GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Unit 4. ACCESS USING COMPONENTS

Part 1. Introducing JavaBeans

Acceso a Datos (ADA) (a distancia en inglés)

CFGS Desarrollo de Aplicaciones Multiplataforma (DAM)

Abelardo Martínez

Year 2024-2025

Credits

- Notes made by Abelardo Martínez.
- Based and modified from Sergio Badal (www.sergiobadal.com).
- The images and icons used are protected by the LGPL licence and have been obtained from:
  - https://commons.wikimedia.org/wiki/Crystal_Clear
  - https://www.openclipart.org

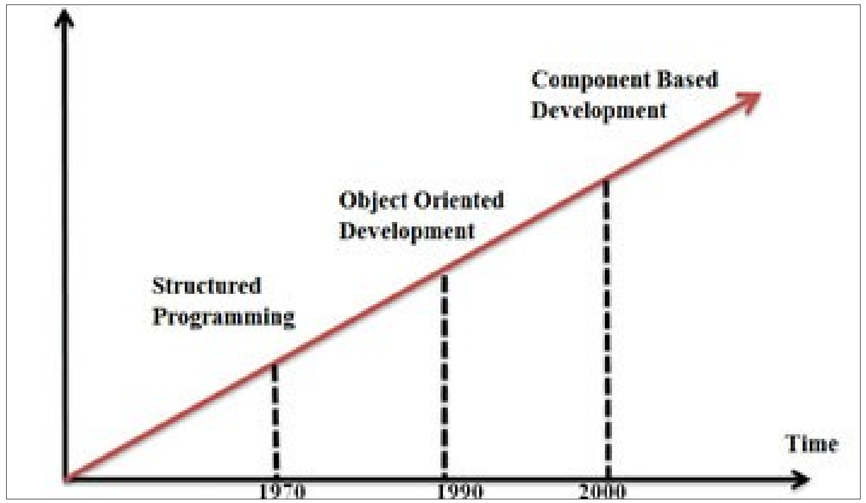
Contents

1. WHAT IS COMPONENT-BASED DEVELOPMENT?
2. USING JAVABEANS
3. JAVABEANS PROPERTY TYPES
4. BASIC EXAMPLE
   1. Part 1. Creating the beans
   2. Part 2. Running the beans
5. ACTIVITIES FOR NEXT WEEK
6. BIBLIOGRAPHY

1. WHAT IS COMPONENT-BASED DEVELOPMENT?

CBD (Component-Based Development)

- **Component-based software engineering (CBSE)**, also called **component-based development (CBD)**, is a process that focuses on the design and development of computer-based systems with the use of **reusable software components**.
- A **component** is a distributable unit of software. It is a physical and replaceable part of the system that conforms to and provides the realization of interfaces.
- It may optionally have an internal structure and down a set of ports that formalize its interaction point.



With CBD, the focus shifts from software programming to software system composing.

Some experts say it is the evolution of Object Oriented Programming.

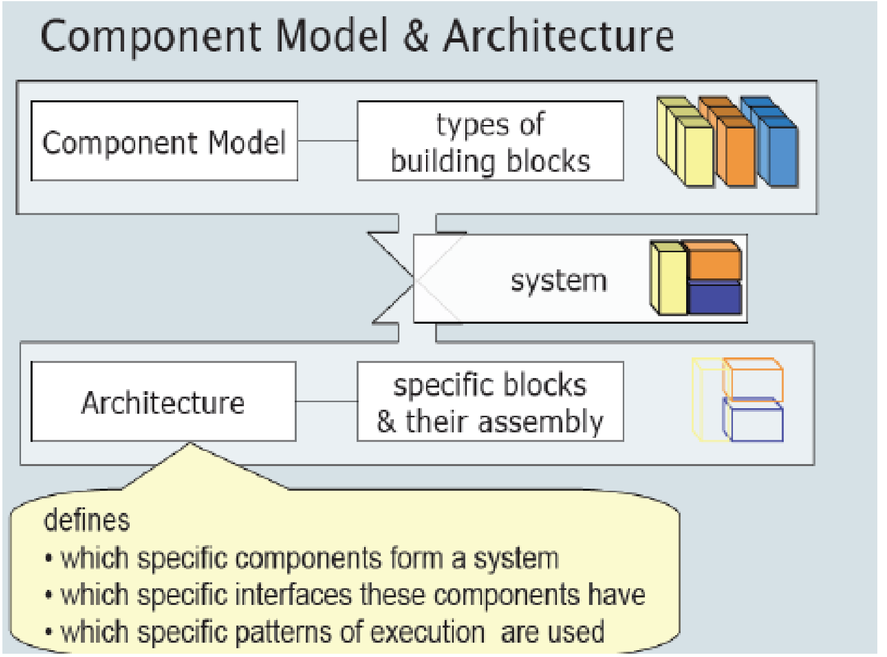
For further information:
https://www.educba.com/component-based-software-engineering/

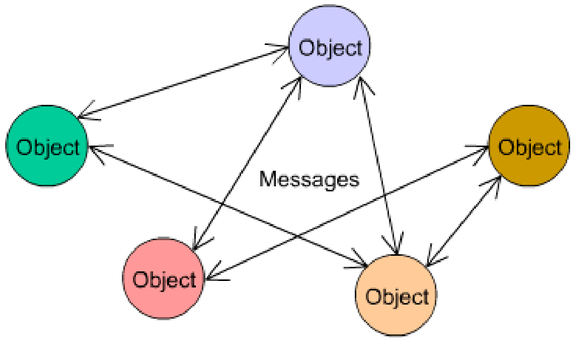
OO vs CBD

If every Java class is a component, and if classes and components share so many qualities, then what is the difference between traditional object-oriented programming and component-oriented programming?

