

UNIT 3

JavaBeans

(7 Hours) — Advanced Java Programming (CAC354)

Important Notice — Please Read

These notes were prepared from the syllabus and general knowledge of Java. **They may contain errors** (typos, outdated API details, or mistakes in code and explanations), so please take care. Verify important points and test all code yourself, and cross-check with your textbook (Core Java Vol. II / Java: The Complete Reference), the official Java documentation, and your teacher's notes.

3.1 What Is a Java Bean?

A **JavaBean** is a reusable software component written in Java that can be manipulated visually in a builder tool (IDE) and assembled into applications. Technically it is an ordinary Java class that follows some conventions:

- It has a **public no-argument constructor**.
- Its fields (*properties*) are **private** and accessed through public **getter and setter** methods following naming patterns.
- It should be **serializable** (implements `java.io.Serializable`) so its state can be saved and restored.
- It may fire **events** and allow listeners to register (e.g. `PropertyChangeListener`).

```
import java.io.Serializable;

public class Student implements Serializable {
    private String name;
    private int age;
    private boolean active;

    public Student() { } // no-arg constructor

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public boolean isActive() { return active; } // boolean getter uses "is"
    public void setActive(boolean a) { this.active = a; }
}
```

Note: Do not confuse JavaBeans with Enterprise JavaBeans (EJB), which are server-side components and a different technology.

3.2 Advantages of Java Beans

- **Reusability** — write once, use in many applications and tools.
- **Platform independence** — runs on any JVM.
- **Visual development** — builder tools can discover properties and let developers configure beans without code.
- **Encapsulation** — data hidden behind getters/setters, allowing validation.
- **Persistence** — state can be saved and restored with serialization.
- **Loose coupling** — beans communicate using events, so components need not know each other.
- Widely used in JSP (<jsp:useBean>), frameworks (Spring, JSF) and Swing components (every Swing component is a bean).

3.3 Introspection

Introspection is the process by which a builder tool (or program) discovers a bean's *properties, events and methods*. It is implemented by the `java.beans.Introspector` class, which uses **reflection** and **design patterns** (naming conventions), or an explicit `BeanInfo` class, to produce a `BeanInfo` object.

```
import java.beans.*;

BeanInfo info = Introspector.getBeanInfo(Student.class, Object.class);
for (PropertyDescriptor pd : info.getPropertyDescriptors())
    System.out.println(pd.getName() + " : " + pd.getPropertyType());
// age : int, active : boolean, name : class java.lang.String
```

Introspection works in two ways: **implicit** (low-level reflection + naming patterns) and **explicit** (the bean author supplies a `BeanInfo` class).

3.4 Properties, Events and Methods — Design Patterns

If the naming conventions below are followed, `Introspector` can find the features automatically.

3.4.1 Property Design Patterns

Kind	Pattern	Example
Simple property	<code>public T getX() / public void setX(T v)</code>	<code>getName() , setName(String)</code>
Boolean property	<code>public boolean isX() / setX(boolean)</code>	<code>isActive() , setActive(boolean)</code>

Kind	Pattern	Example
Read-only	Only getter	getId() with no setter
Write-only	Only setter	setPassword(String)
Indexed property	T getX(int i) / void setX(int i, T v) (and array forms)	getMarks(int), setMarks(int, int)

The property name is obtained by removing *get/set/is* and lower-casing the first letter:
`getFirstName()` → property `firstName`.

3.4.2 Event Design Patterns

A bean that fires events must provide methods to register and remove listeners:

```
public void addXxxListener(XxxListener l)
public void removeXxxListener(XxxListener l)
// e.g.
public void addActionListener(ActionListener l)
public void removeActionListener(ActionListener l)
```

The event class should extend `EventObject` and the listener interface should extend `EventListener`. For *unicast* events, the `add` method throws `java.util.TooManyListenersException`.

3.4.3 Method Design Patterns

All **public** methods of a bean that are not part of a property or event pattern are treated as exposed methods that other components (or tools) can call.

3.5 Using the BeanInfo Interface

If a bean's methods do not follow the patterns, or the author wants to hide some features or provide friendly names and icons, a class implementing `java.beans.BeanInfo` can be supplied. By convention it is named **BeanName + "BeanInfo"** and placed in the same package; it usually extends `SimpleBeanInfo`.

