



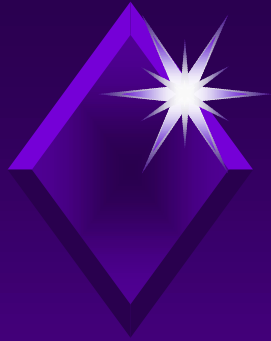
Behind the Beans
- An Introduction to JavaBeans

Michael Stal

Siemens AG, Corporate Technology

Dept. ZT SE 2

E-mail: Michael.Stal@mchp.siemens.de



Content

- ◆ Motivation
- ◆ Essentials of JavaBeans
- ◆ Digging Deeper
- ◆ A Sample Bean
- ◆ The Bean Developer Kit
- ◆ JavaBeans and ActiveX Controls
- ◆ Related Technologies
- ◆ JavaBeans - the Future
- ◆ Summary

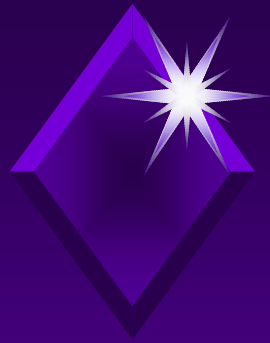




Motivation

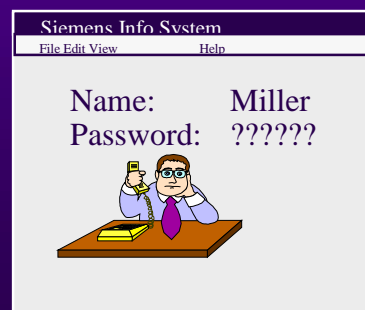
◆ Why Components?



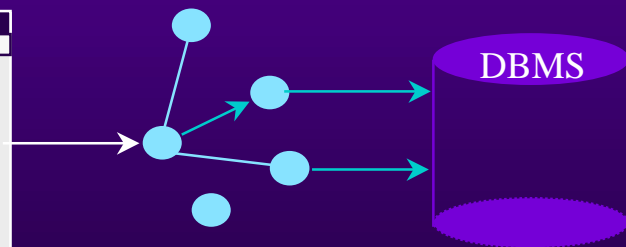


Component-based Software

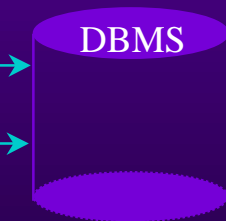
- ◆ Multi-tier architectures lead to a separation of concerns (presentation, application, data).
- ◆ But presentation / application tiers are complex.
- ◆ Thus, an additional separation of these tiers is necessary!



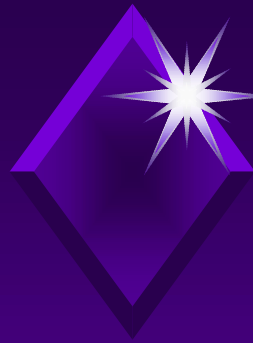
Presentation Layer



Business Objects

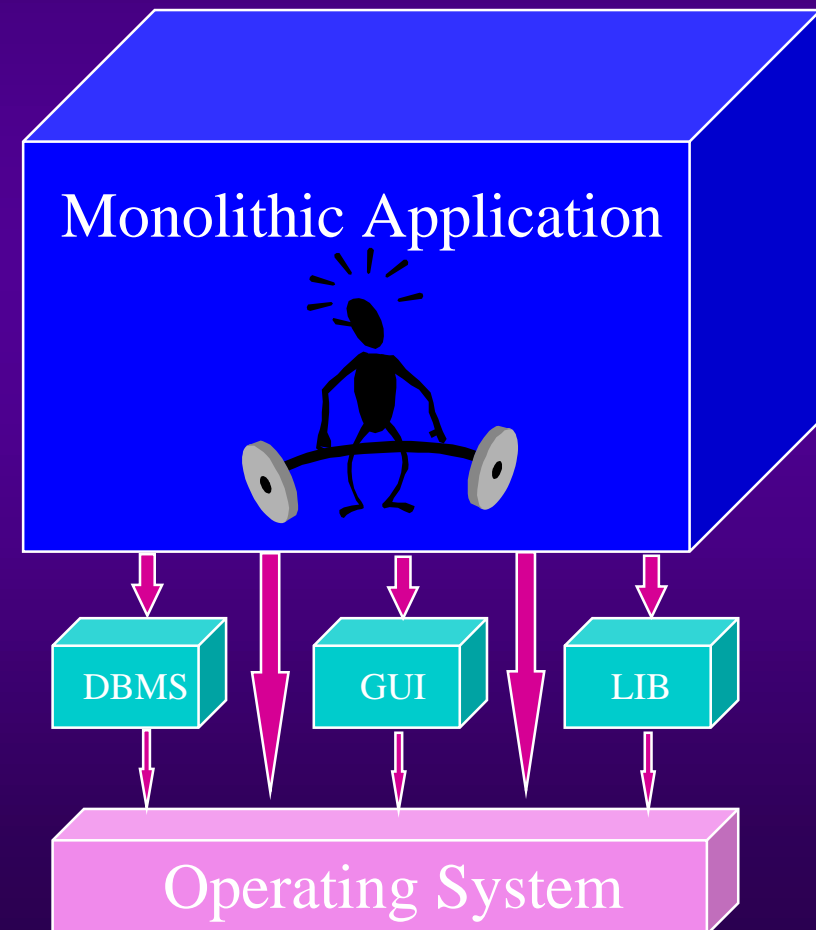


Database Layer



The Problem: Monolithic Software built from scratch

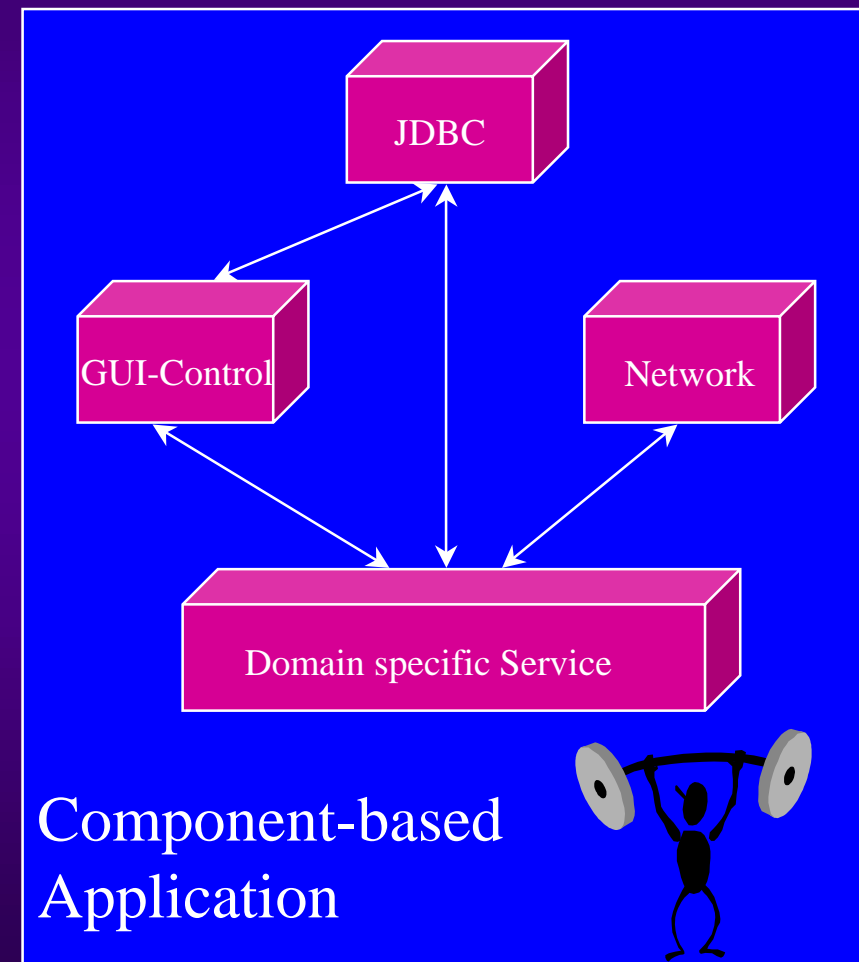
- ◆ Monolithic Applications
- ◆ Multi-Paradigm Approach
- ◆ Little Software Reuse = Low Productivity + High Costs
- ◆ Make-not-Buy
- ◆ Poor Changeability and Extensibility

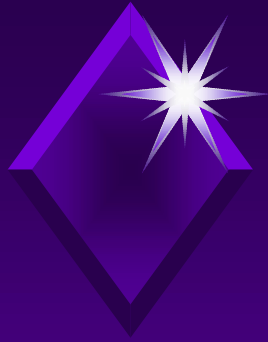




Components

- ◆ Software may be built using existing services
- ◆ One-Paradigm Approach (OO)
- ◆ Reusable Software
- ◆ Make-or-(better) Buy
- ◆ Design for Changeability, Exchangeability



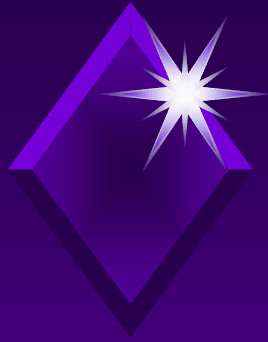


What is a Component?

◆ Definition:

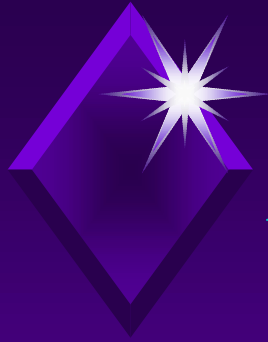
A component denotes a self-contained entity that

- ◆ *provides its functionality via standard interfaces,*
- ◆ *uses functionality from its environment via standard interfaces,*
- ◆ *must be prepared to be used in contexts that cannot be anticipated,*
- ◆ *and may support builder tools in plugging components and applications together.*

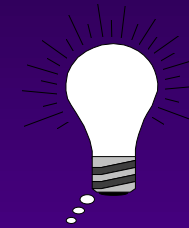


Flavors of Components

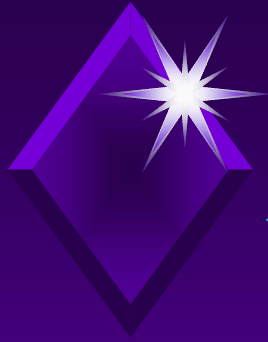
- ◆ JavaBeans are presentation tier components:
 - ◆ they typically represent sophisticated GUI elements.
 - ◆ they share the same address space with their clients.
 - ◆ their clients are containers that provide all the resources.
 - ◆ they send events to their containers.
- ◆ Enterprise JavaBeans are middle tier components:
 - ◆ they typically provide server-side functionality.
 - ◆ they run in their own address space.
 - ◆ they are integrated into a container that hides all system details.