In a nutshell:

- **object-oriented programming** focuses on the relationships between classes that are combined into one large binary executable
- **component-oriented programming** focuses on interchangeable code modules that work independently and do not require you to be familiar with their inner workings to use them.





Interaction of objects via message passing

For further information:
https://www.oreilly.com/library/view/programming-net-components/0596102070/ch01s02.html

2. USING JAVABEANS

Components in Java

You can easily build components in Java using the JavaBean technology, which adds all the capabilities of the Java language, such as ease of use and platform independence, to the equation.
A bean (the JavaBean terminology for a component) is a regular Java class and, as such, can be used programmatically as any other class.
However, to be considered a bean, a class must follow some conventions and guidelines so that it can be manipulated by visual application builder tools.

All Swing and AWT classes (graphics libraries) are JavaBeans since GUI components are ideal JavaBeans, but this is not a requirement neither our case of study.

The JavaBean concept is introduced in 1996 by Sun Microsystems and defined as "***a reusable, platform independent component that can be manipulated visually in a builder tool***".

For further information:
https://docs.oracle.com/javase/tutorial/javabeans/quick/index.html

JavaBeans

- JavaBeans are basically **POJOs** (Plain Old Java Object), defined according to the norms based on the software component model and must be **Serializable**.
- The values assigned to the property of a JavaBean class determine its functionality. Although a property is publicly accessible, its direct use (read/write) is actually restricted by actual implementation to access data with member functions (**setters/getters**).

Syntax for setter methods:

1. It should be public in nature.
2. The return-type should be void.
3. The setter should be prefixed with set.
4. It should take some argument.

Syntax for getter methods:

1. It should be public in nature.
2. The return-type should not be void.
3. The getter method should be prefixed with get.
4. It should not take any argument.

```
class MyDriver{
   private String name;
   // ...
   public String getName(){
      return name;
   }
   public void setName(String name){
      this.name=name;
   }
   // ...
}
```

Properties are also observable; that means they can **trigger notification** to interested parties regarding value changes (using **events**).

3. JAVABEANS PROPERTY TYPES

Property types

Simple property

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

    public int getMouthWidth() {
        return mMouthWidth;
    }

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

Indexed property

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

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

Types of channels

Java standardly contemplates two types of channels linked to JavaBeans. Each channel has a specific type of interface associated with it that subscribers will have to implement.

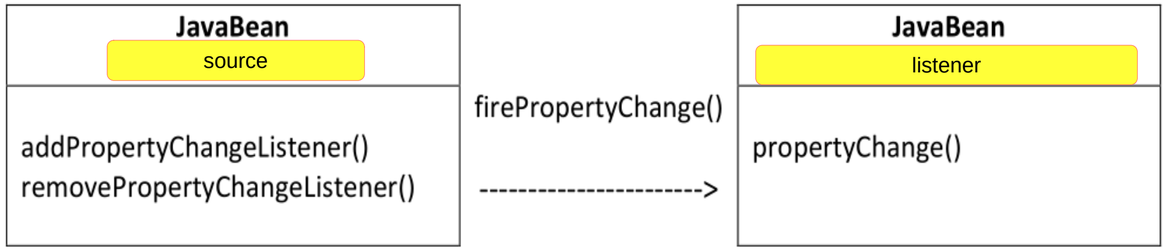
1) **Bound property**
2) **Constrained property**

Bound property

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

1) One bean class has to act as the **SOURCE** and, at least, one bean class has to work as a **LISTENER**.
2) The bean class (source) includes **addPropertyChangeListener()** and **removePropertyChangeListener()** methods for communicating with the LISTENER(S).
3) When a bound property is changed (at the SOURCE bean), the SOURCE bean sends a **PropertyChangeEvent** to its registered LISTENER(S).
4) Properties **PropertyChangeEvent** and **PropertyChangeListener** live in the java.beans package.
5) The java.beans package also includes a class, **PropertyChangeSupport**, that takes care of most of the work of bound properties. This handy class keeps track of property listeners and includes a convenience method that fires property change events to all registered listeners.

Bound property (example)




```
import java.beans.*;

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

    public int getMouthWidth() {
        return mMouthWidth;
    }

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

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

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

Constrained property

A constrained property is a special kind of bound property.

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

- The veto listeners are separate from the property change listeners.

- Fortunately, the java.beans package includes a **VetoableChangeSupport** class that greatly simplifies constrained properties.

