() method to:

```
public void setColor(Color color) {
fieldColor = color;
repaint();
}
```

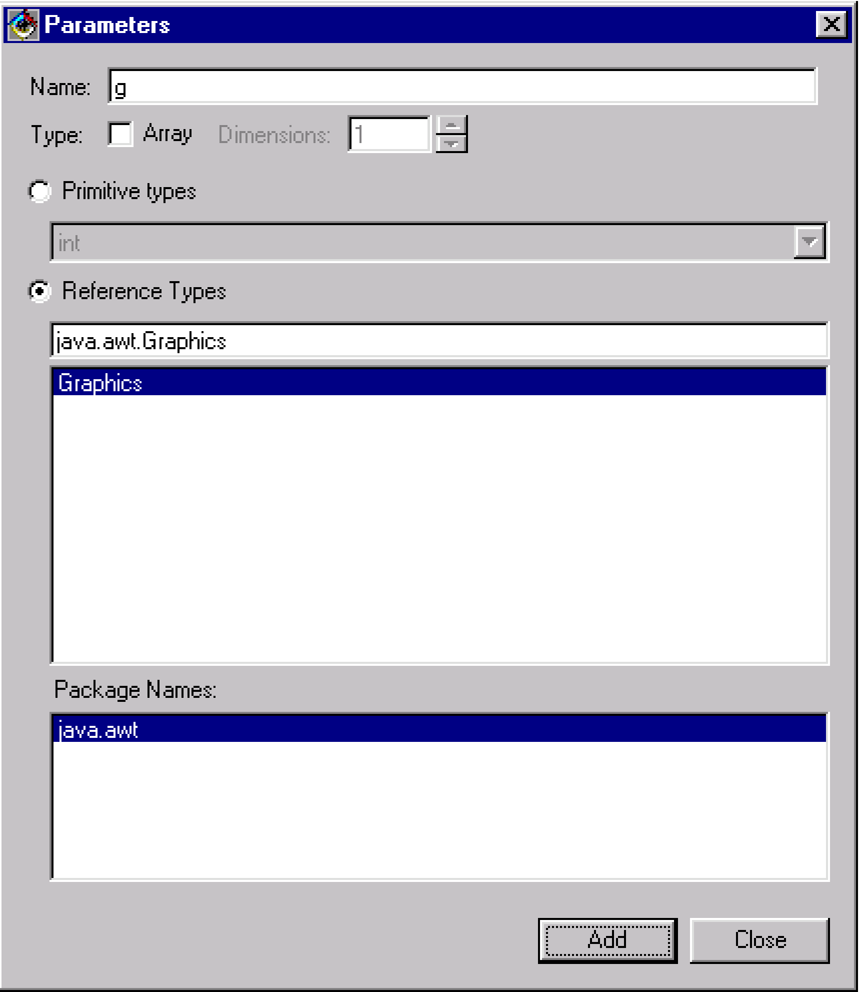
   We call the repaint() method each time the color of the Semaphore changes. The repaint() method refreshes the applet contents in a browser window. This allows us to use the semaphore as an indicator of different events.

7. Make both methods available (**Add Available Features**)

**Making the paint() method of the Semaphore**

Switch to the Members tab and add a new *public void* method called *paint (java.awt.Graphics g)*. To do this:

1. Right-click in the Members pane.
2. Choose **Add** --> **Method...** from the pop-up menu.
3. Enter *public void paint()* in the method name field. ***VisualAge for Java2.0 users:*** to create this method type *void paint(java.awt.Graphics g)* for the method name field and click Finish.
4. Click **Next>** to add a parameter for this method.
5. In the second SmartGuide window click **Add** button.
6. Set parameter name to "g";
7. Click the "Reference Types" radio button.
8. Set the parameter type to "java.awt.Graphics" and click **Add** and then **Close** buttons.



After pressing the **Finish** button you will have the following string:

```
public void paint(java.awt.Graphics g) {}
```

1. Select the paint method and input the following code:

```
Dimension d = getSize();
//This gets the current
//size of the Component

int x = d.width;
int y = d.height;

Color dark = fieldColor.darker().
 darker();
Color bright = fieldColor.
 brighter().brighter();

int sgr = x < y ? x : y;

int sx = (x-sgr)/2;
int sy = (y-sgr)/2;

g.setColor (Color.white);
g.fillArc (sx, sy, sgr, sgr,
   45,-180);

g.setColor (new Color
   (128,128,128));
g.fillArc (sx, sy, sgr, sgr,
   45,180);

g.setColor (Color.black);
g.fillOval (sx+2, sy+2, sgr-4,
   sgr-4);

g.setColor (dark);
g.fillOval (sx+3, sy+3, sgr-6,
   sgr-6);

g.setColor (Color.white);
g.drawArc (sx+5, sy+5, sgr-12,
   sgr-12, 85,110);
g.drawArc (sx+6, sy+6, sgr-14, sgr-14,
   85,130);

g.setColor (bright);
g.drawArc (sx+4, sy+4, sgr-8, sgr-8, 30,-160);
g.drawArc (sx+5, sy+5, sgr-10, sgr-10, 30,-150);
g.drawArc (sx+6, sy+6, sgr-12, sgr-12, 10,-80);
```

***The Graphics class***

This class already provides methods for drawing shapes. The ones used in this example were:

**fillOval(x_pos, y_pos, width, height)** draws a filled oval. The values inside the parentheses are the parameters this method takes. They are the x and y coordinates of the top left corner of the rectangle containing oval, and the width and height of this rectangle. The oval drawn will be filled in with a color specified by the setColor method.

**drawArc (x_pos, y_pos, width, height, start_angle, end_angle)** which draws a filled arc. The values inside the parentheses are exactly as they are in the fillOval method. The difference is that the arc drawn will not be filled in with a color. It has two more parameters - starting and ending angles of the arc.

**fillArc (x_pos, y_pos, width, height, start_angle, end_angle** does the same thing as drawArc, but like fillOval, fills the arc with a color specified.

The **setColor(color)** method of the Graphics class sets the drawing color for the next shape to the specified color.

Now it's time to test your semaphore. Save your changes, switch to the Visual Composition Editor, click the Run button, and try to change the size of your Test Frame. As you see, the size of the semaphore also changes.

**Changing the initialization color of the Semaphore**

Now, switch again to the Hierarchy window, select Semaphore from the Hierarchy pane and change the code as shown here:

```
import java.awt.*;
/**
 * Insert the type's description here.
 * Creation date: (3/6/00 7:25:15 PM)
 * @author:
 */
```

***The Color Class***

Another class used in this example is the *Color class*. The *Color class* encapsulates color information and provides static fields with predefined colors, such as **Color.white**.

```
public class Semaphore extends java.awt.Component {
        private java.awt.Color fieldColor = Color.yellow;
}
```

This code sets the initialization color of the Semaphore to yellow.

Test your Semaphore once again. The last time you executed the Semaphore class, VisualAge for Java automatically generated the *main()* method. Now you can execute the Semaphore from the Hierarchy or Members pane by clicking the Run button.

