DAM. UNIT 4. ACCESS USING COMPONENTS. INTRODUCING JAVABEANS

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

Unit 4. ACCESS USING COMPONENTS

Introducing JavaBeans

Abelardo Martínez

Year 2024-2025

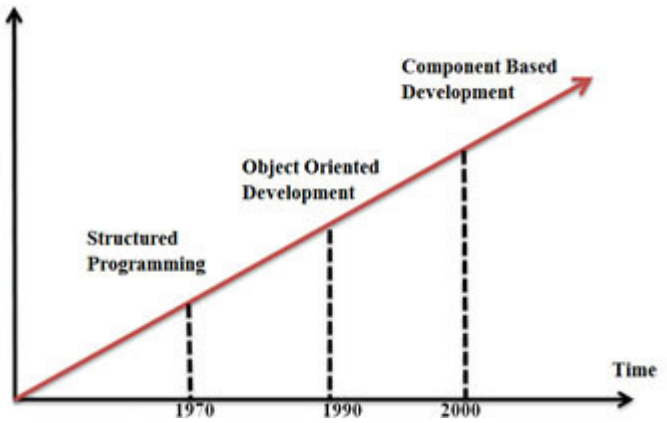
1. What Is Component Based Development?

Component **B**ased **D**evelopment (**CBD**) is an approach to software development that focuses on the design and development of reusable components. You can break your monolith into components:

- Using a producer/consumer model.
- Reusable/shared libraries.
- By front-end/back-end.
- Breaking up software into components is the right thing to do. No one can argue that. It's what the Agile revolution is all about. And using components can better serve business needs than microservices.

What Is Component Based Architecture and Why Should You Use It?

Component based architecture refers to a framework for software development involving the design of reusable components, i.e. component based development. One reason to use a component based architecture is that it stays up-to-date without rebuilding it from scratch. That makes component based architecture a better fit for companies with complex, monolithic codebases. Using components turns a monolith into software building blocks. And these components can be combined, reused, and versioned.



Approach to the concept of a software component

As we have to deal with increasingly complex applications, we will realise that the number of classes will start to grow disproportionately and that managing to bring them all together by applying criteria of reusability, efficiency and quality is a really difficult task. In order to try to alleviate this difficulty, the concept of the **software component** appeared at the beginning of this century. The idea is to **treat software as if it were a mechanical**

product.

In the mechanical industry, these systems are based on making partial constructions, organised in such a way that they can all be made at the same time and that, once they have been assembled, result in the final product. Each of these constructions is called a component. Thus we can see that the engine, the wheels and all their internal mechanisms, the seats, doors, bonnets, dashboards, etc., can be considered components because they have an independent construction system and, once they are finished, they can be coupled together to produce the final finished product.

Similarly, if it were possible to organise the whole set of classes that make up an application into independent components, so that each part could be created concurrently and an easy and efficient final assembly could be achieved, we would have a possible solution to the problem of building complex applications.

1.1. Definition of component

Continuing with the example of the car, with which we have introduced the concept of component, we will take the opportunity to discuss when a grouping of parts can be considered a component and when not. Establishing a parallelism with the example of the car, we will say that **software components** are those executable and independent units that make up an application, which have a perfectly defined functionality and way of interacting, so that their assembly with the rest of the components can be done without having to modify the internal code of any of them and without having to modify the internal code of any of them and, moreover, at any time it is possible to replace them with another equivalent component (with an identically defined functionality).

Probably the most obvious form of software components that you can recognise at first glance are the so-called **plugins** or **extensions**. Plugins are independent, executable units that can be part of an application with which they interact. Their incorporation does not require any modification of the application code or other components. The installation is very easy to carry out and at any time we can replace the plugin with a more efficient one or with a superior version of the same.

Plugins are very independent components that add functionality to the application, but are not necessary for the execution of the rest of the functionality. But not all components are so independent, some are part of the core functionality of the application and therefore their performance will affect the execution of the whole application. It is not possible to run the application without them and if they are removed, they have to be replaced by equivalent ones.