```
import java.beans.*;

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

    public int getMouthWidth() {
        return mMouthWidth;
    }

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

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

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

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

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

4. BASIC EXAMPLE

Creating a basic example

We’re now building a **Java** (Ant) project from scratch to create two **JavaBeans** (beans), setting up their properties and methods and watch them interact!

Eclipse vs NeatBeans

The way of working is very similar, since both IDEs import Java classes (beans).
Knowing that, and in our case, we prefer to continue with Eclipse.

Interacting between a couple of beans

The first bean (source) will be a **product** and the second one an **order** with this simple properties:

- **product** = id + desc + price + current stock + minimum stock
- **order** = id + amount + associated product

And this will be the **interaction between them**:

1)When the current stock of the **product** (source bean) changes it'll be firing an event so the **order** (listener bean) will react doing some stuff (just printing some messages).

2)Also, when the minimum stock of the **product** is changed, it'll be firing an event so the **order** (listener bean) will react doing some stuff (just printing some messages).

How the programme works

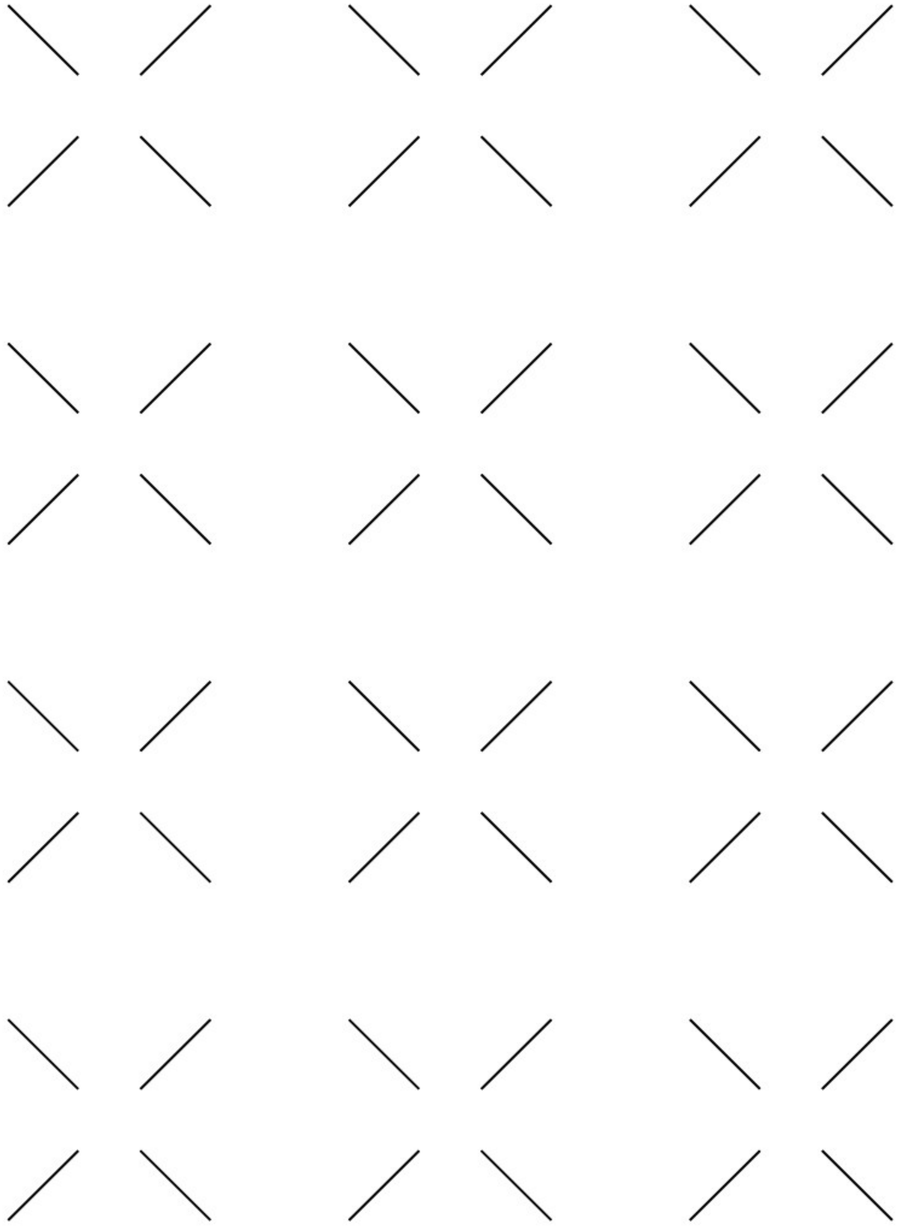
Main class (defined later)

```
public class TestADAbeans {

   public static void main(String[] stArgs) {
      /*
       * Creating the objects
       */
      //Object source
        //ProductBean(int iProductid, String stDescription, float fPrice, int
iCurrentstock, int iMinstock)
        //Setting currentStock to 101 units and minimumStock to 100 units
        ProductBean objProductBean = new ProductBean(1, "Robot hoover", 399, 101, 100);
        //Object listener
        OrderBean objOrderBean = new OrderBean();
        /*
         * Assign the object source to the listener
         * Start the listener object
         */
        objOrderBean.setobjProductBean(objProductBean);
        objProductBean.addPropertyChangeListener(objOrderBean);
        /*
         * Firing events
         */
        //Setting currentStock to 40 (below the minimum advisable)
        System.out.println("****** product.setCurrentStock(40):");
        objProductBean.setiCurrentStock(40);
        //Setting minimumStock to 50 (over the current stock)
        System.out.println("****** product.setMinStock(50):");
        objProductBean.setiMinStock(50);
    }
}
```