---

<Previous Table of Contents Next >

## Adding a MouseListener

Tutorial Contents

User-generated events
Creating a visual bean
Adding a MouseListener
Testing your visual bean
Final Applet


***On this page:***
We will add some additional lines of code to demonstrate the use of events in your own visual bean. To keep it simple we will implement the *MousePressed* event. When the user press any mouse button on the bean, it beeps.

1. Switch to the Hierarchy window and edit the Semaphore class as follows:

```
import java.awt.*;
import java.awt.event.*;

public class Semaphore extends java.awt.Component
 implements MouseListener {
private Color fieldColor = Color.yellow;
}
```

2. Since we have implemented the MouseListener interface in our class – we have to define all the methods of this interface. To do this change the Semaphore class as follows:

```
import java.awt.*;
import java.awt.event.*;

public class Semaphore extends java.awt.Component
implements MouseListener {
   private Color fieldColor = Color.yellow;
   public void mouseClicked(MouseEvent e){}
   public void mouseEntered(MouseEvent e){}
   public void mouseExited(MouseEvent e){}
   public void mousePressed(MouseEvent e){}
   public void mouseReleased(MouseEvent e){}
}
```

When the VisualAge for Java generates needed methods, you can access them from the Members panel in the Hierarchy pane.

1. Edit the *mousePressed* method:

```
public void mousePressed(MouseEvent e){
  Toolkit.getDefaultToolkit().beep();
 }
```

2. Add the following line to the *initialize()* method of the Semaphore:

```
addMouseListener(this);
```

Source

```
private void initialize() {
    try {
        // user code begin {1}
        addMouseListener(this);
        // user code end
        setName("Semaphore");
        setSize(1, 1);
    } catch (java.lang.Throwable ivjExc) {
        handleException(ivjExc);
    }
    // user code begin {2}
    // user code end
}
```

0 items selected.

Now you can check the application again.

Tip: For the purposes of this tutorial, we did not use all of the features of the Create Class SmartGuide to create our listeners and methods. To learn more about how to use the features of this SmartGuide, see Using the Create Class SmartGuide to Create Method Stubs.

---

< Previous Table of Contents Next >

# Testing your visual bean

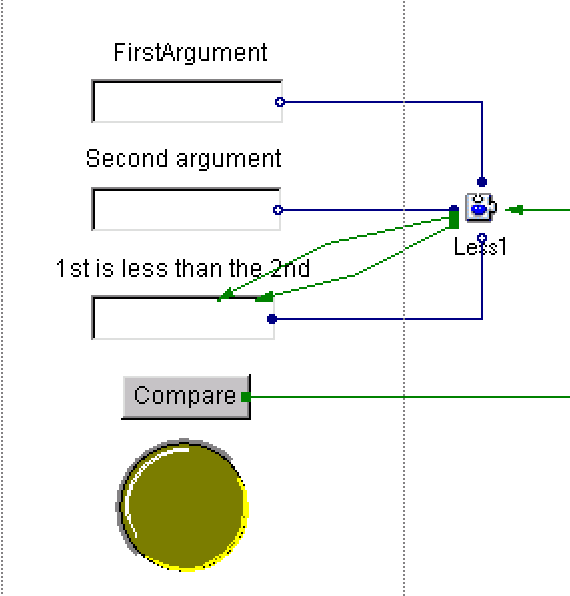
**Tutorial Contents**
User-generated events
Creating a visual bean
Adding a MouseListener
Testing your visual bean
Final Applet

***On this page:***

Placing the Semaphore bean
Making connections
Testing the Semaphore
Conclusion

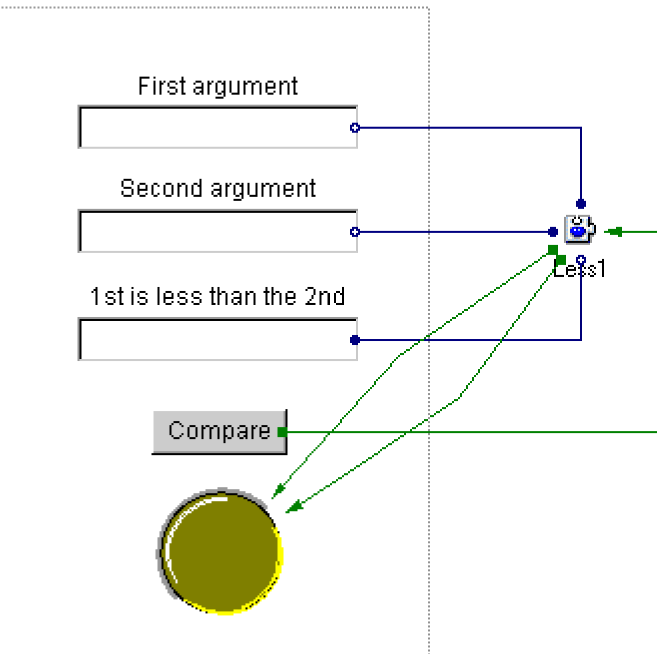
### Placing the Semaphore bean

1. Open the Test applet in the VCE
2. From the palette select the Choose bean icon: In the Choose Bean window add the Semaphore bean from the lesspackage to the applet with the help of the Browse button or by typing in the full name of the bean - "lesspackage.Semaphore".
3. Place and resize the Semaphore bean as shown in the picture below:




### Making connections

1. Delete two event connections from the Less bean to TextField3.
2. Connect Less1.onTrue to Semaphore.color (set the green color as the parameter).
3. Connect Less1.onFalse to Semaphore.color (set the red color as the parameter).

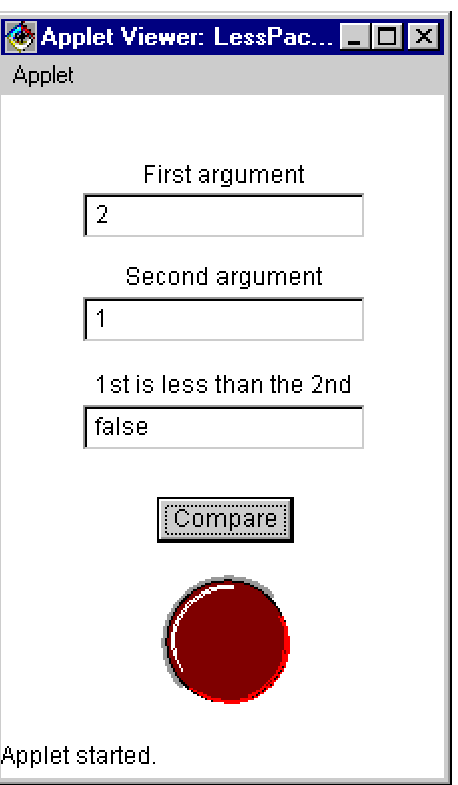



**Testing the Semaphore**

Let's test the application and see how it works:

Enter data in the both text fields and press the **Compare** button. Every time the result of comparison changes, the value of the result property of the Less bean gets assigned to the third TextField, and the color of semaphore indicates whether it is true or false.

Try to press mouse button on a Semaphore and notice how it reacts to this event.