If we look back, we can look at JDBC drivers as a clear example. These are low-level components from which we build other, more application-specific components, but they are, after all, components that meet all the above requirements. They are independent executable units that are part of the applications, that respond to a well-defined functionality and that can be easily resembled and interchanged with other equivalent ones.

1.2. Phases in the construction of software components

The above definition of a software component may help us to get an idea of the concept, but it does not help us much to know what we have to do to ultimately build a component. Like any piece of software, components will have to be **specified**, **implemented**, **packaged** and **deployed** in some specific application and under some specific platform. Each of these aspects has specific characteristics that are worth considering if we want to build a quality product.

1.2.1. Specification of software components

By specification of the components, we must understand all those descriptions that allow the rest of the subsequent phases to be carried out. That is to say, the detailed description of the operations that the component will have to support, of the set of objects that integrate the component, of the operations other components will have to call during execution, of the way in which the component will have to be packaged and which elements will form part of the package or how the installation and configuration of the component will have to be carried out to achieve a correct assembly, etc.

Of all the descriptions mentioned, those that refer to the detailed description of the operations are of paramount importance, because with them we have to ensure the independence of each component, the final assembly and the ability to create several equivalent and interchangeable components. Although specifications are usually made with independent languages that support both textual and diagrammatic descriptions, such as UML, each programming language usually allows, fully or partially, this information in the code to make the implementation phase easier.

Java language

The Java language uses two elements to embody the detailed description of the operations supported by the component:

- **Interfaces**
- **Comments on documentation**

On the one hand, it uses **interfaces** to describe the syntax of the operations and to ensure the independence and equivalence of the components. The interfaces **determine what operations will be available to the component and what their syntax will be**. Subsequently, the interfaces will have to be implemented by means of classes that will code all the methods, but the use of interfaces allows several different implementations to be made without losing interoperability with the rest of the components and application.

The use of Java interfaces is not a requirement for components. In fact, the component's requirement is just a clear and unique syntax for each operation, and this can also be done by directly implementing classes. Interfaces, however, have a great advantage over classes: they allow several versions of the same component to be used at the same time, in the same application.

On the other hand, Java also has **documentation comments** to describe textually the

functionality of the methods of a class. When documentation comments are used in an interface, their description is extended to any of its implementations. Although code documentation is important in any development, it is of paramount importance when it comes to the interfaces of a component, since good documentation facilitates and clarifies the functionality that will have to be achieved in the different implementations that will be made throughout the life cycle of the component. It also defines how other components and applications will have to use the implemented component.

That is why we will always include the following aspects in the documentation of the interfaces:

- We will describe in detail the data input for each operation.
- We shall describe in detail the type of result and how it is obtained or calculated from the initial data. This is not a codification of the documentation, but a detailed description that does not lead to misinterpretation.
- Describe the conditions under which an error will occur and explain what type of exception will be obtained in each case.
- We will describe what requirements are necessary to ensure the proper functioning of each operation and, if known, we will also describe those adverse conditions that are not capable of ensuring that the response of the component is correct. It is obvious that the complexity of certain components will prevent us from being able to make an implementation that controls absolutely all the different situations from which it would be possible to access some of the component's operations. It is therefore necessary to delimit, when necessary, the appropriate operating conditions.

1.2.2. Implementation of software components

The implementation of components refers to the coding of a specification. As already mentioned, the same specification can be implemented in several ways. The reasons for different implementations can be very diverse. Imagine that the same application is used by companies that need to work in a distributed way and others that have enough with a local application. If the functionality of both applications were the same, we could make two implementations of those components that were sensitive to working in a distributed way, so that it would be possible to have a local or distributed application depending on the combination of components that we make.