All U Need Is Glue

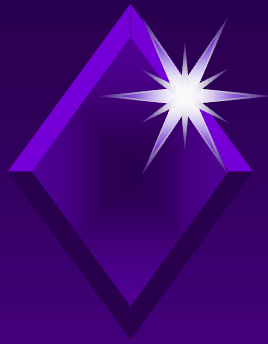


- ◆ How to connect (remote) components and applications:
 - Binary objects (components) as distribution units.
 - Distributed Component Models - *not* limited to the boundaries of a process or network node.
 - Homogeneous APIs on top of heterogeneous systems for encapsulating system details.
 - Remote Method Invocation paradigm.
 - Straight-forward integration of legacy code using bi-directional adapters.



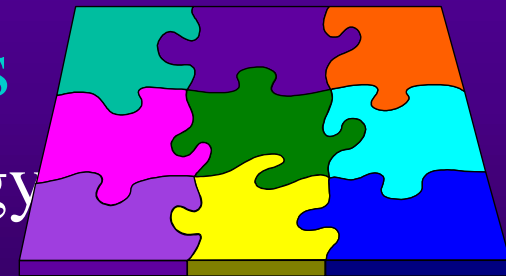
All U Need Is Glue (cont'd)

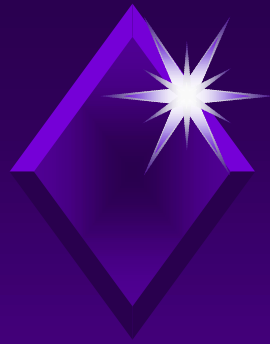
- ◆ Topics to be addressed:
 - ◆ How to access remote services in a location-transparent way?
 - ◆ How to handle (de-)marshaling issues?
 - ◆ How to find remote objects?
 - ◆ How to activate remote objects?
 - ◆ How to keep state persistent and consistent?
 - ◆ How to solve security issues?
 - ◆ Synchronous/asynchronous communication



Component Technologies

- ◆ Microsoft COM+/MTS
 - ◆ Windows-specific Component Technology.
- ◆ OpenDoc Parts
 - ◆ Gone with the Wind !
- ◆ JavaBeans/Enterprise JavaBeans
 - ◆ Java-based Component Technology
 - ◆ JavaSoft's answer to MTS/COM+.

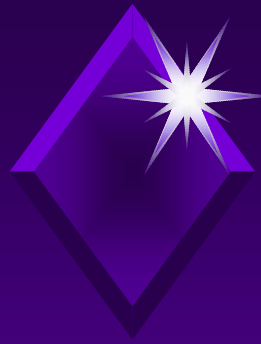




Essentials of JavaBeans

◆ Key Issues of JavaBeans

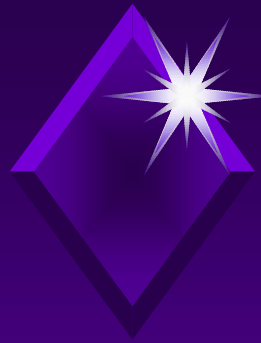




Essentials of JavaBeans

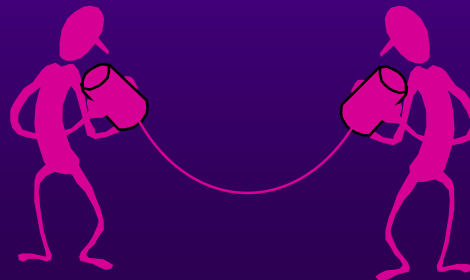
(cont'd)

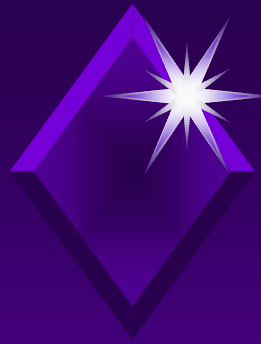
- ◆ The following questions need to be solved by a Component Technology such as JavaBeans:
 - ◆ how to connect a bean to the surrounding environment and to access the functionality it provides.
 - ◆ how to customize a bean and store the customized state.
 - ◆ how to let builder tools dynamically retrieve and use information about “unknown” beans.
 - ◆ how to package a bean and all of its required resources.
 - ◆ how to embed beans in compound documents.



Essentials of JavaBeans *(cont'd)*

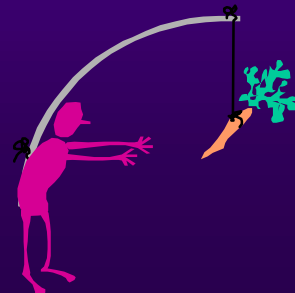
- ◆ Beans can notify registered entities (i.e. containers or other beans) about specific **events**:
 - ◆ Entities can register with all **semantic events** of a bean (e.g., “*button_clicked*”) they are interested in.
 - ◆ **Beans** generate events and **pass** self-describing **event** objects to the registered event listeners.

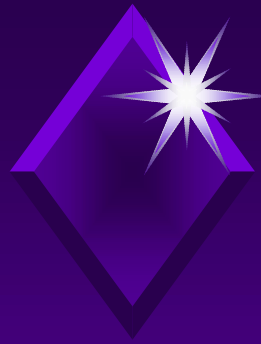




Essentials of JavaBeans *(cont'd)*

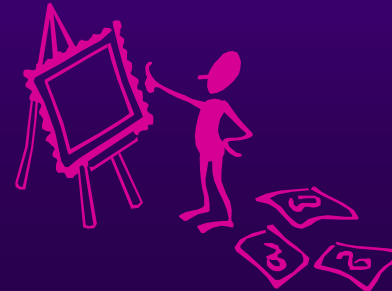
- ◆ Beans export functionality through **Properties** and **Methods**:
 - ◆ **Methods** such as `contains()` can be accessed from any other entity. They define the behavior of a bean.
 - ◆ **Properties** such as `backgroundColor` define the appearance of a bean. They are get or set from other entities for customization and programmatic use.

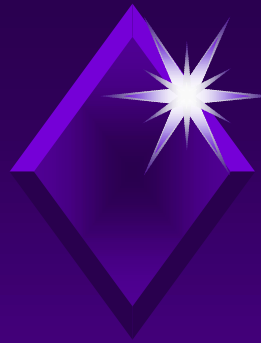




Essentials of JavaBeans *(cont'd)*

- ◆ Beans are **customized** by modifying their properties. A user can customize a bean by:
 - ◆ **programmatically** changing properties *or*
 - ◆ editing properties within a builder tool using **Property Sheets & Editors** *or*
 - ◆ automatically being guided through customization by **customizers**, i.e. graphical user interfaces sometimes also denoted as **wizards**.



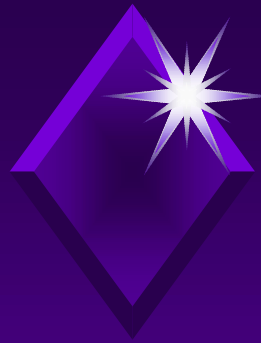


Essentials of JavaBeans

(cont'd)

- ◆ Suppose, you would need to customize all beans whenever the application is loaded. Solution:
 - ◆ The **Java Serialization API** allows builder tools to save the customized state of beans. Future versions of JavaBeans will also support **Externalization**.
 - ◆ Whenever an application is loaded, the saved states of beans are restored.



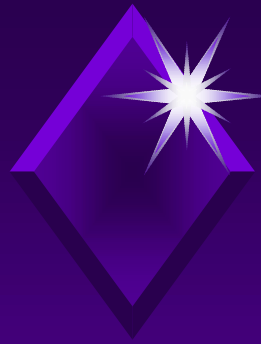


Essentials of JavaBeans


(cont'd)

- ◆ What about the packaging of beans ?
 - ◆ Beans are delivered using compressed archive files (**JAR files**).
 - ◆ JAR files contain the class files of beans as well as resource files such as images, sounds, videos. In addition, a manifest file describes the contents of the archive.





Essentials of JavaBeans *(cont'd)*

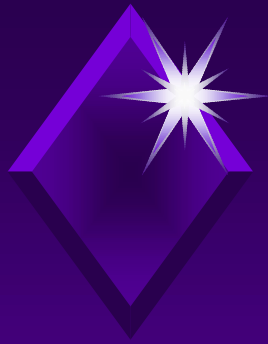
- ◆ We want to use builder tools to help us building applications consisting of beans. How can a builder tool handle beans it does not know ?
- ◆ A special **Introspection** API allows builder tools to analyze existing beans and access them programmatically.
- ◆ Beans are wired together in builder tools (*design-time mode*) and utilized in applications (*run-time mode*).



Digging Deeper

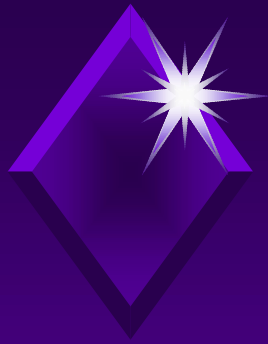
- ◆ Now, that we know the essentials let us grasp the details of bean development







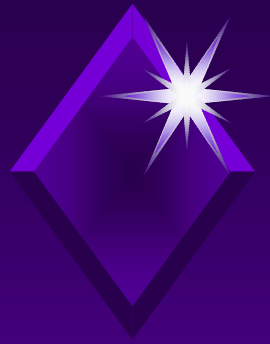
Miscellaneous Issues

- ◆ There is no common super class for beans. The `java.beans API` just provides helper classes.
- ◆ Everything is specified in Java. No specification language is required such as Microsoft IDL.
- ◆ To support builder tools programmers must conform to some `naming conventions`.
- ◆ The `separation of code for run-time and design-time` is supported to reduce memory footprint.



Miscellaneous Issues (cont'd)

- ◆ **Security:** Since beans run in the same address space as their container, the same security restrictions apply. Never assume a specific trust level for your bean. 
- ◆ **Multi-threading:** Always suppose, your bean is running in a multi-threaded environment when you design its properties, methods and event-handling. 
- ◆ **Constructors:** Must provide a default constructor.



Bean Events

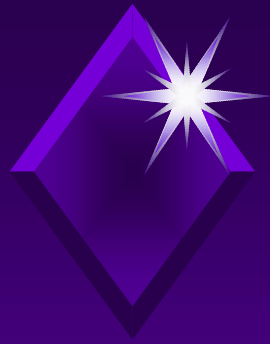
- ◆ An entity can use a bean by accessing its methods and properties.
- ◆ On the other hand, beans may need to notify users when something interesting has happened.
- ◆ Thus, communication between users and beans is not constrained to one direction.
- ◆ JavaBeans uses the delegation-based Event Model introduced with JDK 1.1.