This lesson has shown you how to construct your own visual, reusable bean and provide event listeners for it.

Check out our final applet.

**Conclusion**

You have used fairly complex code to create this Semaphore. However, you can see how easy it is to use this bean in the Visual Composition Editor. Your bean successfully encapsulates a particular task and hides all the complexity inside, and your users can easily make this bean interact well without code.

We hope that in this tutorial we've shown you that developing components is as easy as developing applications, and intuitively leads to better, more modular code. If you follow these design principles you really can reach a broader audience of visual programmers. In further installments we'll teach you more component development design patterns and answer any questions you might have.

**Express yourself:**

We want your ideas on what you'd like to see in future installments. Need some help with a particular bean or got a question on this tutorial? Let us hear from you at:

stephp@us.ibm.com

**Check out these sites:**

IBM's Java site
Javasoft
alphaWorks
SanFrancisco

---

< Previous Table of Contents Next >

# The code behind connections

**Bean design tutorials - Table of Contents**

**Tutorial Contents**

The code behind connections
Connections in practice
Using method return values

***On this page:***

Kinds of connections
Method names for connections
Connections one by one

***Time required:*** 1.5 hours

Can you imagine a manufacturer of car internal combustion engines that doesn't have a clue about how these engines are used in real vehicles? Of course you know that any successful manufacturer has to understand the real needs of customers in order to make these engines better and more suitable for their intended purpose.

Now, imagine a bean developer that doesn't understand what's happening behind the scenes when the application assembler wires these beans in an application. Most beginning bean developers fit this description (including your authors) -- we trust that it will all somehow magically work together but don't bother to learn how exactly VisualAge codes these wirings. When you learn the mechanisms that VisualAge for Java uses to produce programs, you'll not only learn how it is done, but this knowledge will enable you to produce better beans.

So - in this tutorial installment we'll show you the different connections that VisualAge for Java offers, and what code it generates to implement them.

**As you probably remember, there are five kinds of connections:**

1. ***Property-to-property*** () – synchronizes two property values on an event (for unbound properties) or continuously (for bound properties)
2. ***Event-to-method*** () – to make a method call whenever an event gets fired
3. ***Event-to-property*** () – to update a property whenever an event gets fired
4. ***Parameter*** () – to supply parameters for another connection
5. ***Event-to-code*** () – to execute a code fragment whenever an event gets fired

When you haven't provided sufficient information to make a connection, the connection line appears dashed (for example, an Event-to-method for which you haven't provided all parameters appears like this: ).

When users make connections, VisualAge for Java generates a private *initConnections()* method that gets called at the beginning of applet or application execution, does the initial execution of *property-to-property* connections and establishes listeners for event connections.

VisualAge for Java generates a private method for each method or code connection. For each *property-to-property* connection where both end points are writeable, it generates two methods. *Parameter* connections are the simplest type of connections - they appear in code as secondary calls within the original connection. Their only duty is to pass parameters to methods or property setter methods, which is why you can't find their names in class/interface browser.

The name of the connection method depends on the type of connection you draw. By default, the naming convention is *connFtoFx* , where *F* represents the type of feature being connected and *x* is an index number to ensure uniqueness.

To access the source code for any connections, choose the *Members* or *Hierarchy* page from the Class/Interface browser and click on the name of the current connection. The source code of the selected connection appears below in the source window.

**These connections translate into the following method names:**

- *connEtoM1* - the first event-to-method or event-to-property connection drawn
- *connEtoC1* - the first event-to-code connection drawn
- *connPtoP1setTarget* - the first property-to-property connection drawn, setting the target from the source
- *connPtoP1setSource* - the first property-to-property connection drawn, setting the source from the target

**Let's consider these connections one by one:**

**1. Event-to-code** ( - *connEtoC1.*)

Used to execute your code on an event. You can think of this connection as a event-to-method connection where the method belongs to the class that you are visually composing at the moment. Usually you will create a new method, but you can also use an existing one.

**2. Event-to-property** ( - *connEtoM1*)

Used to set a property value (specified in the parameter editor or by using a parameter connection) of an event. You can think of this connection as an event-to-method connection with the setter method of a property (as the naming convention suggests).

**3. Event-to-method** ( - *connEtoM1*)

Used to execute a public method of beans that you are composing your application from. You can select whether to pass event data or not for these connections. If you do, VisualAge for Java uses the event object as the first parameter to the method, if it can establish the type conversion. You specify the method parameters in the parameter editor or by using parameter connections.

**4. Property-to-property** ( - *connPtoP1*)

Used when you want two bean properties to have equal values at certain times. The key characteristics of these connections are:

- Source property
- Target property
- Source event
- Target event

Source and target events have to be set by you as a developer. By default VisualAge for Java sets events to <none>. If you leave both source and target events set to '<none>', the target only gets aligned with the source on initialization. When you set a source event, the target gets aligned with the source value when the event is triggered as well. When you set a target event, the source gets aligned with the target value, when that event is triggered. These events let you control whether data synchronization is unidirectional, bidirectional, or only performed at initialization.

**5. Parameter connections** ( )

Use these types of connections when providing parameters for other connections with methods (or specifying property values) if they are not constants that you can supply in the property editor. There are 3 kinds of parameter connections:

- *Parameter-from-code* - generates a new method whose name is constructed from the connection name this connection is providing a parameter for, and the parameter name. This method is then used just like if you had a parameter-from-method connection.
- *Parameter-from-method* - used to get the parameter value from the specified method.
- *Parameter-from-property* - if a property is readable, Visual Age for Java generates a getter method for it. A parameter-from-property connection is identical to a parameter-from-method connection with the getter method for the appropriate property. Imagine, for example, that you're using an Event-to-method connection to call bean A's *foo()* method that takes one string argument. You are supplying this argument from bean B's *bar* property. The code statement that VisualAge for Java generates looks like this:

```
getA().foo( getB().getBar() );
```

---

< Previous Table of Contents Next >

## Connections in practice

**Tutorial Contents**
The code behind connections
Connections in practice
Using method return values

***On this page:***

Setting up your project
Placing components
Making an Event-to-Code connection
Making an Event-to-Property connection
Making an Event-to-Method and a Parameter connection
Making a Property-to-Property connection
Reordering the connections

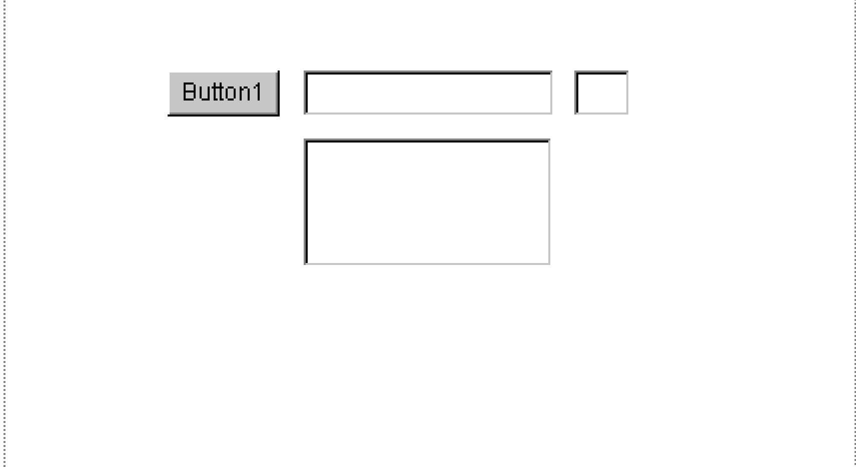
To see the code that gets generated by each type of connection, we'll develop a simple applet that uses all of them. You can download the final code and this tutorial in vptut3cd.zip.