On other occasions, there may not be an ideal way to implement a component, and depending on the conditions where the installation has to be done, we would have to choose one or the other alternative. Often there are processes that can be accelerated at the expense of wasting memory and/or hard disk space, but not all companies are willing to expand their hardware resources and, therefore, depending on the amount of resources available, we could compose a more austere or more wasteful application.

Sometimes, implementations are only intended to adapt components to a specific platform, and so we can have specific components for Linux, Windows, web systems, etc.

Java language

In Java, the implementation will have to be done with **one or more classes that implement all the interfaces defined in the component**. Although it is possible to implement all the interfaces in a single class, it is important not to create megaclasses that absorb all the functionality of the component. It is important to distribute it among several classes to reduce complexity.

1.2.3. Packaging of components

One of the basic characteristics of any component refers to the unit. In order to be distributed, combined and integrated in different applications, it is necessary to organise all the resources that make up the component into a unit that can be easily serialised and copied wherever it is needed.

Generally, we will distinguish between two types of resources:

1. The resources that will make up the functional component (executables, images, graphical interface designs, etc.).
2. The installation and configuration resources necessary to adapt the component to a specific application (configuration of the specific connections to the DBMS, configuration of the printed reports, of the registry systems or other configurations that the component requires).

The former are usually packaged together in a single file or in a small number, for ease of distribution. Executable code is usually organised in dynamic library files. It is often possible to incorporate other resources, such as images or icons within the same library to reduce the number of files involved. Configuration resources, on the other hand, are usually kept in separate files to facilitate their modification. When components are implemented with other interpreted or pseudo-compiled languages such as Java, the component will also have to be packaged in some way, but this depends on the utilities available in each language.

Packaging of Java code

As you know, the Java language does not create a single executable binary file, but compiles each class separately. The compiled classes are linked during the execution process to the virtual machine itself. For this reason, Java does not need to create a single executable file. However, in order to keep the code of the components together, Java usually packages all the compiled classes in a single JAR file. This is a .*zip file consisting of at least the compiled classes and a descriptive file called **manifest.mf** located in the internal folder of the JAR file called meta-inf.

The manifest.mf file is useful because it contains information about the component, including where other components and libraries required by the component are located (classpath needed during execution). JAR files can also contain resource files that are not specifically code, for example images, or other formats that are needed during the execution of the component classes.

1.2.4. Deployment of software components

Deployment of software components means the installation or copying of the resources that make up the component to the appropriate location in order to integrate it into the application of which it is to be a part, making it operational.

Often, the component to be deployed contains multiple resources that will need to be copied to different locations on the system where the deployment is made. It will probably require specific configuration, which will need to be detailed. In case the component to be deployed requires the installation of other components, it will be necessary to check that these are already installed on the system and activate their deployment in case they are not. In addition, minor modifications may have to be made to the configuration of the application that incorporates the component in order to make the new component operational.

Making all these changes manually can be very costly when the deployment has to be repeated on multiple workstations or computer systems. To minimise this cost, the industry has created several solutions to automate these changes. Generally, it is a matter of creating packages containing the changes to be made and taking advantage of an unpacking tool to automate them. Typically, operating systems themselves have their own packaging format to express the actions to be performed during a software deployment. For example, the Windows operating system works with the MSI (Microsoft Installer) format, while Linux usually works with RPM (Redhat Package Manager) packages if the platform is derived from a RedHat distribution or with deb packages if it is derived from a Debian distribution.

Each operating system will also have the tool that executes the actions indicated in a particular installation package, so that the deployment consists only of executing a specific package for each component. The format of the packages is usually textual and orders and sections can be identified describing the requirements and actions of the deployment that the installation tool will interpret and execute.

There are also tools to help create the deployment packages. In general, all these tools allow the selection of a set of files to be copied during deployment to a specified path (either expressed in absolute or relative form), the list of requirements to be checked before deployment (usually also indicating the minimum version required), the order files to be executed during deployment, and so on.