Output

```
****** product.setCurrentStock(40):
[OrderBean says... ]
Current stock is now less than minimum stock!
=> Old current Stock: 101
=> New current Stock: 40
It will place an order for this product: Robot
hoover
****** product.setMinStock(50):
[OrderBean says... ]
Minimum stock is now greater than current
stock!
Old minstock Stock: 100
New minstock Stock: 50
It will place an order for this product: Robot
hoover
```

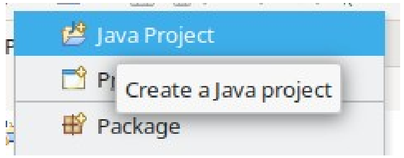
4.1 Part 1. Creating the beans

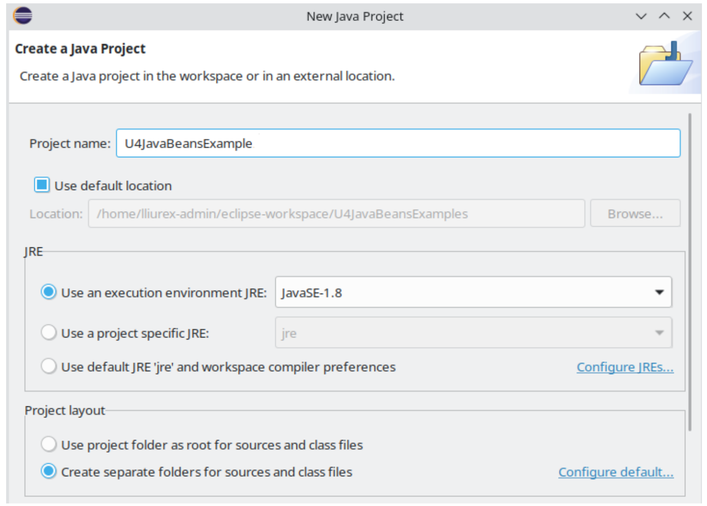


STEP 1.1: Create a project and two classes (beans)

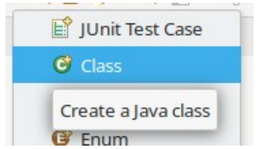
Create a Project and two classes (JavaBeans):

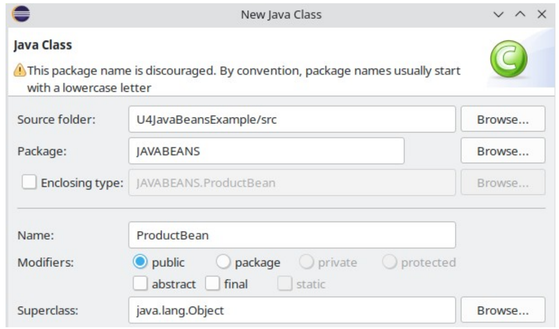
1 Create Java Project

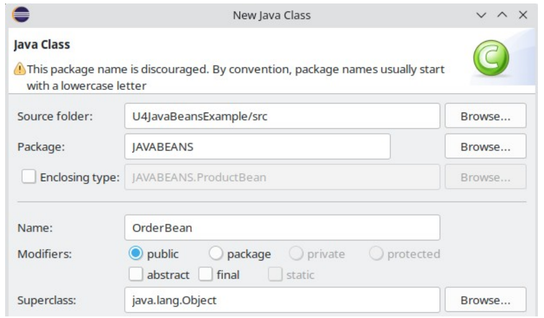




2 Create Java Class x 2







3 Result

STEP 1.2: Set the SOURCE and the LISTENER(S)

- The source bean must implement the *Serializable* interface.
- The listener(s) bean(s) must implement the *Serializable* interface and the *PropertyChangeListener*.

```java
import java.beans.*;
import java.io.Serializable;

public class ProductBean implements Serializable {

   /*
    * -----------------------------------------
    * ATTRIBUTES
    * -----------------------------------------
    */
    private static final long serialVersionUID = 1L;
    private int iProductID;
    private String stDescription;
    private float fPrice;
    private int iCurrentstock;
    private int iMinstock;
```

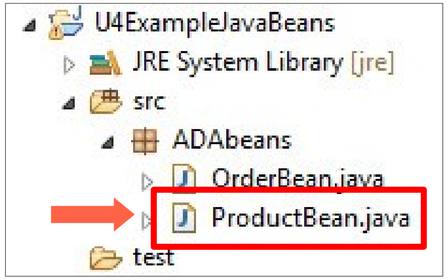
```java
import java.beans.*;
import java.io.Serializable;

public class OrderBean implements Serializable, PropertyChangeListener {

   /*
    * -----------------------------------------
    * ATTRIBUTES
    * -----------------------------------------
    */
   private static final long serialVersionUID = 1L;
   private int iOrdernumber;
   private ProductBean objProductBean;
   private int iAmount;
```