### Setting up your project

1. Create a project called "*Connections*"
2. Create a package named "*connections*" in it.
3. Import beans from the packages provided in vptut3cd.jar.
   (You need com.ibm.uicontrols.BoundList, com.ibm.wiringhelpers.Iterator and com.ibm.sort.SortBean beans from this package.)
4. Create an applet called "*ConnectionsApplet*".

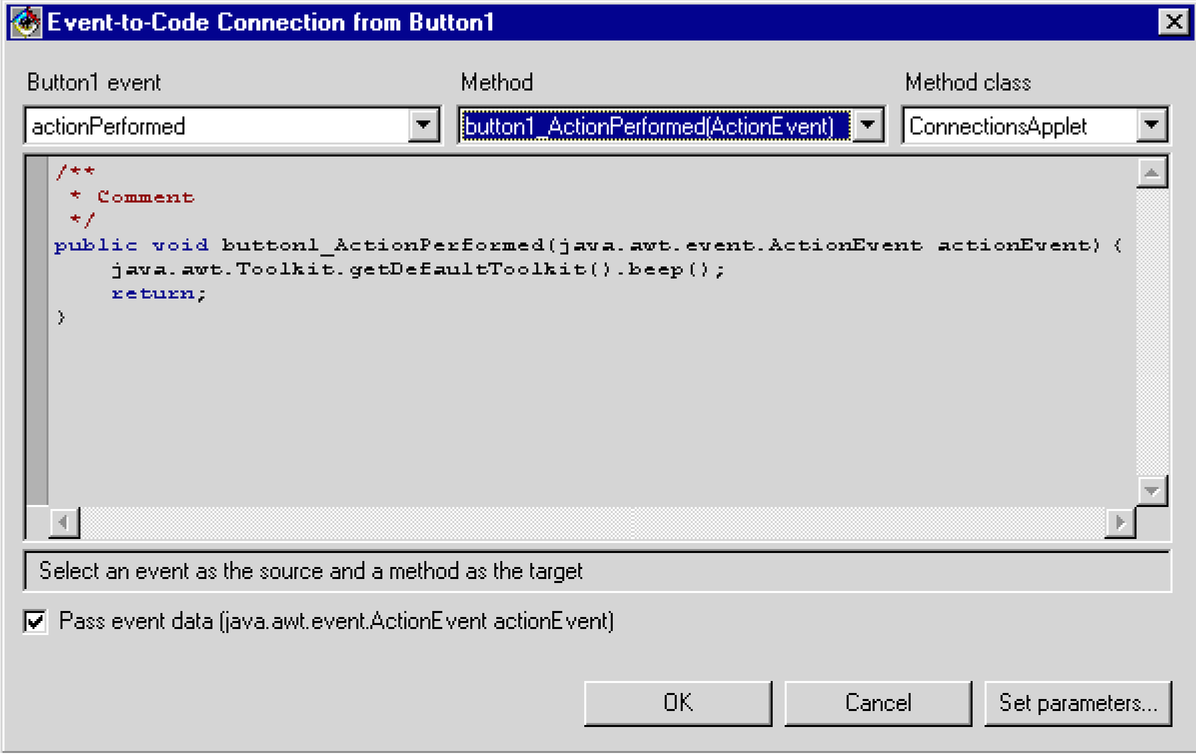
### Placing components

Place a java.awt.Button, two java.awt.TextFields and a com.ibm.uicontrols.BoundList as shown below:

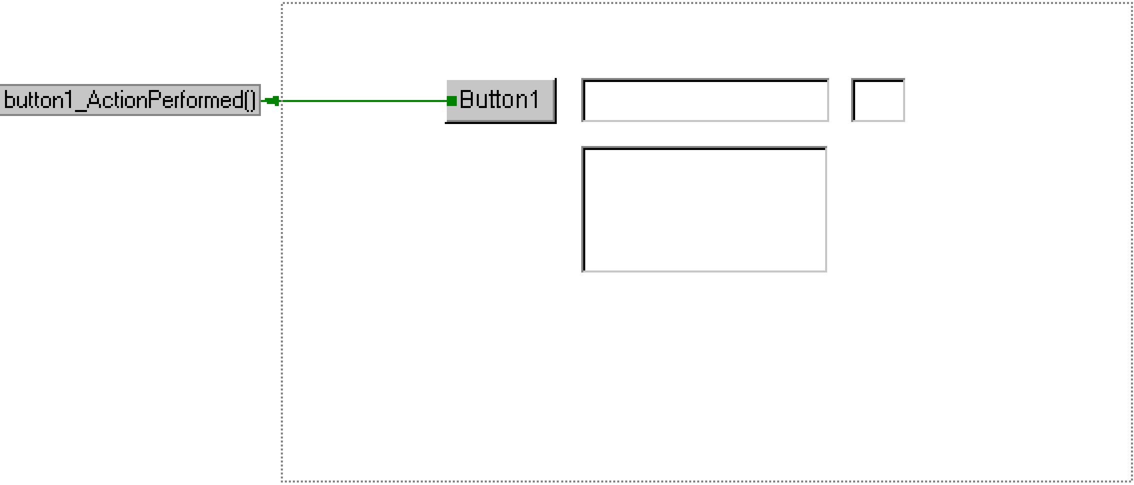



### Making an Event-to-Code connection to beep on the button click

1. Right-click Button1, and select **actionPerformed** from the pop-up menu that appears.
2. Drag the connection line to the free-form surface outside the applet container, and left-click the mouse.
3. Select the **Event-to-Code**... option from the pop-up menu.
4. An Event-to-Code connection window appears that lets you input code to implement your connection.
5. Type "`java.awt.Toolkit.getDefaultToolkit().beep();`" as shown below, and save your changes.




6. Rearrange the design surface and connections to get this alignment:




**Understanding the generated code.**

To see generated code, you need to save your work so that changes take effect.

- Your applet now implements java.awt.event.ActionListener.
- VisualAge for Java creates the following *initConnections* method and calls it from the *init()* method, which allows the applet to receive ActionEvent notification events from the button:

```
/**
* Initializes connections
* @exception java.lang.Exception The exception description.
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void initConnections() throws java.lang.Exception {
    // user code begin {1}
    // user code end
    getButton1().addActionListener(ivjEventHandler);
}
```

- The *actionPerformed* method which (this resides in the IvjEventHandler class - inner class of our application) determines the source for the ActionEvents, and calls the appropriate connection's method.