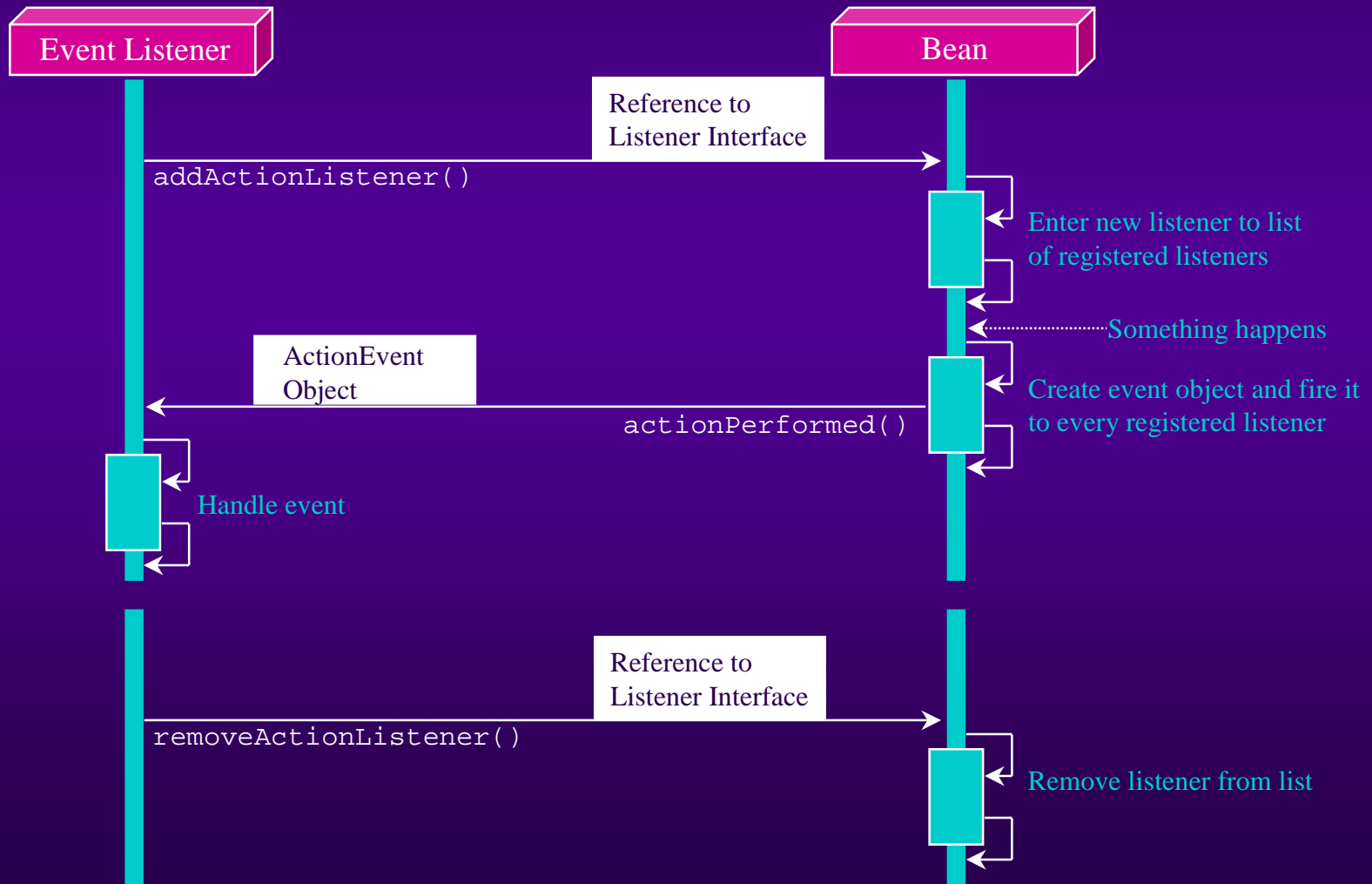


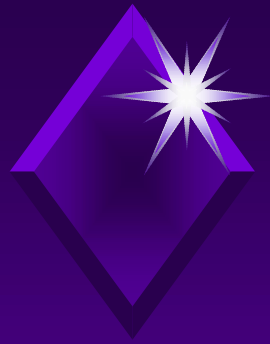
Bean Events (cont'd)

- ◆ Overview of delegation-based event handling:
 - ◆ Beans are **event sources** that generate events.
 - ◆ **Event listeners** want to get notified about these events.
 - ◆ Event listeners must **register** with event sources.
 - ◆ On registration they pass references to their **event handling interfaces** to the event sources.
 - ◆ Event sources create **event objects** and pass them as arguments to the event handling methods of listeners.
 - ◆ For each kind of event there is **one event handler**.



Bean Events (cont'd)





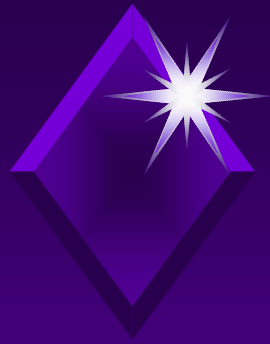
Bean Events (cont'd)

◆ Event Objects:

◆ ... describe events that are passed to the listeners.

◆ Example:

```
public class CellChangedEvent extends java.util.EventObject {
    private String position;
    CellChangedEvent(Spreadsheet source, String position) {
        super(source);
        this.position = position;
    }
    // Event objects should be considered immutable:
    public String getPosition() { return position; }
}
```



Bean Events (cont'd)

◆ Event Listener Interfaces:

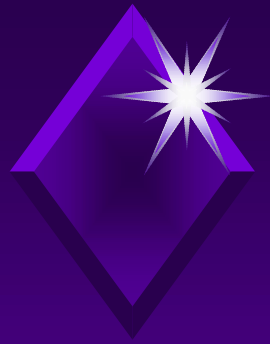
◆ ... describe event handling functionality of listeners.

◆ Example:

```
interface CellChangeListener extends java.util.EventListener
{
    void cellChanged(CellChangedEvent cce);
}

class MyApplication implements CellChangeListener {
    public void cellChanged(CellChangedEvent cce) {
        // handle event ...
    }
}

// ...
```



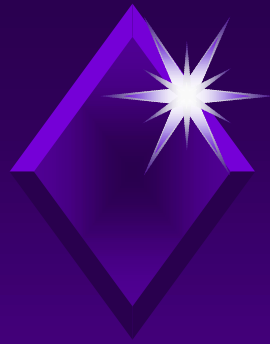
Bean Events (cont'd)

◆ Implementation of Registration:

◆ Beans implement methods for registration of listeners.

◆ Example:

```
public class Spreadsheet {
    private Vector listeners = new Vector();
    public synchronized void
    addCellChangeListener(CellChangeListener ccl) {
        listeners.addElement(ccl);
    }
    public synchronized void
    removeCellChangeListener(CellChangeListener ccl) {
        listeners.removeElement(ccl);
    }
}
```



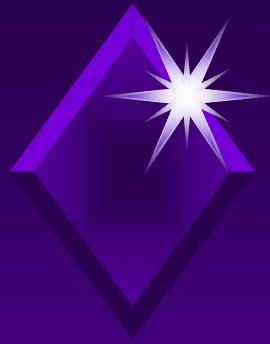
Bean Events (cont'd)

◆ Firing of events:

◆ Beans fire events to all registered listeners.

◆ Example:

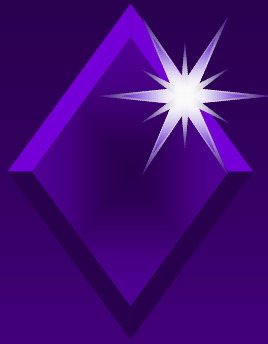
```
public class SpreadSheet {
    protected void fireCellChangedEvent() {
        Vector t;
        CellChangedEvent cce = new CellChangedEvent(this, pos);
        synchronized(this){ t = (Vector) listeners.clone(); }
        for (int i = 0; i < t.size(); i++) {
            (CellChangeListener)t.elementAt(i).cellChanged(cce);
        }
    }
}
// ....
```



Bean Events (cont'd)

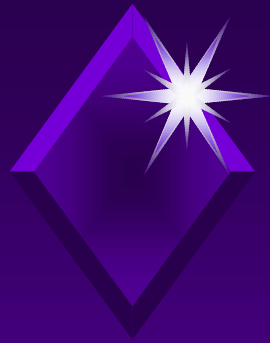
◆ Semantics of Event Delivery:

- ◆ Event delivery normally is multicast. For unicast delivery `addMyEventListener`-method must throw `java.util.TooManyListenersException`.
- ◆ Events are delivered **synchronously** w.r.t. the source.
- ◆ In multi-threaded environments deadlocks may occur. Thus, **use synchronization** where necessary.
- ◆ **Listener methods may throw exceptions.**
- ◆ A listener might **remove** itself **during event delivery.**



Event Adapters

- ◆ Suppose, you are using a builder tool for visibly connecting beans.
- ◆ Bean A implements an animation.
- ◆ Bean B is a button bean. It fires an **ActionEvent** whenever the user clicks on it.
- ◆ A should be started whenever B is clicked.
- ◆ How can the builder tool wire A and B together although they were developed independently?



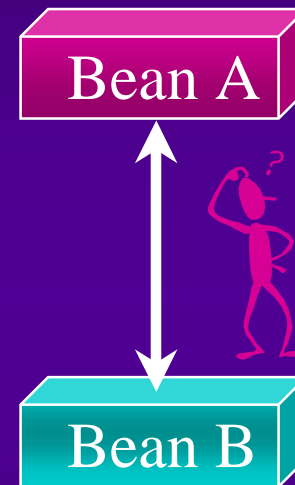
Event Adapters (cont'd)

◆ Here is bean A:

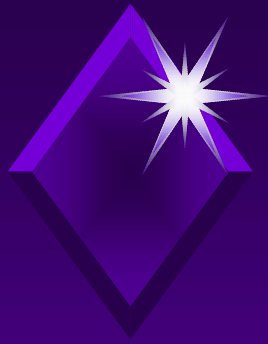
```
public class Animation ... {  
    void play() { .... }  
    // ....  
}
```

◆ Here is bean B:

```
public class MyButton extends java.awt.Canvas ... {  
    private Vector listeners = new Vector();  
    public void addActionListener(ActionListener al) { ... }  
    public void removeActionListener(ActionListener al) { ... }  
    protected void fireActionEvent() { ... } // fire event  
    ...  
}
```



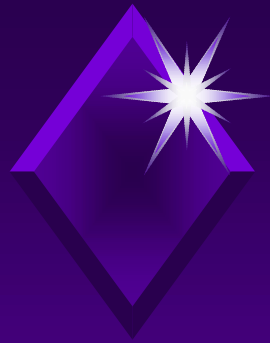
When the button is clicked, the animation should be played.