STEP 1.3: Set up the source bean (bean #1)

Open ProductBean, remove the sample stuff, set the imports, create the properties, add the **PropertyChange** support and the setters and getters except for the ones in **red**.

```
package ADAbeans;

import java.beans.*;
import java.io.Serializable;


public class ProductBean implements Serializable {

    private int iProductid;
    private String stDescription;
    private float fPrice;
    private int iCurrentstock;
    private int iMinstock;

    private PropertyChangeSupport propertySupport;

    public ProductBean() {
        propertySupport = new
PropertyChangeSupport(this);
    }

    public ProductBean(int iProductid, String
stDescription, float fPrice, int iCurrentstock, int
iMinstock) {
        propertySupport = new
PropertyChangeSupport(this);
        this.iProductid = iProductid;
        this.stDescription = stDescription;
        this.fPrice = fPrice;
        this.iCurrentstock = iCurrentstock;
        this.iMinstock = iMinstock;
    }
```

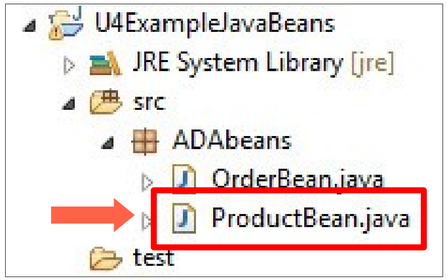
```
public String getstDescription() {
        return stDescription;
    }

public void setstDescription(String stDescription) {
        this.stDescription = stDescription;
    }

public int getiProductid() {
        return iProductid;
    }

public void setiProductid(int iProductid) {
        this.iProductid = iProductid;
    }

public float getfPrice() {
        return fPrice;
    }

public void setfPrice(float fPrice) {
        this.fPrice = fPrice;
    }

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

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



STEP 1.4: Add the critical setters and getters (bean #1)

Create now the setters and getters to knock at the door of the listener(s) by calling firePropertyChange with a **tagname** (whatever) and **old** and **new** value.

```
public int getiCurrentStock( ) {
   return iCurrentstock;
}

public void setiCurrentStock(int newValue) {
   // If NEW current stock is below minimum, order this product!
   int oldValue = this.iCurrentstock;
   this.iCurrentstock = newValue;

   if (this.iCurrentstock < getMinStock( )) // Call OrderBean
   {
      propertySupport.firePropertyChange("currentStockBelowMinStock", oldValue, this.iCurrentstock);
   }
}
```



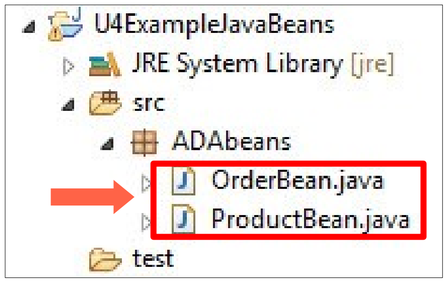

```
public int getiMinStock( ) {
   return iMinstock;
}

public void setiMinStock(int newValue) {
   // If MIN stock has been raised over current stock, order this product!
   int oldValue = this.iMinstock;
   this.minstock = newValue;
   if (this.iMinstock > getCurrentStock( )) // Call OrderBean
   {
         propertySupport.firePropertyChange("minStockRaisedOverCurrentStock", oldValue, this.iMinstock);
   }
}
```

STEP 1.5: Set up the listener bean (bean #2)

Open **OrderBean**, remove the sample stuff, set the imports, create the properties, add the **PropertyChange** support and the setters and getters.

```
import java.beans.*;
import java.io.Serializable;


public class OrderBean implements Serializable, PropertyChangeListener {

    private int iOrdernumber;
    private ProductBean objProductBean;
    private int iAmount;

    public OrderBean( ) {

    }

public OrderBean(int iOrdernumber, int iAmount, ProductBean objProductBean)
{
        this.iOrdernumber = iOrdernumber;
        this.objProductBean = objProductBean;
        this.iAmount = iAmount;
    }
```

```
public int getiOrderNumber( ) {
      return iOrdernumber;
  }
public int getiAmount( ) {
      return iAmount;
  }
public ProductBean getobjProductBean( ) {
      return this.objProductBean;
  }
public void setiOrderNumber(int iOrdernumber) {
      this.iOrdernumber = iOrdernumber;
  }