***VisualAge for Java 2.0 users:*** Because VisualAge for Java 2.0 does not generate any inner classes for handling events, the *actionPerformed()* method is generated directly in the ConnectionsApplet class.

```
/**
* Insert the type's description here.
* Creation date: (4/27/00 2:53:32 PM)
* @author:
*/
public class ConnectionsApplet extends java.applet.Applet {

class IvjEventHandler implements java.awt.event.ActionListener {
        public void actionPerformed(java.awt.event.ActionEvent e) {
            if (e.getSource() == ConnectionsApplet.this.getButton1())
                connEtoC1(e);
        };
    };
    private com.ibm.uicontrols.BoundList ivjBoundList1 = null;
    private java.awt.Button ivjButton1 = null;
    IvjEventHandler ivjEventHandler = new IvjEventHandler();
    private java.awt.TextField ivjTextField1 = null;
    private java.awt.TextField ivjTextField2 = null;
}
```

- The new method you just created, *button1_ActionPerformed*, gets called when the event occurs, and now contains the code that you added.

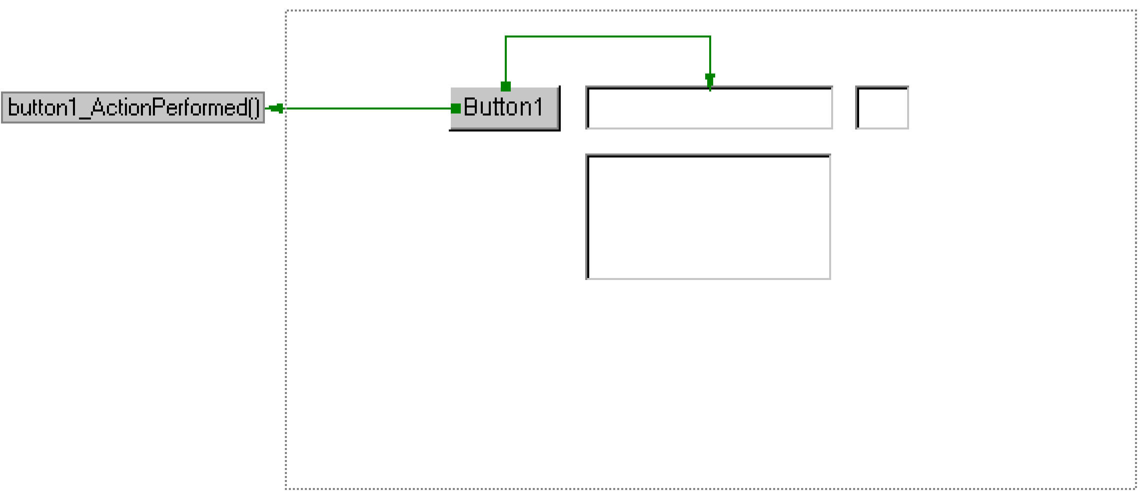
```
 /**
* Comment
*/
public void button1_ActionPerformed(java.awt.event.ActionEvent actionEvent) {
    java.awt.Toolkit.getDefaultToolkit().beep();
    return;
}
```

- A connection method for the connection we just made calls the *button1_ActionPerformed* method you just created. If you had used an existing method, it would be called here instead.

```
/**
* connEtoC1: (Button1.action.actionPerformed(java.awt.event.ActionEvent) -->
ConnectionsApplet.button1_ActionPerformed(Ljava.awt.event.ActionEvent;)V)
* @param arg1 java.awt.event.ActionEvent
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void connEtoC1(java.awt.event.ActionEvent arg1) {
    try {
        // user code begin {1}
        // user code end
        this.button1_ActionPerformed(arg1);
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}
```

**Making an Event-to-Property connection to clear the text field**

1. Connect the *actionPerformed* event of the Button to the text property of the *TextField*.
2. Set the *value* property of this connection to an empty string. To do this you will have to first set it to something else, and then to blank string.




3. Now save your changes before continuing to the next section.

**Understanding the generated code**

- VisualAge for Java adds the following lines of code to the *actionPerformed( ActionEvent e )* method to invoke the connection we just created, when the event is fired. The order of these code blocks corresponds to the order in the **Reorder connections from...** menu.

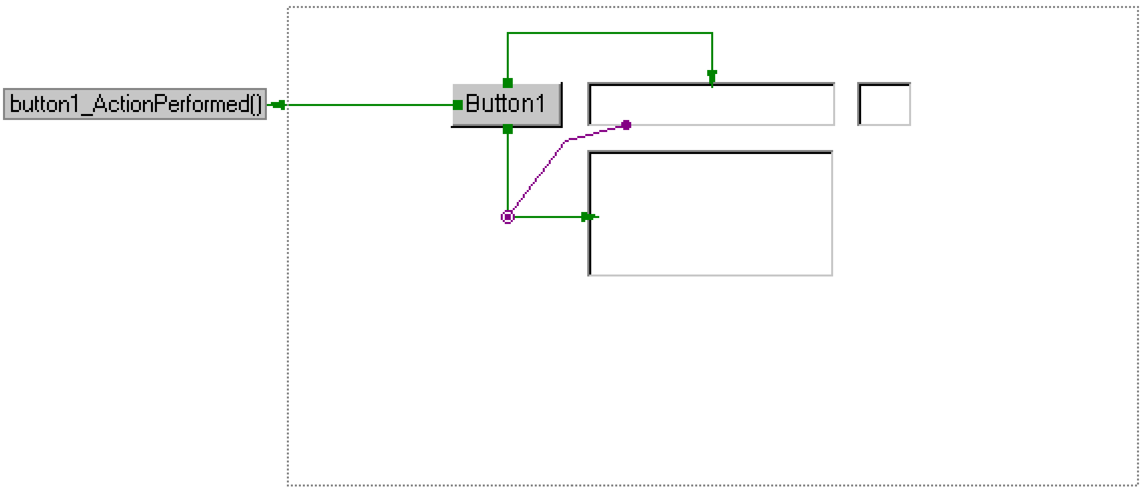
```
if (e.getSource() == ConnectionsApplet.this.getButton1())
connEtoM1(e);
```

- *connEtoM1* calls the appropriate setter method for the property that you set, with the specified parameters.

```
/**
* connEtoM1: (Button1.action.actionPerformed(java.awt.event.ActionEvent) -->
TextField1.text)
* @param arg1 java.awt.event.ActionEvent
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void connEtoM1(java.awt.event.ActionEvent arg1) {
    try {
        // user code begin {1}
        // user code end
        getTextField1().setText("");
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}
```

**Making an Event-to-Method and a Parameter connection to add the string to the bound list**

1. Connect Button1's *actionPerformed* event to the *add(String)* method of the BoundList.
2. Connect the *item* property of this connection to the *text* property of the TextField.