Event Adapters (cont'd)

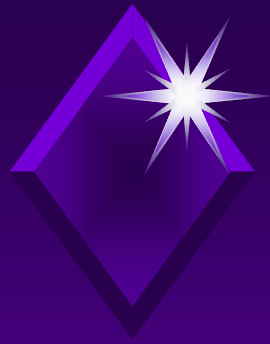
- ◆ Since A and B do not fit together, the builder tool generates an intermediate layer (adapter):

```
public class SimpleEventAdaptor implements ActionListener {
    protected Animation anm;
    protected MyButton btn;
    public void actionPerformed(ActionEvent ae) {
        anm.play(); // forward event
    }
    public SimpleEventAdaptor(Animation anm, MyButton btn) {
        this.anm = anm; this.btn = btn;
        btn.addActionListener(this);
    }
}
```



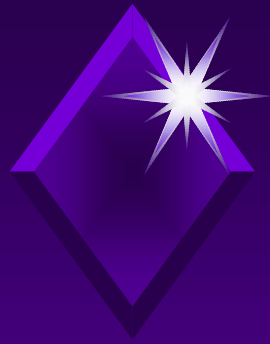
Event Adapters (cont'd)

- ◆ Some final remarks on event adapters:
 - ◆ To prevent an inflation of globally visible event handler classes, inner classes were introduced in JDK 1.1.
 - ◆ Generic demultiplexing adapter classes might be provided for reducing the amount of event handlers.
 - ◆ Event Adapters might also be used for:
 - ◆ Event queuing
 - ◆ Implementation of filters
 - ◆ Demultiplexing of event sources onto a single listener



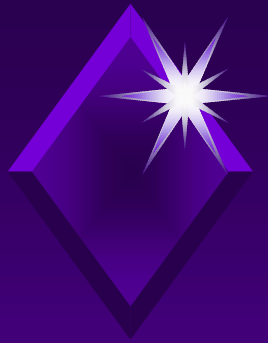
Bean Methods

- ◆ All *public* methods of a bean's class (and of its parent classes) are exported to bean users.
Example: the method `boolean contains(Point)` of an AWT widget.
- ◆ Through mechanisms of the Introspection API a developer might hide some of these public methods from users.



Bean Properties

- ◆ Properties are used in different contexts:
 - ◆ They may be accessed in **scripting environments**:
`okButton.size = 200;`
 - ◆ They can be modified **programmatically** using setter/getter methods: `myBean.setColor(Color.red)`.
 - ◆ They may be **customized** in a builder tool through property sheets or wizards.
 - ◆ Some of them are stored into/retrieved from **persistent storage**.

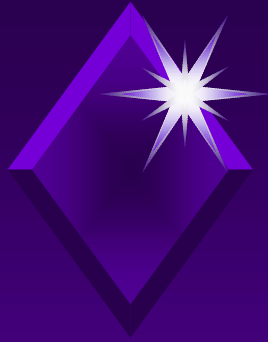


Bean Properties (cont'd)

- ◆ A Bean Property is implemented as a pair of setter/getter-methods, e.g., the property `capacity`:

```
private float capacity; // internal representation
public float getCapacity() { // get property
    return capacity;
}
public void setCapacity(float newval) { // set property
    capacity = newval;
}
```

- ◆ The setter/getter method might be left out for defining read-only/write-only properties.



Bean Properties (cont'd)

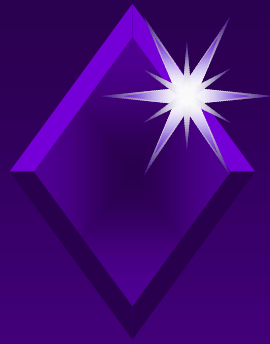
- ◆ The naming conventions for defining properties are as follows:

- ◆ getter method:

```
<PropType> get<PropName>()
```

- ◆ setter method:

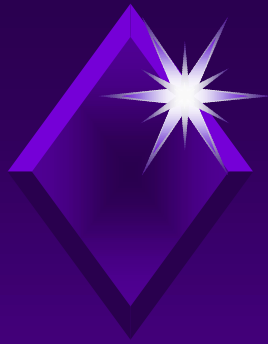
```
void set<PropName>(<PropType> value)
```



Indexed Properties

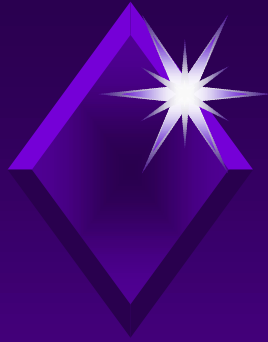
- ◆ Some properties might also represent whole sequences of values, e.g.:

```
private String names[];  
void setNames(int index, String newName);  
String getNames(int index);  
void setNames(String newval[]);  
String[] getNames();
```



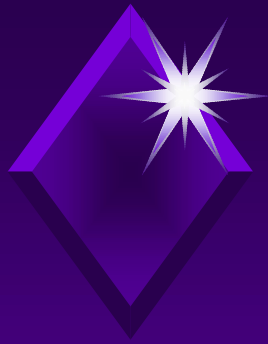
Bound Properties

- ◆ Whenever the state of a **bound property** changes, all registered listeners are notified. Thus, external behavior can be *bound* to property changes.
- ◆ For this purpose additional registration methods are required:
 - ◆ **public void**
addPropertyChangeListener(PropertyChangeListener x);
 - ◆ **public void**
removePropertyChangeListener(PropertyChangeListener x);



Bound Properties (cont'd)

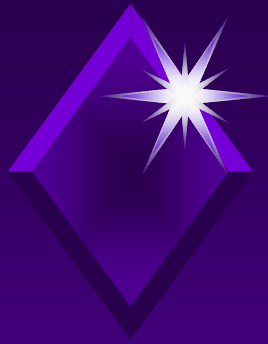
- ◆ For programming convenience the helper class `PropertyChangeSupport` is available.
- ◆ Note, that `PropertyChange` events are always fired **after** the change has been completed.



Bound Properties (cont'd)

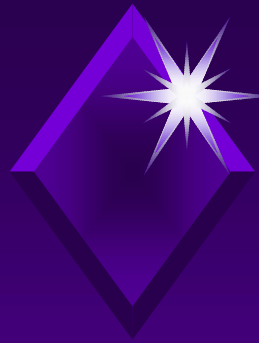
◆ Example:

```
private PropertyChangeSupport changes = new
    PropertyChangeSupport(this);
public void addPropertyChangeListener(PropertyChangeListener
    x) {
    changes.addPropertyChangeListener(x);
}
public void setColor(Color newval) {
    Color tmp = color;
    color = newval;
    changes.firePropertyChange("color", tmp, color);
}
```



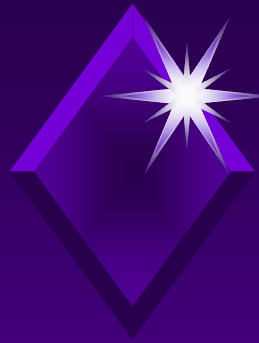
Constrained Properties

- ◆ Sometimes users of a beans want to be notified **before a property is changed** in order to reject inappropriate changes.
- ◆ Properties that support this kind of behavior are called **Constrained Properties**.
- ◆ Beans propagate special events to listeners before changing constrained properties. The change is only applied when there is no veto from any listener.



Constrained Properties *(cont'd)*

- ◆ The following methods for registering and removing listeners are provided by the bean:
 - ◆ **public void**
addVetoableChangeListener(PropertyChangeListener x);
 - ◆ **public void**
removeVetoableChangeListener(PropertyChangeListener x);
- ◆ For programming convenience the helper class `VetoableChangeSupport` is supplied.



Constrained Properties

(cont'd)

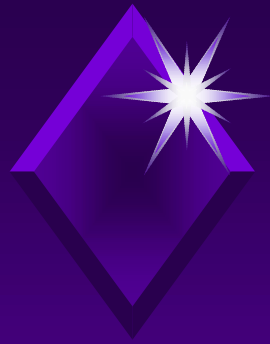
◆ Example:

```
private VetoableChangeSupport vetos = new
    VetoableChangeSupport(this);
public void addVetoableChangeListener(VetoableChangeListener
    x) {
    vetos.addVetoableChangeListener(x);
}
public void setSpeed(float newval) {
    try {
        vetos.fireVetoableChange("speed", speed, newval);
        speed = newval; // no veto was thrown
    } catch (PropertyVetoException pve) { /* do nothing */ }
}
```



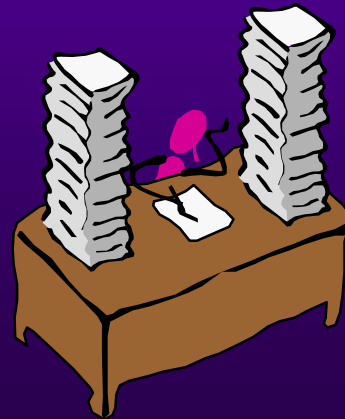
Persistence

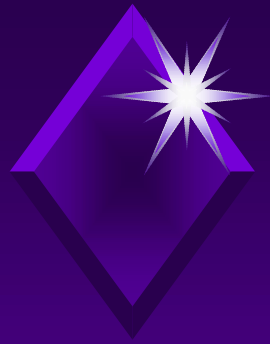
- ◆ We need mechanisms to make the state of a bean persistent.
- ◆ Complex beans should be able to store themselves using an appropriate data format.
- ◆ Simple beans should automatically be stored in a platform-independent format.
- ◆ For this purpose, JavaBeans provides a composite solution for persistence:



Persistence (cont'd)

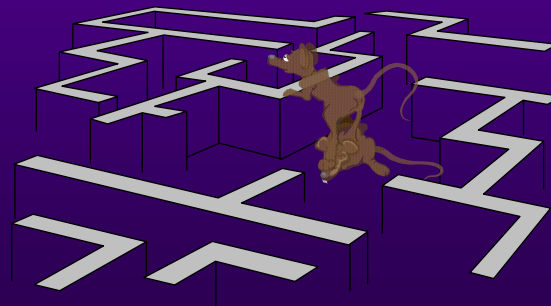
- ◆ **Object Serialization** allows builder tools to automatically store the non-transient members of beans. Classes of data members must implement the empty marker interface **Serializable**.
- ◆ **Externalization** gives the bean complete control of how it is stored.