BeanInfo method	Purpose
<code>getPropertyDescriptors()</code>	Returns the properties to expose
<code>getEventSetDescriptors()</code>	Returns the events the bean fires
<code>getMethodDescriptors()</code>	Returns the methods to expose
<code>getBeanDescriptor()</code>	Name, display name, customizer class
<code>getIcon(int kind)</code>	Icon shown in the builder tool

```
import java.beans.*;

public class StudentBeanInfo extends SimpleBeanInfo {
    @Override
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor name = new PropertyDescriptor("name", Student.class);
            PropertyDescriptor age = new PropertyDescriptor("age", Student.class);
            name.setDisplayName("Student Name");
            return new PropertyDescriptor[] { name, age }; // "active" is hidden
        } catch (IntrospectionException e) { return null; }
    }
}
```

3.6 Bound and Constrained Properties

3.6.1 Bound Properties

A **bound property** notifies other objects *after* its value changes. Interested listeners register with the bean, and the bean fires a `PropertyChangeEvent` containing the property name, old value and new value. The helper class `PropertyChangeSupport` does most of the work.

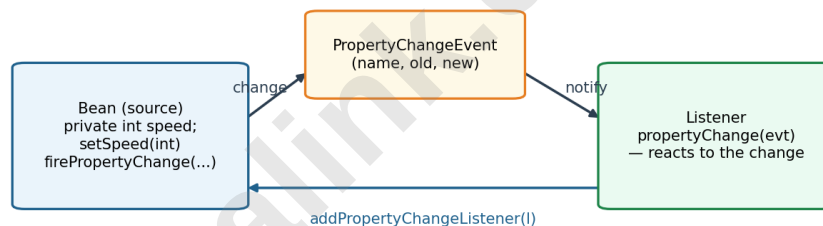


Figure 3.1 — Bound property: bean notifies listeners after change

```
import java.beans.*;

public class Car implements java.io.Serializable {
    private int speed;
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener l) { pcs.addPropertyChangeListener(l); }
    public void removePropertyChangeListener(PropertyChangeListener l) { pcs.removePropertyChangeListener(l); }

    public int getSpeed() { return speed; }
    public void setSpeed(int newSpeed) {
        int old = speed;
        speed = newSpeed;
        pcs.firePropertyChange("speed", old, newSpeed); // notify after change
    }
}

// listener
car.addPropertyChangeListener(evt -> {
    System.out.println(evt.getPropertyName() + ": " + evt.getOldValue() + " -> " + evt.getNewValue());
});
car.setSpeed(60); // prints speed: 0 -> 60
```

3.6.2 Constrained Properties

A **constrained property** allows listeners to **veto** (reject) a change *before* it happens. The bean fires a `PropertyChangeEvent` to `VetoableChangeListeners`; if any listener throws `PropertyVetoException`, the change is not applied. The setter must declare `throws PropertyVetoException`. The helper class is `VetoableChangeSupport`.

```
public class Account {
    private int balance;
    private final VetoableChangeSupport vcs = new VetoableChangeSupport(this);
    public void addVetoableChangeListener(VetoableChangeListener l) { vcs.addVetoableChangeListener(l); }

    public void setBalance(int nb) throws PropertyVetoException {
        vcs.fireVetoableChange("balance", balance, nb); // listeners may throw
        balance = nb; // only if no veto
    }
}

// listener that vetoes negative balance
acc.addVetoableChangeListener(evt -> {
    if ((Integer) evt.getNewValue() < 0)
        throw new PropertyVetoException("Balance cannot be negative", evt);
});
```

Bound property	Constrained property
Listeners are told <i>after</i> the change	Listeners are asked <i>before</i> the change
<code>PropertyChangeListener</code>	<code>VetoableChangeListener</code>
<code>PropertyChangeSupport.firePropertyChange()</code>	<code>VetoableChangeSupport.fireVetoableChange()</code>
Cannot stop the change	Any listener can reject the change
Setter has no extra exception	Setter throws <code>PropertyVetoException</code>

Note: A property can be both bound and constrained: first ask for vetoes, then change the value, then notify the property-change listeners.

3.7 Persistence

Persistence means saving the state of a bean so that it can be restored later. Beans achieve this using **object serialization**: the bean implements `Serializable` and its non-transient fields are written with `ObjectOutputStream` and read with `ObjectInputStream`. Fields marked `transient` (and static fields) are not saved.

```
// save
try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("student.ser"))) {
    out.writeObject(student);
}
// restore
try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("student.ser"))) {
    Student s = (Student) in.readObject();
}
```

An alternative is the **XML persistence** mechanism, `java.beans.XMLEncoder` and `XMLDecoder`, which stores only public properties as human-readable XML and does not require

Serializable.

```
try (XMLEncoder enc = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("student.xml"))))
    enc.writeObject(student);
}
```

3.8 Customizers

A **customizer** is a special GUI supplied by the bean author that lets users configure the bean in a more user-friendly way than the default property sheet — for example a wizard that sets several related properties together. It is a class that:

- implements the `java.beans.Customizer` interface,
- extends `java.awt.Component` (usually a `JPanel`) to provide the GUI,
- has a public no-argument constructor,
- implements `setObject(Object bean)` to receive the bean being customised and `addPropertyChangeListener / removePropertyChangeListener`.
- is associated with the bean through `BeanDescriptor` in its `BeanInfo` class.

```
public class StudentCustomizer extends JPanel implements Customizer {
    private Student bean;
    private JTextField nameField = new JTextField(15);
    private PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public StudentCustomizer() {
        add(new JLabel("Name:")); add(nameField);
        nameField.addActionListener(e -> {
            String old = bean.getName();
            bean.setName(nameField.getText());
            pcs.firePropertyChange("name", old, bean.getName());
        });
    }
    public void setObject(Object o) { bean = (Student) o; nameField.setText(bean.getName()); }
    public void addPropertyChangeListener(PropertyChangeListener l) { pcs.addPropertyChangeListener(l); }
    public void removePropertyChangeListener(PropertyChangeListener l) { pcs.removePropertyChangeListener(l); }
}
```

3.9 The Java Beans API

The main classes and interfaces are in the package `java.beans`:

Class / Interface	Description
<code>Introspector</code>	Gets <code>BeanInfo</code> about a bean's properties, events and methods
<code>BeanInfo</code> , <code>SimpleBeanInfo</code>	Explicit information about a bean; <code>SimpleBeanInfo</code> is a convenient base class
<code>PropertyDescriptor</code>	Describes one property (getter, setter, type)

Class / Interface	Description
IndexedPropertyDescriptor	Describes an indexed property
EventSetDescriptor	Describes a set of events the bean fires
MethodDescriptor	Describes an exposed method
BeanDescriptor	Bean class and customizer class
PropertyChangeListener / PropertyChangeEvent / PropertyChangeSupport	Bound properties
VetoableChangeListener / VetoableChangeSupport / PropertyVetoException	Constrained properties
Customizer	Interface for custom configuration GUIs
PropertyEditor, PropertyEditorSupport	Edits a property value (e.g. colour chooser)
XMLEncoder / XMLDecoder	XML-based persistence
Beans	Utility class (e.g. instantiate, isDesignTime)

Note: Since Java 9, java.beans lives in the module java.desktop, which is available in a normal JDK.

3.10 Writing JavaBeans

Example: A complete bean with a client program

Steps to create and use a bean

1. Write the class with private fields, getters/setters and a no-arg constructor.
2. Implement Serializable.
3. Compile it (`javac Student.java`) and optionally package it in a JAR.
4. Use it from code, from an IDE builder, or in JSP with `<jsp:useBean>`.

```
// Counter bean – bound property "count"
import java.beans.*; import java.io.Serializable;

public class Counter implements Serializable {
    private int count;
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
    public Counter() { }
    public int getCount() { return count; }
    public void setCount(int c) { int o = count; count = c; pcs.firePropertyChange("count", o, c); }
    public void increment() { setCount(count + 1); }
    public void addPropertyChangeListener(PropertyChangeListener l) { pcs.addPropertyChangeListener(l); }
    public void removePropertyChangeListener(PropertyChangeListener l) { pcs.removePropertyChangeListener(l); }
}
```

```
// Client
public class CounterDemo {
    public static void main(String[] args) {
        Counter c = new Counter();
        c.addPropertyChangeListener(e ->
            System.out.println("count changed " + e.getOldValue() + " -> " + e.getNewValue()));
        c.increment();    // count changed 0 -> 1
        c.increment();    // count changed 1 -> 2
    }
}
```

Using a bean in JSP

```
<jsp:useBean id="s" class="com.example.Student" scope="session" />
<jsp:setProperty name="s" property="name" value="Ram" />
Name: <jsp:getProperty name="s" property="name" />
```

3.11 Short Questions for Revision

- What is a JavaBean? List its conventions and advantages.
- What is introspection? How does BeanInfo help?
- Differentiate bound and constrained properties with code.
- What are customizers and why are they used?
- How is persistence achieved in beans? Compare Serializable and XMLEncoder.
- Write a bean class with a bound property and a listener to test it.

Key Takeaways — Unit 3

- ✓ A JavaBean is a reusable class with a no-arg constructor, private fields with getters/setters, and (usually) Serializable.
- ✓ Introspection (java.beans.Introspector) discovers properties, events and methods through naming patterns or an explicit BeanInfo class.
- ✓ Design patterns: getX/setX/isX for properties; addXListener/removeXListener for events.
- ✓ Bound properties notify after change (PropertyChangeSupport); constrained properties allow veto before change (VetoableChangeSupport).
- ✓ Persistence uses serialization (or XMLEncoder/XMLDecoder); transient fields are not saved.
- ✓ Customizers give builder tools a friendlier GUI to configure a bean; registered through BeanDescriptor.

— End of Unit 3 —

Reminder: these notes may contain errors — verify against your textbook and run the code yourself.