public void setiAmount(int iAmount) {
      this.iAmount = iAmount;
  }
public void setobjProductBean(ProductBean objProductBean) {
      this.objProductBean = objProductBean;
  }
```



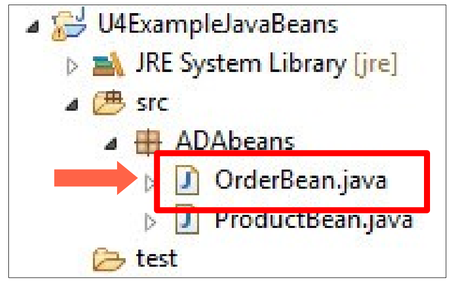
STEP 1.6: Set up what to do when event fires (bean #2)

Now we must decide what to do when the PropertyChange event is fired.

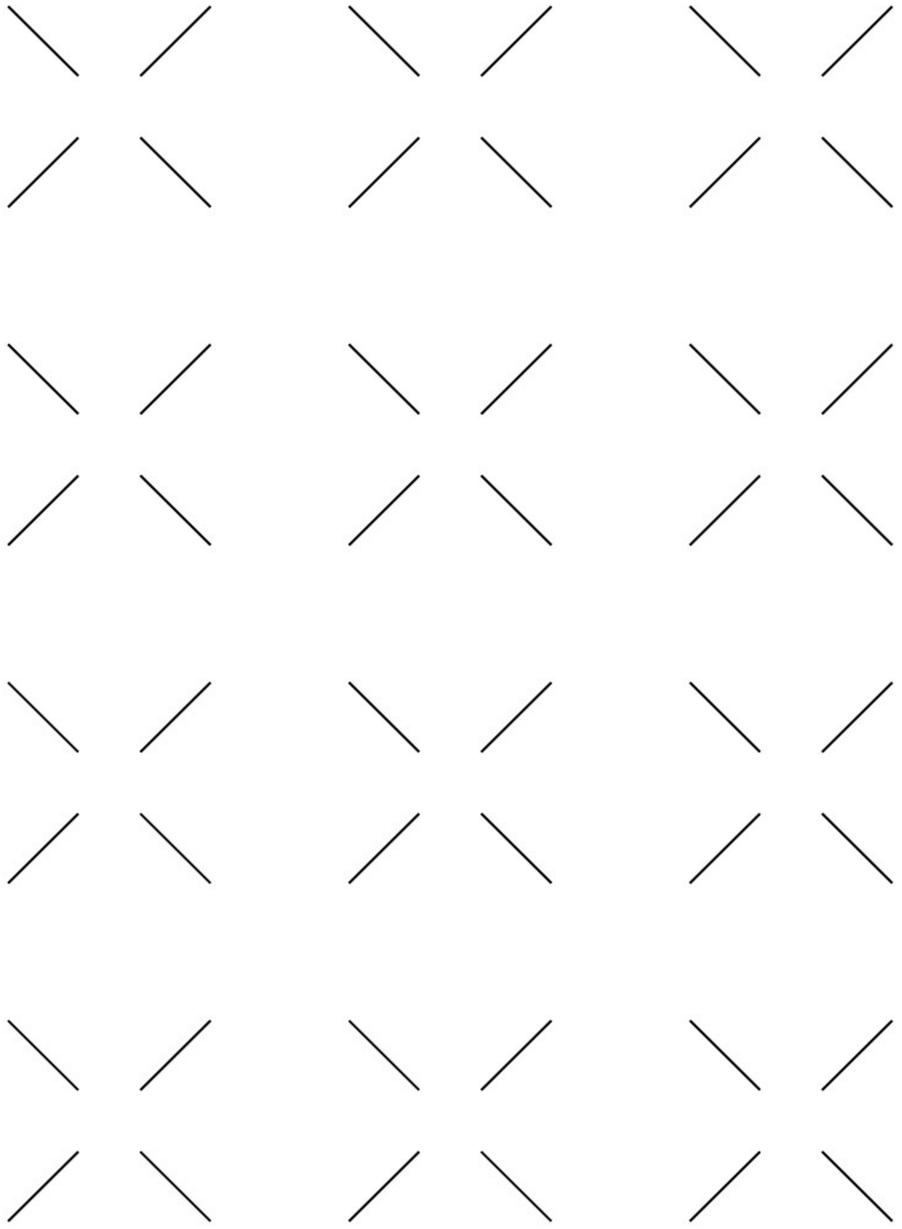
In this BASIC example, we're just showing some messages.

As an extension, we should call the database to create the orders.

```
    public void propertyChange(PropertyChangeEvent evt) {
      if (evt.getPropertyName().equals("currentStockBelowMinStock"))
      {
          System.out.printf("[OrderBean says... ]%n");
          System.out.printf("Current stock is now less than minimum stock!%n");
          System.out.printf("=> Old current Stock: %d%n", evt.getOldValue());
          System.out.printf("=> New current Stock: %d%n", evt.getNewValue());
          System.out.printf("It will place an order for this product: %s%n",
              product.getDescription());
      }
      if (evt.getPropertyName().equals("minStockRaisedOverCurrentStock"))
      {
          System.out.printf("[OrderBean says... ]%n");
          System.out.printf("Minimum stock is now greater than current stock!%n");
          System.out.printf("Old minstock Stock: %d%n", evt.getOldValue());
          System.out.printf("New minstock Stock: %d%n", evt.getNewValue());
          System.out.printf("It will place an order for this product: %s%n",
              product.getDescription());
      }