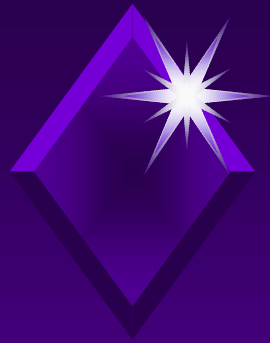




Introspection

- ◆ Suppose, you are developing a builder tool with JavaBeans support.
- ◆ The user has installed a new collection of beans.
- ◆ Question: How can the builder tool dynamically find out and use the features of an arbitrary bean?

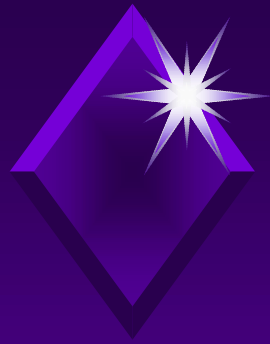




Introspection (cont'd)

- ◆ Solution 1: Let the developer explicitly specify the information in a description file.
- ◆ **Solution 2:** Usage of the Reflection API of Java 1.1 to dynamically analyze and use
 - ◆ Methods,
 - ◆ Events,
 - ◆ Properties.
- ◆ In JavaBeans the second approach is supported by implementing a special Introspection API.





Introspection (cont'd)

- ◆ Thus, introspection comes at no cost with one small exception: users must conform to some notational patterns.
- ◆ For instance, set/get-methods of properties:

```
// Defining a property named "speed":
```

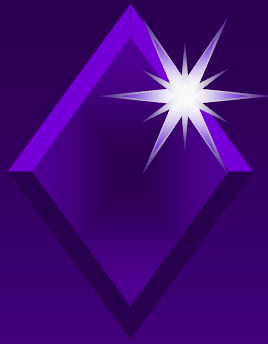
```
private float spd; // name does not matter
```

```
public float getSpeed() { return spd; }
```

```
public void setSpeed(float speed) { this.spd = speed; }
```

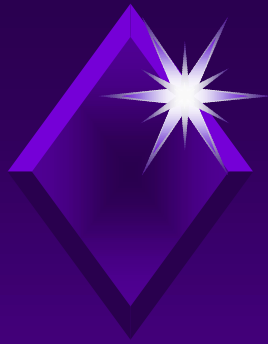
```
// The Introspector derives from this definitions that
```

```
// there is a Read/Write property speed.
```



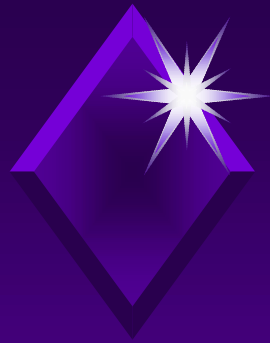
Introspection (cont'd)

- ◆ Sometimes, developers want to specify the bean information explicitly (e.g., to enforce their own notational conventions).
- ◆ For this purpose, JavaBeans allows to provide a specific **BeanInfo** class to list the events, methods, and properties of a bean.
- ◆ Both approaches might be combined.



Introspection (cont'd)

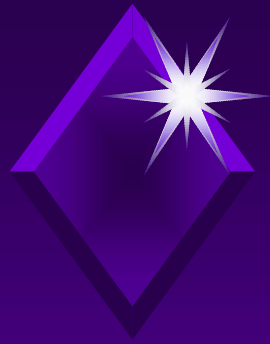
- ◆ An **Introspector** traverses the bean class and its ancestors.
- ◆ If for a class `X` a class `XBeanInfo` is available, this class is used to retrieve the bean information.
- ◆ Otherwise, the Reflection API is used to find out whether the available class members match with the notational conventions. Events, methods, and properties can be determined automatically.



Introspection (cont'd)

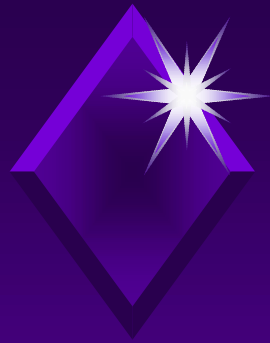
- ◆ The Interface BeanInfo must be implemented by a BeanInfo class:

```
interface BeanInfo {  
    public final static int ICON_COLOR_16x16; // ...  
    public abstract BeanInfo[] getAdditionalBeanInfo();  
    public abstract BeanDescriptor getBeanDescriptor();  
    public abstract int getDefaultEventIndex();  
    public abstract int getDefaultPropertyIndex();  
    public abstract EventSetDescriptor getEventSetDescriptors();  
    public abstract getImage(int iconKind);  
    public abstract MethodDescriptor getMethodDescriptors();  
    public abstract PropertyDescriptor getPropertyDescriptors();  
}
```



Introspection (cont'd)

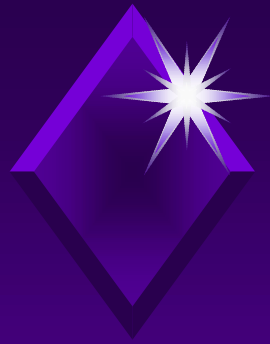
- ◆ For programmer's convenience a utility class `SimpleBeanInfo` is available.
- ◆ This class provides “no-op” implementations for all methods of the `BeanInfo` interface.
- ◆ In addition, a support method `loadImage` is implemented.
- ◆ Developers derive their own `MyBeanInfo` class from `SimpleBeanInfo` and implement some methods.



Introspection (cont'd)

◆ Example:

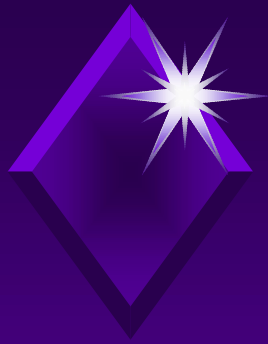
```
public class AnimationBeanInfo extends SimpleBeanInfo {
    public Image getIcon(int iconKind) {
        if (iconKind == BeanInfo.ICON_COLOR_32x32)
            return loadImage("MyIcon32.gif");
        return null;
    }
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor rate =
                new PropertyDescriptor("rate", Animation.class);
            PropertyDescriptor propArray[] = { rate };
            return propArray;
        } catch (IntrospectionException ie) {}
    }
}
```



Customization

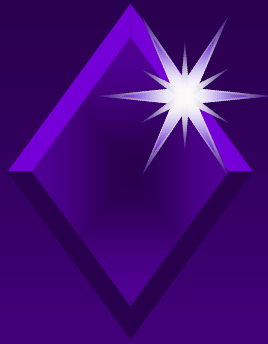
- ◆ When composing applications, users need to customize (the properties of) integrated beans.
- ◆ JavaBeans offers two different ways for customization:
 - ◆ Builder tools display a **property sheet**. For each property a separate **property editor** is loaded.
 - ◆ Developers may implement a **customizer** class - a GUI-based wizard application that guides the user through property customization.





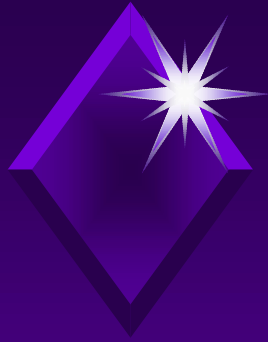
Customization (cont'd)

- ◆ **Property Editors:**
 - ◆ For each type of property a corresponding `PropertyEditor` class must be available.
 - ◆ It allows to modify bean properties within a builder.
 - ◆ A `PropertyEditorManager` maps between Java types and their corresponding property editor classes.
 - ◆ Editors must register with this manager which is pre-loaded with editors for standard Java types such as `int`.
 - ◆ Editors should fire `PropertyChange` events.



Customization (cont'd)

- ◆ Customizers:
 - ◆ For complex beans property editors are not sufficient.
 - ◆ A special **customizer** class may be implemented to automatically guide the user through customization.
 - ◆ A customizer should directly/indirectly inherit from `java.awt.Component`.
 - ◆ If you want to supply a customizer class, you need to provide a `BeanInfo` class too.



Customization (cont'd)

◆ Example for a User-defined Property Editor:

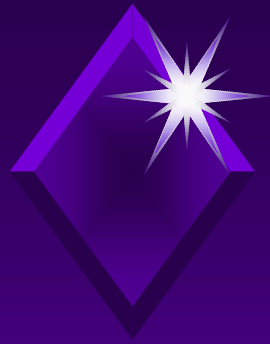
```
public class YesNoEditor extends PropertyEditorSupport {
    public String getJavaInitializationString() {
        if ((YesNo)getValue()).equals(YesNo.Yes) return "YesNo.Yes";
        else return "YesNo.No";
    }
    public String getAsText() {
        if ((YesNo)getValue()).equals(YesNo.Yes) return "Yes";
        else return "No";
    }
    // ... some stuff left out for brevity
    public String[] getTags() {
        String tags[] = { "Yes", "No" }; return tags;
    }
}
```



Packaging

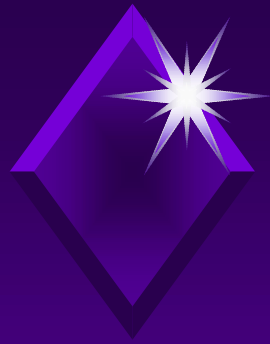
- ◆ Beans are supplied to programmers using JAR files.
- ◆ JAR files consist of:
 - ◆ class files,
 - ◆ resource files (sound, video, images),
 - ◆ help files,
 - ◆ serialized objects,
 - ◆ and an optional Manifest file.





Packaging (cont'd)

- ◆ The entries in a JAR file are associated with hierarchical names such as “`DE/Siemens/ZT/ust/x.ser`”
- ◆ Optional manifest files may be used to describe the contents of the JAR file.
- ◆ Manifest files must be named “`META-INF/MANIFEST.MF`”.
- ◆ They are structured as a sequence of sections where each section contains one or more headers.



Packaging (cont'd)

◆ Example for a Manifest file:

```
Name: DE/Siemens/ZT/animation/Animation.class  
Java-Bean: True
```

```
Name: DE/Siemens/ZT/mybutton/MyButton.class  
Java-Bean: True  
Tested: True
```



A Sample Bean

- ◆ Sample Bean *OurButton* from the JavaSoft Bean Developer Kit (Version 1.0, 6/97)





A Sample Bean (cont'd)

- ◆ We want to build a very flexible button bean `OurButton`.
- ◆ The bean should be customizable by the user through a integrated customizer class.
- ◆ The button should be implemented as a “light-weight” component.