3. To view the generated code, save your changes and go to the Members panel.

**Understanding the generated code**

- Another code block for ActionEvent processing was added to the *actionPerformed* method:

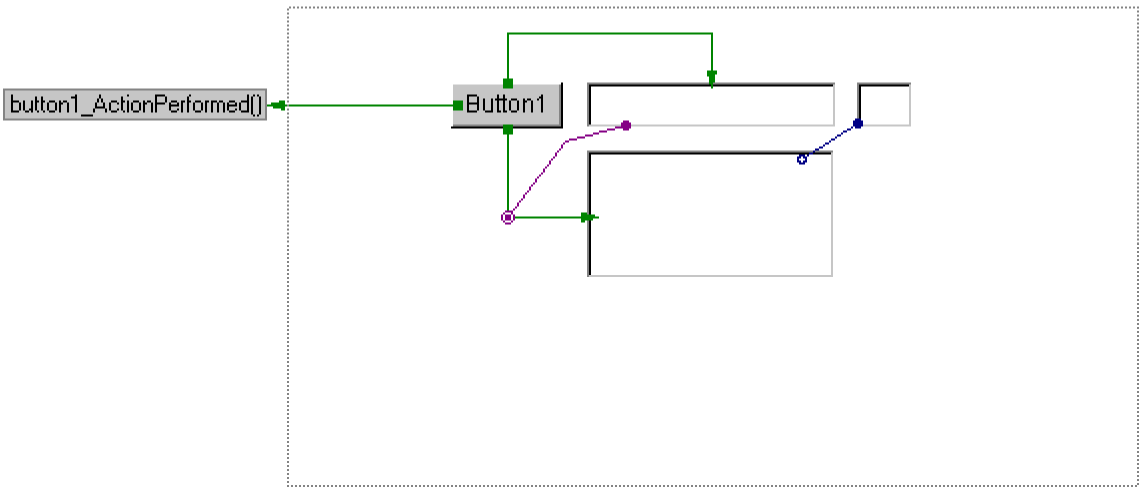
```
if (e.getSource() == ConnectionsApplet.this.getButton1())
connEtoM2(e);
```

- As you can see here, VisualAge for Java translated the parameter-from-property connection into "getTextField1().getText()" and used that code as a parameter to the method this connection refers to.

```
/**
* connEtoM2: (Button1.action.actionPerformed(java.awt.event.ActionEvent) -->
BoundList1.add(Ljava.lang.String;)V)
* @param arg1 java.awt.event.ActionEvent
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void connEtoM2(java.awt.event.ActionEvent arg1) {
    try {
        // user code begin {1}
        // user code end
        getBoundList1().add(getTextField1().getText());
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}
```

**Making a Property-to-property connection to show the number of items in the bound list**

- Connect the *itemCount* property of the BoundList to the *text* property of the right TextField.




Now save your work to see the generated code.

**Understanding the generated code**

- The applet is now also a java.beans.PropertyChangeListener so that it can listen for events that get fired on a bound property change.
- This `connPtoP1SetTarget` method sets the target property from the source property for this connection (a similar method to set the source did not get generated, because the source is a read-only property):

```
/**
* connPtoP1SetTarget: (BoundList1.itemCount <--> TextField2.text)
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void connPtoP1SetTarget() {
    /* Set the target from the source */
    try {
        getTextField2().setText(String.valueOf(getBoundList1().getItemCount()));
        // user code begin {1}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}
```

- VisualAge for Java adds the following lines to the *initConnections* method to receive PropertyChangeEvents, and to initialize this property-to-property connection.

```
getBoundList1().addPropertyChangeListener(ivjEventHandler);
connPtoP1SetTarget();
```

***VisualAge for Java 2.0 users:*** generated code looks like it is shown below:

```
getBoundList1().addPropertyChangeListener(this);
connPtoP1SetTarget();
```

- The *propertyChange* event processes all PropertyChangeEvents, and invokes the appropriate methods that in turn invoke setter methods for the properties involved. This resides in the *IvjEventHandler* inner class of our application.

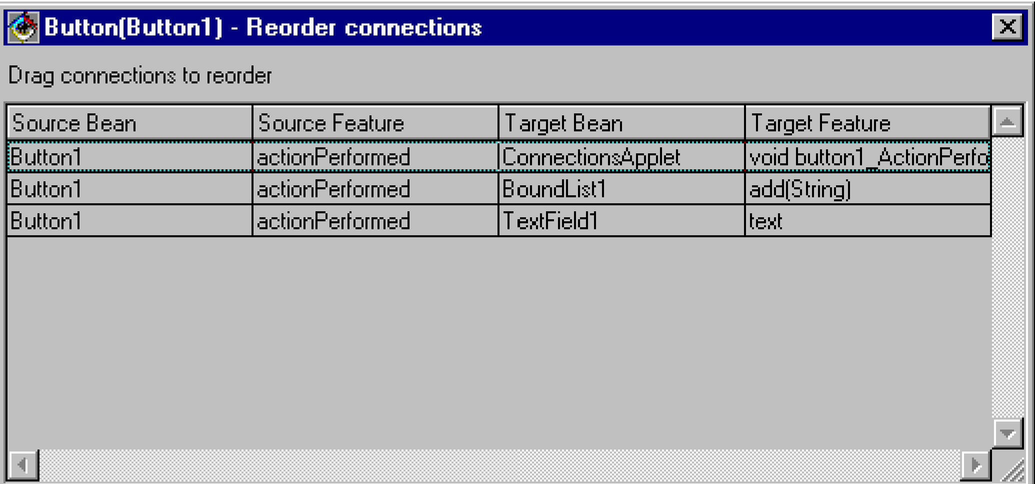
```
    public void propertyChange(java.beans.PropertyChangeEvent evt) {
            if (evt.getSource() == ConnectionsApplet.this.getBoundList1() &&
(evt.getPropertyName().equals("itemCount")))
                connPtoP1SetTarget();

    };
```

***VisualAge for Java2.0 users:*** the method shown above could be found directly in the ConnectionsApplet class and looks like it shown below:

```
/**
* Method to handle events for the PropertyChangeListener interface.
* @param evt java.beans.PropertyChangeEvent
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
public void propertyChange(java.beans.PropertyChangeEvent evt) {
    // user code begin {1}
    // user code end
    if ((evt.getSource() == getBoundList1()) &&
(evt.getPropertyName().equals("itemCount"))) {
        connPtoP1SetTarget();
    }
    // user code begin {2}
    // user code end
}
```

**Reordering the connections**

If you try to run your applet now, you can see that the items added to the bound list are all empty. As you know, the order in which connections are set up is important, which can sometimes cause unexpected errors. The defect in example is that a Parameter connection tries to get text from the text field after it has been cleared by the first Event-to-Property connection. To resolve this problem you have to reorder the connections using the Reorder connections window.  To access this window, right-click on Button1 and choose **Reorder connections from**. Drag the connection where the Target Feature is *text* to the bottom so that it looks like this:




Save the bean and select the ConnectionsApplet class from the workbench. Now take a look at the source code of the inner class *IvjEventHandler*:

***VisualAge for Java 2.0 users:***
To see the code that executes connections: save the bean, select the ConnectionsApplet class from the workbench, find its *actionPerformed(java.awt.ActionEvent e*) method and take a look at its source code.