  }
```



4.2 Part 2. Running the beans



Executing a basic example

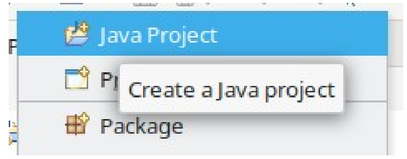
Once the beans are ready, we need to let them run!
These are the steps:

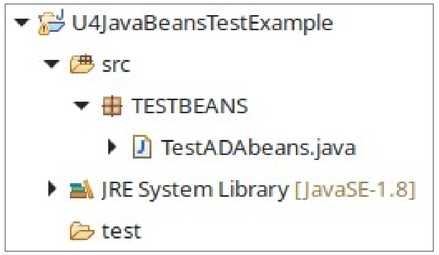
1)First, we’re saving our beans in a jar file (**clean & build**).
2)Then, we’re importing that JAR file (library) in a **new class**.
3)Finally, we’re **running a piece of code** to watch both beans interact.

STEP 2.1: Create and import JAR file

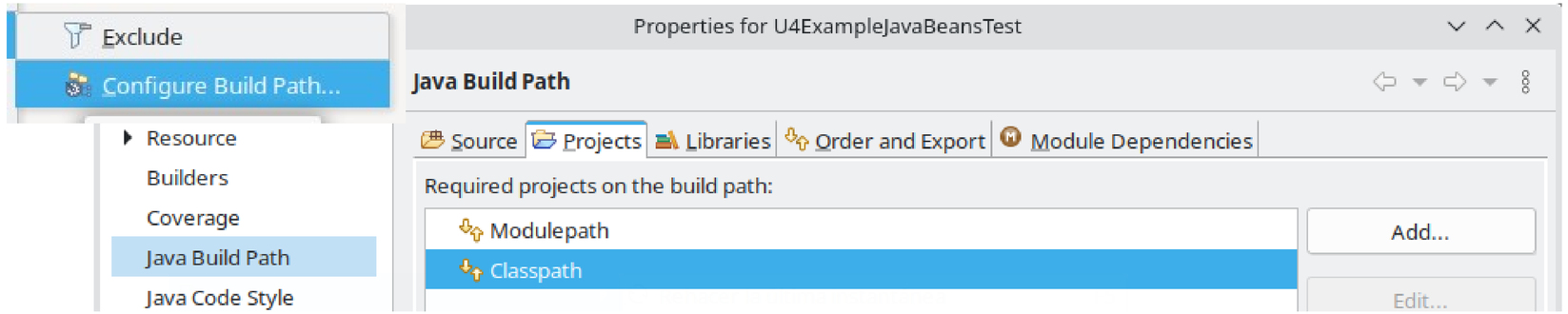
Create a Project, a Java Class and import our beans:

1 Create new Java Project

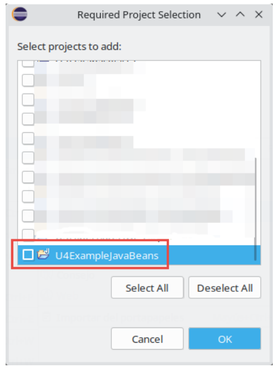


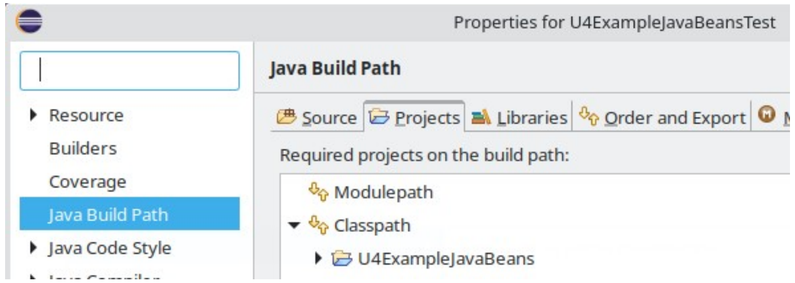


2 Configure build path



3 Import JAR file





4 Result

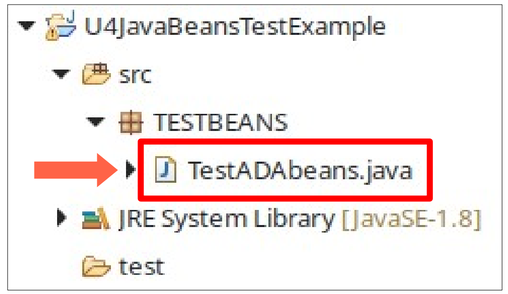
Apply and close

STEP 2.2: Run a piece of code

Run this simple piece of code within the recently created class to fire the events when some properties (attribute values) change.

We could have done the same inside the first class, but this way seems more interesting since we're REUSING COMPONENTS via libraries (importing JAR file).