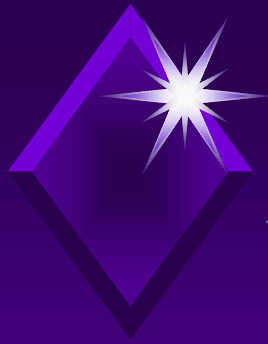


A Sample Bean (cont'd)

- ◆ The bean is directly derived from `java.awt.Component`:

```
public class OurButton extends Component
    implements      Serializable,
                   MouseListener,
                   MouseMotionListener {

    // ...
}
```



A Sample Bean (cont'd)

◆ Properties of the Bean:

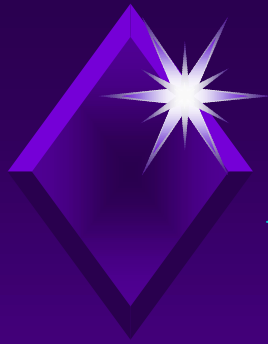
- ◆ **String** label text to appear on the button
- ◆ **Font** font font to be used
- ◆ **Boolean** largeFont large size yes/no
- ◆ **int** FontSize size of font
- ◆ **Boolean** debug debug output yes/no
- ◆ **Color** foreground foreground color
- ◆ **Color** background background color
- ◆ All properties are readable and writeable



A Sample Bean (cont'd)

- ◆ All properties are designed as bound properties.
- ◆ Example Debug:

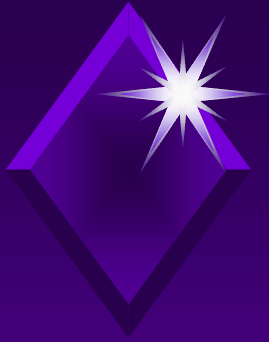
```
private boolean debug;  
public boolean getDebug() {  
    return debug;  
}  
public void setDebug(boolean x) {  
    boolean old = debug;  
    debug = x;  
    changes.firePropertyChange("debug", new Boolean(old), new  
Boolean(x));  
}
```



A Sample Bean (cont'd)

- ◆ Implementing code for registering property change listeners and firing events:

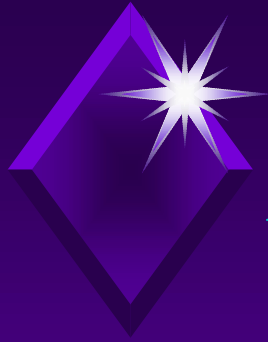
```
private PropertyChangeSupport changes
    = new PropertyChangeSupport(this);
public void
addPropertyChangeListener(PropertyChangeListener l) {
    changes.addPropertyChangeListener(l);
}
public void
removePropertyChangeListener(PropertyChangeListener l) {
    changes.removePropertyChangeListener(l);
}
```



A Sample Bean (cont'd)

◆ The paint method of the bean:

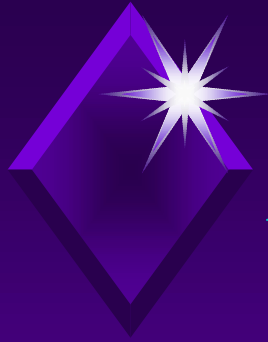
```
public synchronized void paint(Graphics g) {
    int width = getSize().width;
    int height = getSize().height;
    g.setColor(setBackground());
    g.fill3DRect(0, 0, width - 1, height - 1, !down);
    g.setColor(getForeground());
    g.setFont(getFont());
    g.drawRect(2, 2, width - 4, height - 4);
    FontMetrics fm = g.getFontMetrics();
    g.drawString(label, (width - fm.stringWidth(label)) / 2,
        (height + fm.getMaxAscent() - fm.getMaxDescent()) / 2);
}
```



A Sample Bean (cont'd)

- ◆ **Events:** Our bean is firing an `ActionEvent` whenever a mouse button is clicked and released while the button is enabled.

```
public void mouseReleased(MouseEvent evt) {  
    if (!isEnabled()) return; // if the button is disabled  
    if (down) { // if the mouse was pressed and now released  
        fireAction(); // fire ActionEvent  
    }  
}
```



A Sample Bean (cont'd)

◆ Implementing the bean as an `ActionEvent` source:

```
private Vector pushListeners = new Vector();
```

```
public synchronized
```

```
void addActionListener(ActionListener l) {  
    pushListeners.addElement(l);  
}
```

```
public synchronized
```

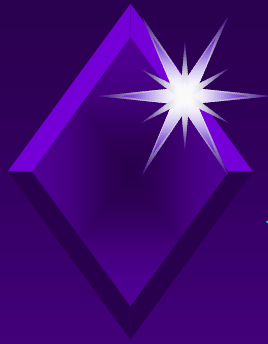
```
void removeActionListener(ActionListener l) {  
    pushListeners.removeElement(l);  
}
```



A Sample Bean (cont'd)

◆ Firing an action event:

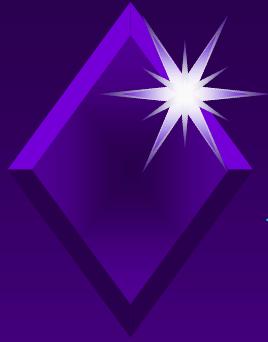
```
public void fireAction() {
    if (debug) System.err.println("Button " + getLabel() +
        "pressed.");
    Vector targets;
    synchronized(this) {
        targets = (Vector) pushListeners.clone();
    }
    ActionEvent actionEvt = new ActionEvent(this, 0, null);
    for (int i = 0; i < targets.size(); i++) {
        ActionListener t = (ActionListener)targets.elementAt(i);
        t.actionPerformed(actionEvt); } // end for-loop
}
```



A Sample Bean (cont'd)

◆ Providing a Customizer class:

```
public class OurButtonCustomizer extends Panel
    implements Customizer, KeyListener {
    private OutButton target;
    private TextField labelField;
    public OurButtonCustomizer() { setLayout(null); }
    public void setObject(Object obj) {
        target = (OurButton) obj;
        Label t1 = new Label("Caption:", Label.RIGHT);
        add(t1); t1.setBounds(10, 5, 60, 30);
        labelField = new TextField(target.getLabel(), 20);
        add(labelField); labelField.setBounds(80, 5, 100, 30);
        labelField.addKeyListener(this);
    } // to be continued ...
```



A Sample Bean (cont'd)

◆ Registering listeners for property changes:

```
private PropertyChangeSupport support =  
    new PropertyChangeSupport(this);
```

```
public void  
addPropertyChangeListener(PropertyChangeListener l) {  
    support.addPropertyChangeListener(l);  
}
```

```
public void  
removePropertyChangeListener(PropertyChangeListener l) {  
    support.removePropertyChangeListener(l);  
}
```



A Sample Bean (cont'd)

◆ Implementing the event handling methods:

```
public void keyTyped(KeyEvent e) {} // ignore
```

```
public void keyPressed(KeyEvent e) {} // ignore
```

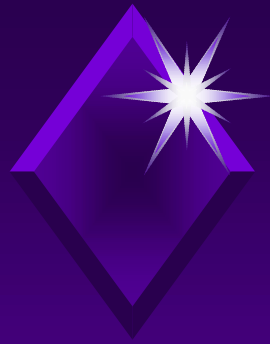
```
public void keyReleased(KeyEvent e) {  
    String txt = labelField.getText();  
    target.setLabel(txt);  
    support.firePropetyChange("", null, null);  
}
```



A Sample Bean (cont'd)

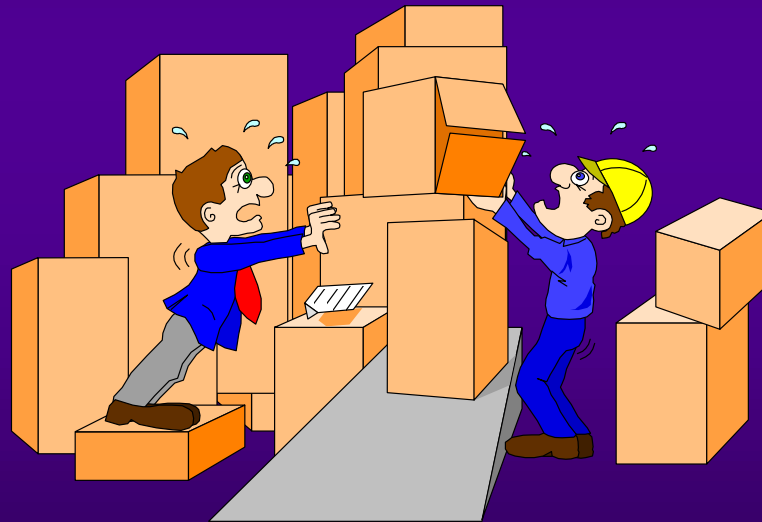
- ◆ That's it! You can browse the full example code in `<jdk-directory>/demo/sunw/demo/buttons`

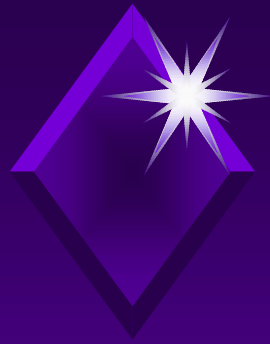




BDK - Beans Developer Kit

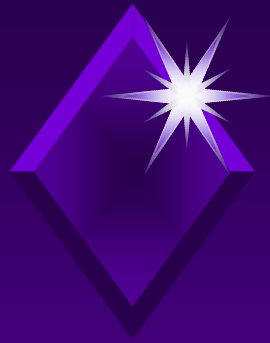
- ◆ All you need are tools





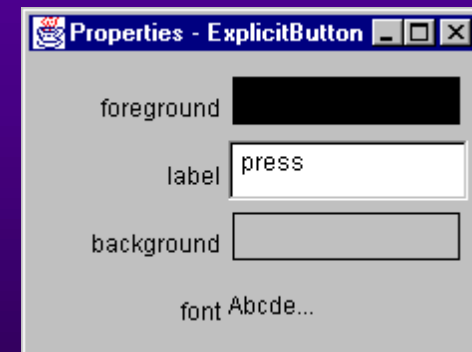
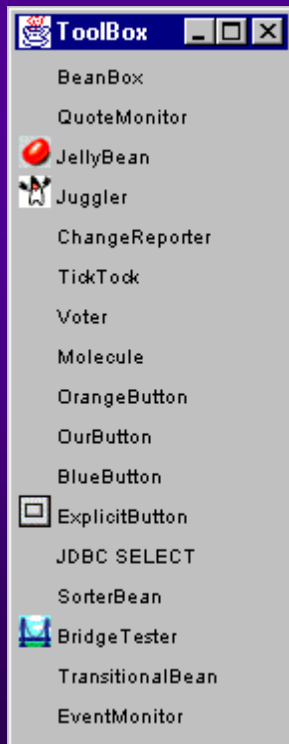
BDK (cont'd)