Source

```
public class ConnectionsApplet extends java.applet.Applet {
    private com.ibm.uicontrols.BoundList ivjBoundList1 = null;
    private java.awt.Button ivjButton1 = null;
    private java.awt.TextField ivjTextField1 = null;
    private java.awt.TextField ivjTextField2 = null;
    IvjEventHandler ivjEventHandler = new IvjEventHandler();

class IvjEventHandler implements java.awt.event.ActionListener, java.bea
        public void actionPerformed(java.awt.event.ActionEvent e) {
            if (e.getSource() == ConnectionsApplet.this.getButton1())
                connEtoC1(e);
            if (e.getSource() == ConnectionsApplet.this.getButton1())
                connEtoM2(e);
            if (e.getSource() == ConnectionsApplet.this.getButton1())
                connEtoM1(e);
```

connections.ConnectionsApplet (00.03.24. 12:09:47)

As you see, the *connEtoM1* code fragment now is after *connEtoM2* to reflect the changes we've made, and our application now works as planned.

---

< Previous Table of Contents Next >

# Using method return values

***On this page:***

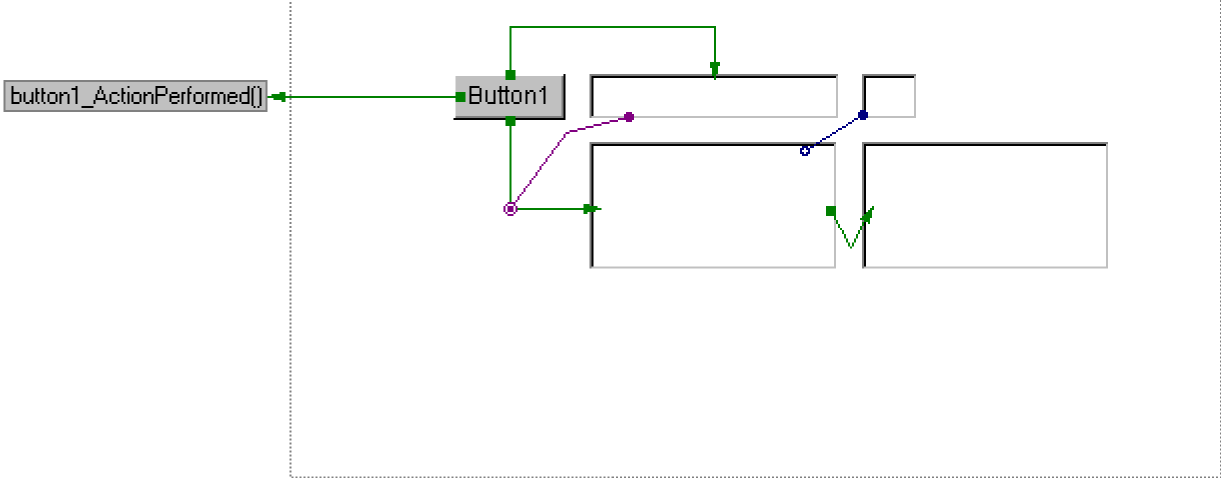
[Making connections](#)
[Understanding the generated code](#)
[Conclusion](#)

**Tutorial Contents**
The code behind connections
Connections in practice
Using method return values

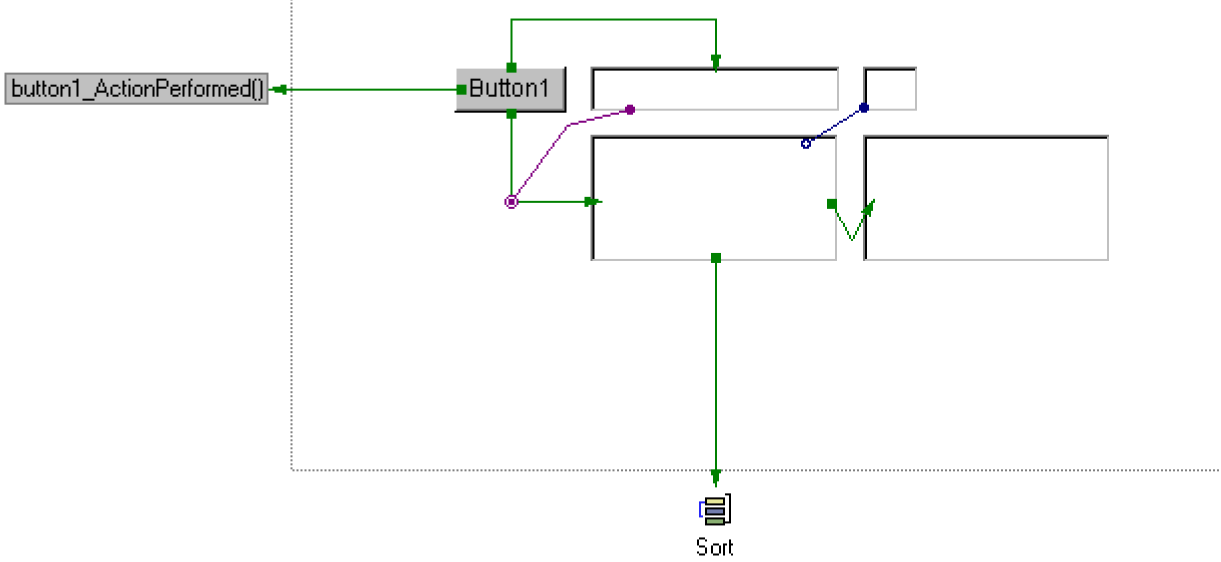
### Making the connections

As you remember, the first time we used method return values in the Visual programming tutorial was in Working with the task list  when sorting tasks in the TodoList. So, let's try to combine our current application with the Working with the task list part of the TodoList applet, to examine the code that gets generated when we are using Event-to-Method connections that provide method return values.

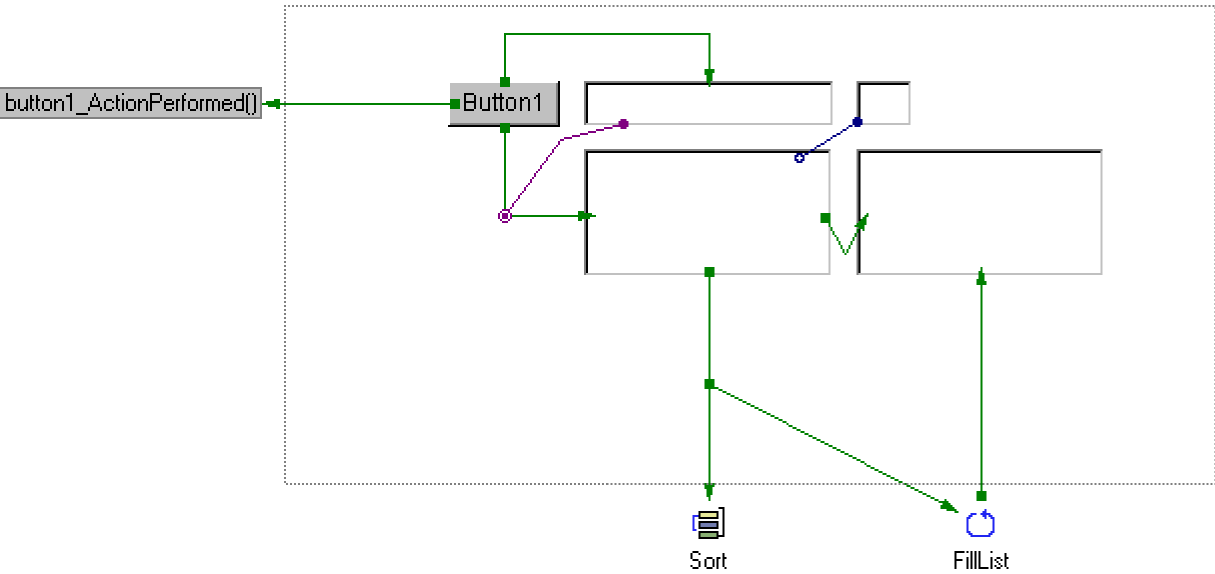
1. Add one more BoundList to the ConnectionsApplet to place the sorted items in.
2. Right-click BoundList1, and connect *Items* property to the *removeAll()* method of the BoundList2 as shown below. This clears the second bound list before reinserting items there.