```
public class TestADAbeans {

   public static void main(String[] stArgs) {
      /*
       * Creating the objects
       */
      //Object source
       //ProductBean(int iProductid, String stDescription, float fPrice, int iCurrentstock, int
iMinstock)
       //Setting currentStock to 101 units and minimumStock to 100 units
       ProductBean objProductBean = new ProductBean(1, "Robot hoover", 399, 101, 100);
       //Object listener
       OrderBean objOrderBean = new OrderBean( );
       /*
        * Assign the object source to the listener
        * Start the listener object
        */
       objOrderBean.setobjProductBean(objProductBean);
       objProductBean.addPropertyChangeListener(objOrderBean);
       /*
        * Firing events
        */
       //Setting currentStock to 40 (below the minimum advisable)
       System.out.println("****** product.setCurrentStock(40):");
       objProductBean.setiCurrentStock(40);
       //Setting minimumStock to 50 (over the current stock)
       System.out.println("****** product.setMinStock(50):");
       objProductBean.setiMinStock(50);
    }
}
```

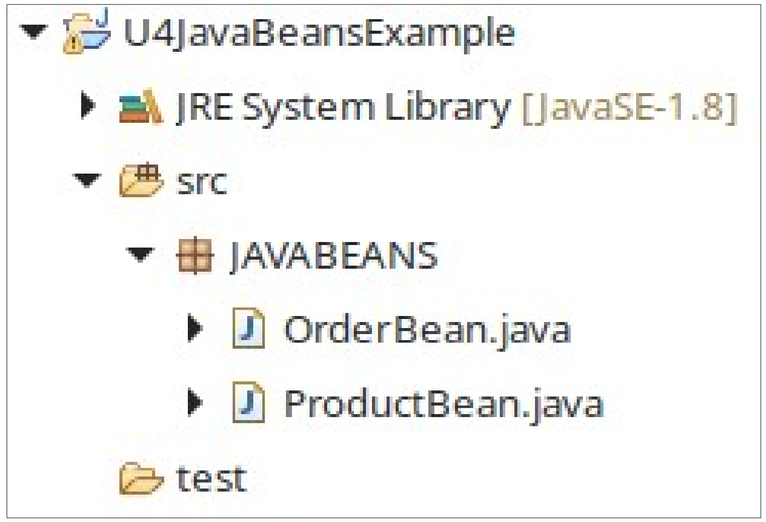


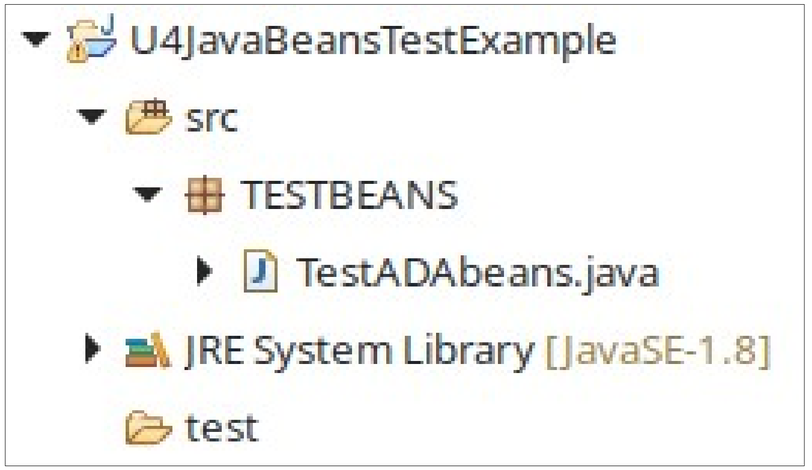
Output

```
****** product.setCurrentStock(40):
[OrderBean says... ]
Current stock is now less than minimum stock!
=> Old current Stock: 101
=> New current Stock: 40
It will place an order for this product: Robot
hoover
****** product.setMinStock(50):
[OrderBean says... ]
Minimum stock is now greater than current
stock!
Old minstock Stock: 100
New minstock Stock: 50
It will place an order for this product: Robot
hoover
```

Download the code

Download now the projects **U4JavaBeansExample** and **U4JavaBeansTestExample** from the Aula Virtual and try it yourself.





5. ACTIVITIES FOR NEXT WEEK

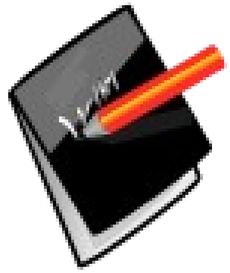
Proposed activities

Check the suggested exercises you will find at the “Aula Virtual”. **These activities are optional and non-assessable but** understanding these non-assessable activities is essential to solve the assessable task ahead.

Shortly you will find the proposed solutions.

6. BIBLIOGRAPHY

Resources



- Oracle Java Documentation. JavaBeans Component API. https://docs.oracle.com/javase/8/docs/technotes/guides/beans/index.html
- JavaBeans Tutorial - MIT - Massachusetts Institute of Technology. http://web.mit.edu/javadev/doc/tutorial/beans/index.html
- Tutorials freak. JavaBeans Class in Java: Properties, Examples, Benefits, Life Cycle. https://www.tutorialsfreak.com/java-tutorial/javabeans
- I/O Flood. Java Bean Explained: Object Encapsulation Guide. https://ioflood.com/blog/java-bean/
- Josep Cañellas Bornas, Isidre Guixà Miranda. Accés a dades. Desenvolupament d'aplicacions multiplataforma. Creative Commons. Departament d'Ensenyament, Institut Obert de Catalunya. Dipòsit legal: B. 29430-2013. https://ioc.xtec.cat/educacio/recursos