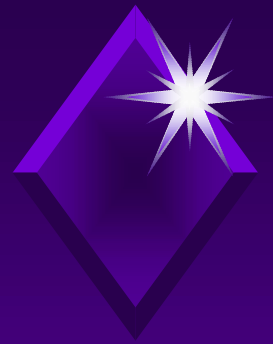
- ◆ The BDK is intended to
 - ◆ introduce JavaBeans 1.0
 - ◆ demonstrate how to develop builders tools
 - ◆ provide an example container to support developers
- ◆ It includes:
 - ◆ **BeanBox**: A sample Bean container with source code
 - ◆ Examples, Documents
 - ◆ Demonstration of 3rd Party tools
- ◆ It is *not* a programming tool



BDK (cont'd)

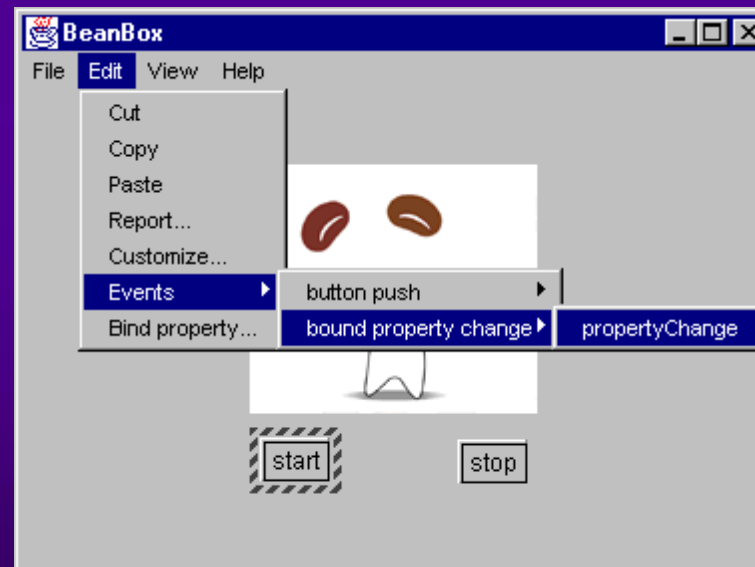
◆ BeanBox: Arranging & Customizing Beans

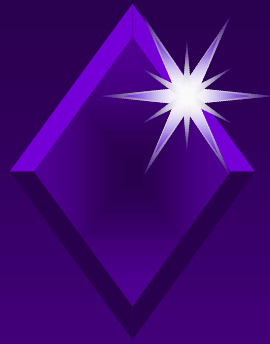




BDK (cont'd)

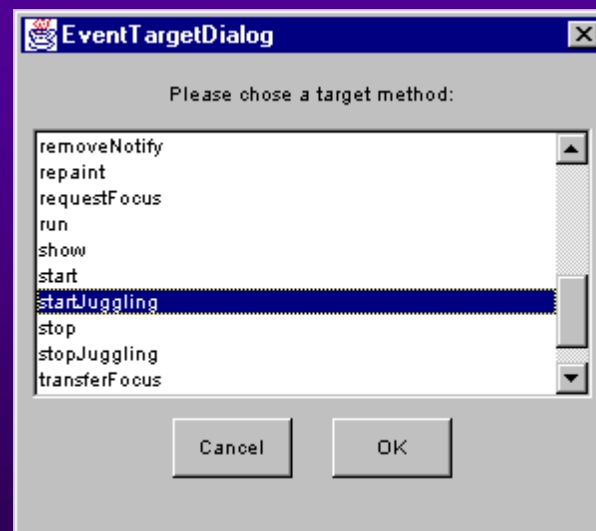
- ◆ Events from a bean may be selected ...

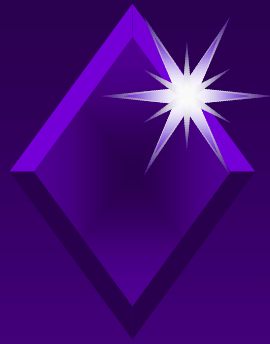




BDK (cont'd)

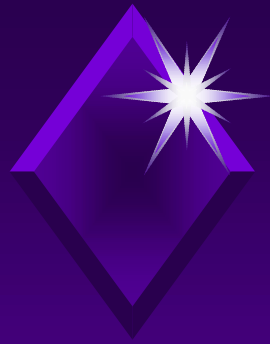
- ◆ ... and connected to a listener method (as a result the builder generates an adapter class).





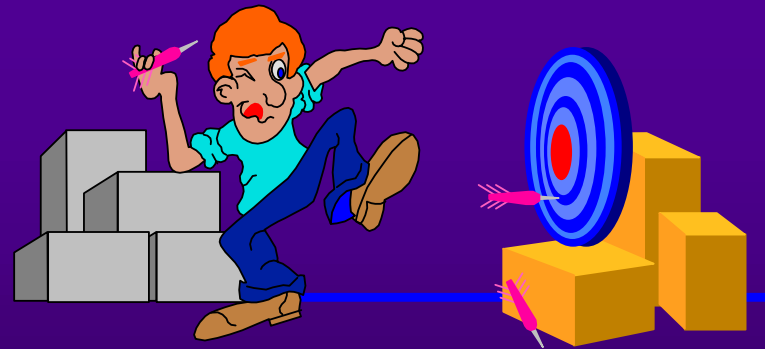
Other Tools

- ◆ The BeanBox is not intended to be a full-fledged environment for visual programming.
- ◆ Full programming support is available:
 - ◆ Borland JBuilder
 - ◆ IBM VisualAge for Java
 - ◆ Lotus BeanMachine
 - ◆ SunSoft Visual Studio, Visual Workshop
 - ◆ Symantec Visual Cafe
 - ◆ ... and countless other products.



JavaBeans and ActiveX Controls

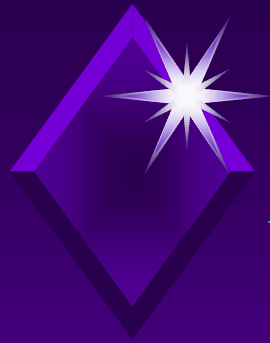
- ◆ JavaBeans is competing with ActiveX Controls





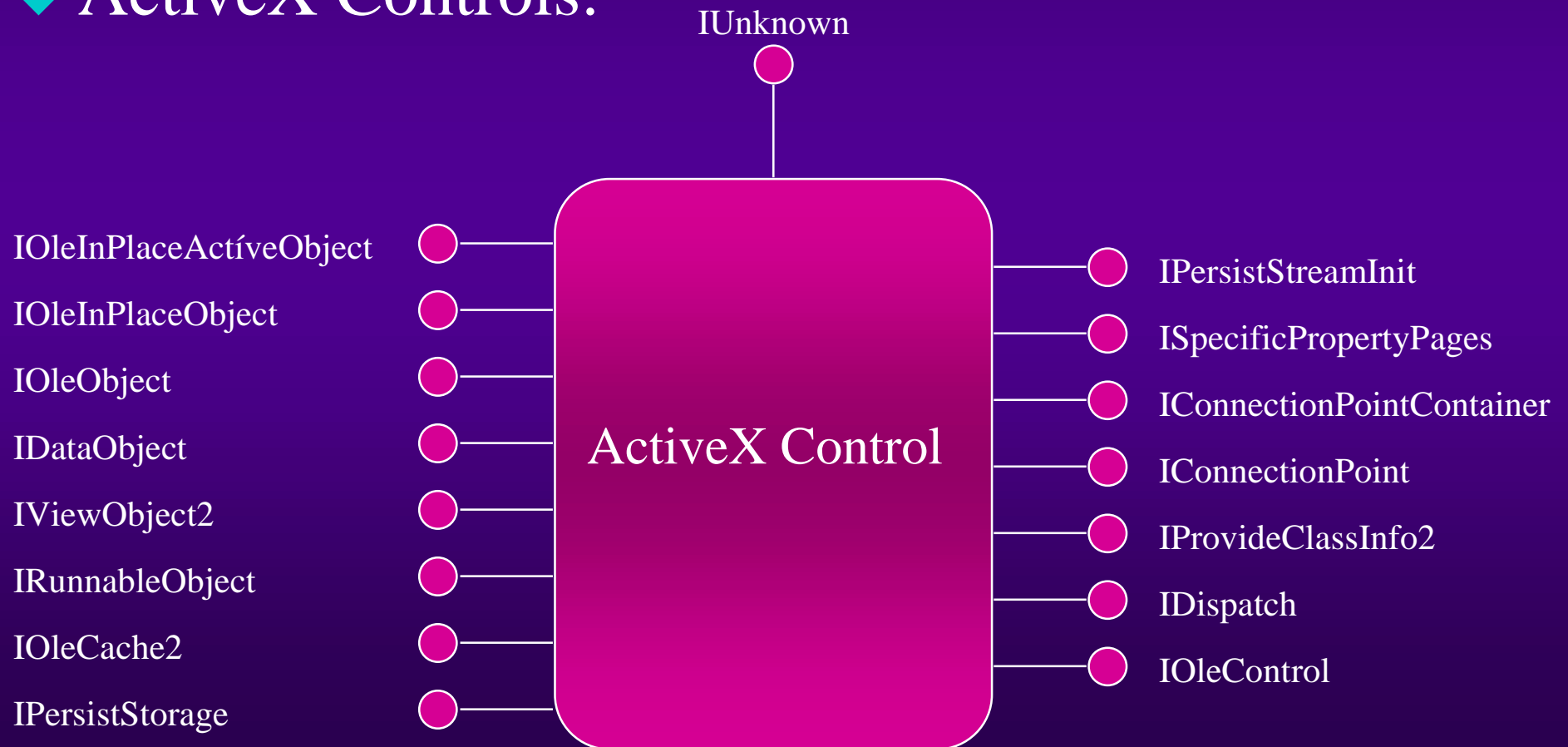
ActiveX Controls

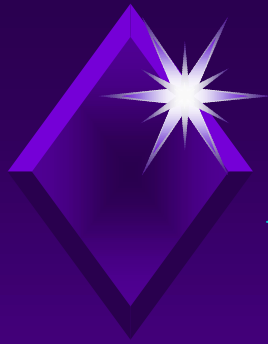
- ◆ ActiveX Controls are **components based on Microsoft COM** (Component Object Model).
- ◆ Functionality is imported/exported by **interfaces**.
- ◆ A COM object may implement **multiple interfaces**.
- ◆ Each interface is derived from **IUnknown**.
- ◆ Interfaces are specified using **MIDL**.
- ◆ Type Information is provided by **Type-Libraries**.