3. Now add the SortBean to the free-form surface and set the bean name to *Sort.* The Sort bean has a *sortStringArray* method that we use to sort our list of Strings.
4. Connect the *Items* event of the first BoundList to the *sortStringArray(java.lang.String[])* of the Sort bean. This creates an Event-to-Method connection that sorts the list items and returns the sorted String array.
5. Double-click on this connection and check the **Pass event data** check box. The event data for a property change event is the property that has changed - we want to pass it as the first parameter to the sort method (so that it knows what to sort).
6. For now your application should look like this:



7. Now add the *Iterator* bean to the free-form surface and rename it to *FillList*. Iterator takes a String array and fires the *currentStringResult* event for each item processed.
8. Connect the *normalResult* of the most recently created connection (that's the return value of the preceding connection) to the *inputStringArray* of the FillList bean. This provides the input data for the Iterator bean.
9. Connect the *currentStringResult* of the FillList bean to the *add(String)* method of the BoundList2. This adds every String from the sorted String list to the second BoundList.
10. Double-click on this connection and check the **Pass event data** check box. This provides a parameter for the *add(String)* method - the current element in the iteration.
11. Your final application now looks like this and works as expected - the items in the second listbox are sorted:




12. Save your changes and we'll see what code has been generated.

**Understanding the generated code**

- The first connection we made is the Event-to-Method connection between the two BoundLists. When the *items* property changes, the *connEtoM3* method clears all items in BoundList2. It is important that you make this connection before all other connections are made, otherwise the *connEtoM3* method clears the second BoundList after the FillList bean adds sorted items in it, and you have to reorder connections as we showed you before.

```
/**
* connEtoM3: (BoundList1.items --> BoundList2.removeAll()V)
* @param arg1 java.beans.PropertyChangeEvent
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void connEtoM3(java.beans.PropertyChangeEvent arg1) {
    try {
        // user code begin {1}
        // user code end
        getBoundList2().removeAll();
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}
```

- We have created an Event-to-Method connection that returns a String array that the sortStringArray method of the Sort bean returns (`connEtoM4Result = getSort().sortStringArray(getBoundList1().getItems());`) as a normalResult. As you see from the generated code, the connEtoM5 method uses the return value of this method as a parameter to pass to the FillList bean.

```
/**
```

```
* connEtoM4: (BoundList1.items --> Sort.sortStringArray([Ljava.lang.String;)
[Ljava.lang.String;)
* @return java.lang.String[]
* @param arg1 java.beans.PropertyChangeEvent
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private java.lang.String[] connEtoM4(java.beans.PropertyChangeEvent arg1) {
    java.lang.String[] connEtoM4Result = null;
    try {
        // user code begin {1}
        // user code end
        connEtoM4Result = getSort().sortStringArray(getBoundList1().getItems());
        connEtoM5(connEtoM4Result);
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
    return connEtoM4Result;
}
```

- Now let's try to see what happens with the returned value of the preceding connection. The lines of code in the connEtoM5 method shown below set the String array passed to it by the *connEtoM4* connection and set it as the value for the *inputStringArray* property of the FillList bean:

```
/**
* connEtoM5: ( (BoundList1,items --> Sort,sortStringArray([Ljava.lang.String;)
[Ljava.lang.String;)
  .normalResult --> FillList.inputStringArray)
* @param result java.lang.String[]
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void connEtoM5(java.lang.String[] result) {
    try {
        // user code begin {1}
        // user code end
        getFillList().setInputStringArray(result);
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
    }
}
```

- As expected, the FillList bean adds an array of sorted items to the BoundList2 using an Event-to-Method connection (connEtoM6). The following generated code implements the connEtoM6 method:

```
/**
* connEtoM6: (FillList.currentStringResult -->
BoundList2.add(Ljava.lang.String;)V)
* @param arg1 java.beans.PropertyChangeEvent
*/
/* WARNING: THIS METHOD WILL BE REGENERATED. */
private void connEtoM6(java.beans.PropertyChangeEvent arg1) {
    try {
        // user code begin {1}
        // user code end
        getBoundList2().add(String.valueOf(getFillList()
         .getCurrentStringResult()));
        // user code begin {2}
        // user code end
    } catch (java.lang.Throwable ivjExc) {
        // user code begin {3}
        // user code end
        handleException(ivjExc);
```

```
     }
}
```

- The following line was added to the *initConnections()* method to add this applet as a PropertyChangeListener for the FillList bean, so that we can resort the items whenever the *items* property changes:

  ```
  getFillList().addPropertyChangeListener(ivjEventHandler);
  ```

  ***VisualAge for Java2.0 users:*** this string looks like it is shown below:

  ```
  getFillList().addPropertyChangeListener(this);
  ```

- The following lines of code in the *propertyChange(PropertyChangeEvent)* method process events that get fired on a bound property change, and call the appropriate Event-to-Method connection methods:

  ```
     if (evt.getSource() == ConnectionsApplet.this.getBoundList1() &&
  (evt.getPropertyName().equals("items")))
                  connEtoM3(evt);
     if (evt.getSource() == ConnectionsApplet.this.getBoundList1() &&
  (evt.getPropertyName().equals("items")))
                  connEtoM4(evt);
     if (evt.getSource() == ConnectionsApplet.this.getFillList() &&
  (evt.getPropertyName().equals("currentStringResult")))
                  connEtoM6(evt);
  ```

**Conclusion**

Hopefully the applet you've just developed has helped you understand the relationship between connections and the code that implements them. You also learned what VisualAge for Java generates for the code behind methods that use return values. One of the most effective ways to solve a problem with a wiring is to take a look at the code that implements it, and even try to debug it.

---

| **Express yourself:** | **Check out these sites:** |
|---|---|
| We want your ideas on what you'd like to see in future installments. Need some help with a particular bean or got a question on this tutorial? Let us hear from you at: stephp@us.ibm.com | IBM's Java site<br>Javasoft<br>alphaWorks<br>SanFrancisco |

---

< Previous    Table of Contents    Next >