ActiveX (cont'd)

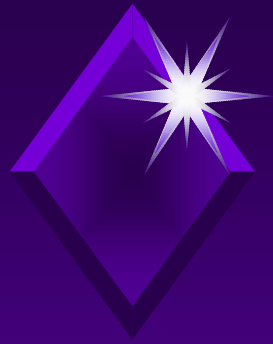
◆ ActiveX Controls:





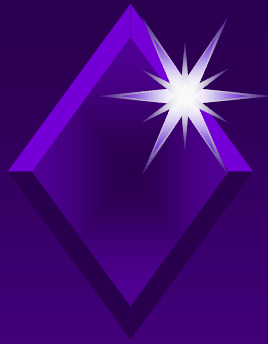
ActiveX Controls (cont'd)

- ◆ ActiveX Controls are COM objects that implement a predefined set of interfaces.
- ◆ COM objects may be implemented/used in **any programming language**.
- ◆ However, they are supplied in binary form and therefore **bound to a native platform**.



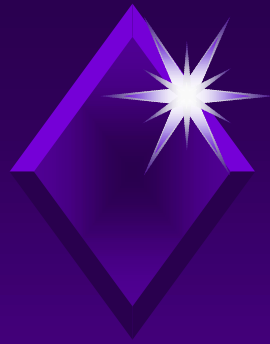
Comparison

| Feature | JavaBeans | ActiveX Controls |
|----------------------|-------------------------------|-----------------------------|
| <i>Platforms</i> | Each Java-enabled OS | Windows {9x, NT} |
| <i>Language</i> | Java only | arbitrary |
| <i>Introspection</i> | Automatic or explicit in Java | MIDL (Type Libraries) |
| <i>Events</i> | Java Events | Connection Points |
| <i>Life Cycle</i> | Garbage Collection | Manually w. AddrOf, Release |
| <i>Persistence</i> | Serialization | Persist-Interfaces |
| <i>Packaging</i> | JAR File | Dynamic Link Libraries |



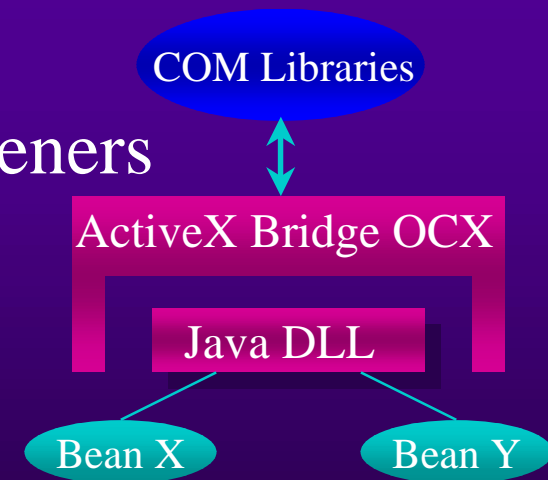
Comparison (cont'd)

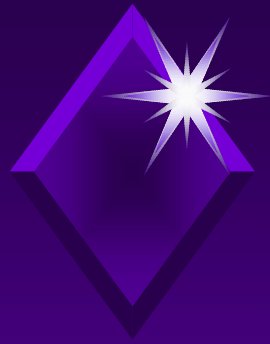
| Feature | JavaBeans | ActiveX Controls |
|-----------------------------------|-----------------------|--|
| <i>Licencing</i> | Not supported | Supported: |
| <i>Customization</i> | Property Pages | IClassFactory2 Property Editors, Customizers |
| <i>Data Exchange</i> | Uniform Data Transfer | Uniform Data Transfer |
| <i>Compound Doc.</i> | Not yet supported | Fully supported |
| <i>Aggregation</i> | Not yet supported | Fully supported |
| <i>3rd Pty Support</i> | Yet to come ... | Very good |



From JavaBeans to ActiveX

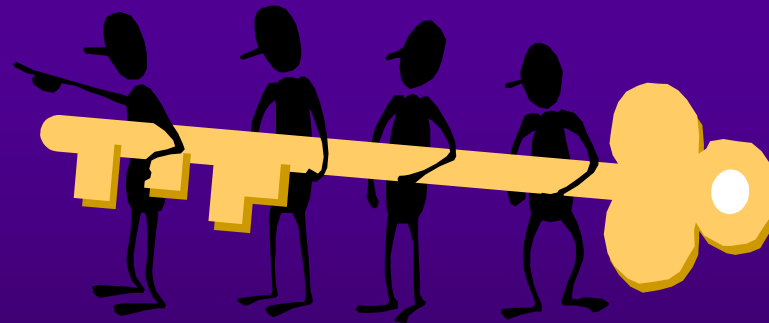
- ◆ A Bean is introspected by a bridge tool.
- ◆ The following files are generated:
 - ◆ A Registry file
 - ◆ A Type Library
 - ◆ Stub files for registering as event listeners
 - ◆ A Setup file
- ◆ OLE persistence is supported
- ◆ Automation is also supported

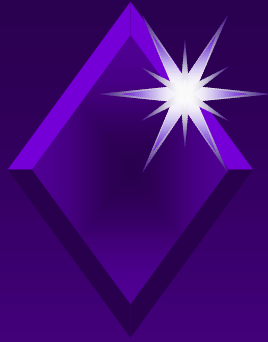




Related Technologies

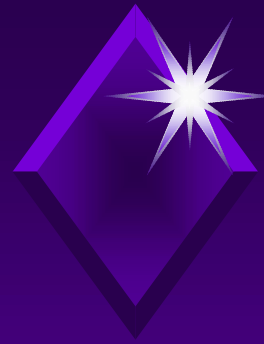
- ◆ Other technologies are the key for adding even more value to JavaBeans





Related Technologies (cont'd)

- ◆ Beans might use **Java JDBC** to access database systems.
- ◆ **RMI** helps to distribute component functionality.
- ◆ Java + **CORBA IDL** allows to provide components for heterogeneous distributed systems. JavaBeans are now the Component standard of the OMG.
- ◆ **JFC & AWT** support the development of fancy beans and containers.



JavaBeans - Present and Future

- ◆ A look into the crystal ball





Summary

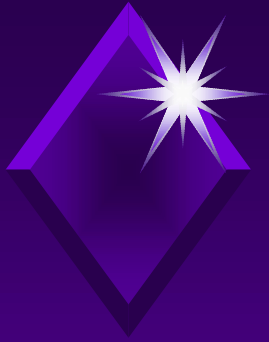
◆ Famous last words



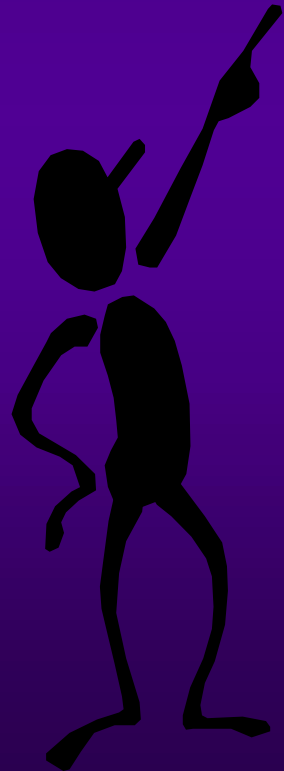


Summary

- ◆ JavaBeans is a promising, **easy-to-use** presentation tier technology for building **GUI elements and client-side controls**.
- ◆ Enterprise JavaBeans and JavaBeans are *different* technologies.
- ◆ JavaBeans allow to decompose the monolithic client tier.
- ◆ Tool support for JavaBeans is emerging rapidly.
- ◆ JavaBeans are useful in many other areas such as Java ServerPages.

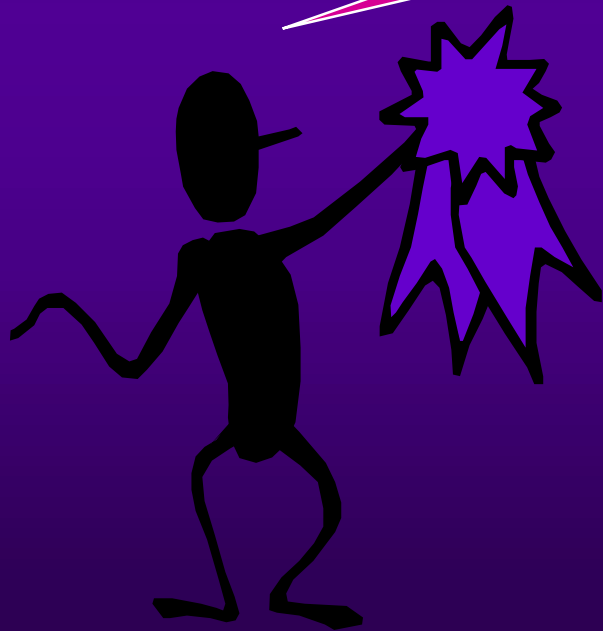


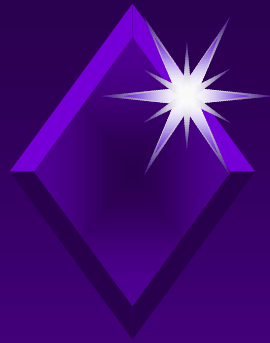
Any Questions ?





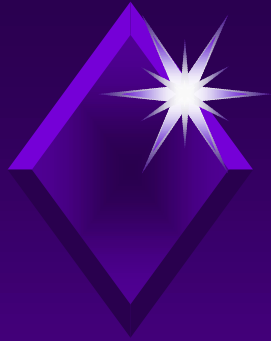
Thanks a lot for attending
this talk!!





References

- ◆ JavaSoft: **Java Beans 1.0 API specification**, December 4, 1996.
- ◆ Michael Morrison: **Presenting JavaBeans**, SamsNet, 1997.
- ◆ Alden DeSoto: **Using the Beans Development Kit 1.0 - A Tutorial**, JavaSoft, February 1997.
- ◆ Laurence Cable, Graham Hamilton: **A Draft Proposal to define an Extensible Runtime Containment and Services Protocol for JavaBeans**, JavaSoft, 1997.
- ◆ Laurence Cable, Graham Hamilton: **A Draft Proposal for a Object Aggregation/Delegation Model for Java and JavaBeans**, JavaSoft, 1997.
- ◆ Laurence Cable: **Proposal for a Drag and Drop subsystem for the Java Foundation Classes**, JavaSoft, 1997.
- ◆ Lotus, Sun Microsystems: **Infobus Specification**, 1997.
- ◆ Bart Calder, Bill Shannon: **JavaBeans Activation Framework Specification**, Sun Microsystems, 1997.



Internet References

- ◆ JavaSoft: <http://www.javasoft.com>
- ◆ Gamelan: <http://www.gamelan.com>
- ◆ JavaWorld Online: <http://www.javaworld.com>
- ◆ Inprise/Borland: <http://www.borland.com>
- ◆ Symantec: <http://www.symantec.com>
- ◆ Marimba: <http://www.marimba.com>
- ◆ Bean Homepage: <http://java.sun.com/products/ejb>