The **main problem with creating deployment packages** with these formats is that they depend on the platform where they are to be deployed. That is to say, in the case of having

to deploy them on more than one platform, it will be necessary to create an installation package for each of them. The problem, however, becomes more complicated if the platform where the component is to be installed is unknown. In fact, when updates are made from a public address, the platform where the component will be deployed is not always known, and although a different address may be available for each platform, it is still an added complexity. That is why there are also ways to package components independently of the platform.

It is possible to use standard packaging (e.g. ZIP or TAR format) to achieve this independence. However, this type of packaging only serves to move the component resources to the device where the deployment has to be done and the relevant copy has to be made. It is not possible to control the dependencies or to execute any batch order file. Therefore, if this option is chosen, it is possible that the intervention of an administrator may be necessary to complete the deployment.

Automatic task management tools

This type of file can be created automatically during the development of components using automatic task management tools during software development. We refer to the tools associated with any language that allow compilation, the creation of executables, the generation of packages and the copying of files to local or remote paths. If we use the C or C++ language, we will have the Make tool. On the other hand, if we use **Java**, the automated task management tool is called **Ant**.

Other utilities in Java

There are other utilities for managing the deployment of components and applications regardless of the platform used. These are fairly complex tools for managing the deployment of an entire software project. The **Maven** utility is an example. It is a Java project manager that works with a repository of components and libraries, both the project's own and external. The utility allows to control the different components that are part of a project, as well as the dependencies they have, even taking into account the different versions of a component or a library that may exist. Maven also allows you to synchronise different installations with the main repository of the project. Each time the main repository is updated, Maven will be able to synchronise each installation incorporating only the necessary changes.

2. Creating components with Java. JavaBeans

Although the construction of components should not depend on the language used, it is true that some languages such as Java define a standard model for creating quality components adapted to the characteristics of the language.

Under the name of JavaBeans, Java has embodied its idea of how software components are implemented. In fact, a **JavaBean** is nothing more than a normal **Java class, which has to comply with a series of conventions**: for example, it has to have a default constructor, it does not have to have public access to its attributes but, in the event that they can be manipulated, this must always be done by means of write and read access descriptors.

These simple requirements, in combination with the potential of the language, provide the set of characteristics that make it possible to define a standard component model for Java. You remember that the Java language, when it instantiates any object, also loads in memory a lot of information regarding the class (meta-information). The convention that JavaBeans always have a default constructor allows the creation of object instances from this meta-information, by invoking the *newInstance* method.

```
Class.forName(nameClass).newInstance();
```

The use of descriptors to manipulate the state of instances allows for independent control of read and write access. This would not be the case if access to the fused attribute were public. When access to attributes is not direct, the JavaBeans specification calls them **properties**. In addition, the use of descriptors also allows you to add validations, controls or processes associated with one of the descriptors. This is a simple way to add to components the ability to handle events or to set constraints on manipulating the state of objects.

The meta-information that Java maintains during the execution of its applications allows methods to be executed without the programmer having to know beforehand to which class they belong. This is known as introspection or reflection processes. This makes it possible for libraries such as JAXB or JPA to obtain and manipulate the state of any object without much intervention from the programmer.

In summary, we can say that the **main aspects that JavaBeans support** are the following:

- Manipulation of the state of JavaBeans instances through properties, on which access, validation or general control limitations can be described.

- Event management linked to the changes that occur in the JavaBeans instances during executions.

- Introspection processes that enable the creation and customisation of new instances of JavaBeans and the interaction with other components or tools in a totally dynamic way (at runtime).

- Persistence capability. That is to say, it must be possible to store the state of the component in a storage support and be able to retrieve it without losing information during the process. It is necessary that the JavaBeans classes that require persistence implement the *Serializable* interface.

Issues to consider

Although there is a tendency to define JavaBeans only as reusable **GUI components**, this is a biased and restrictive view. Probably, the reason for this confusion has been the fact that graphical components are very widespread horizontal components, highly reusable and easy to integrate in any application, but the rest of Java components would also have to follow the conventions described by JavaBeans.

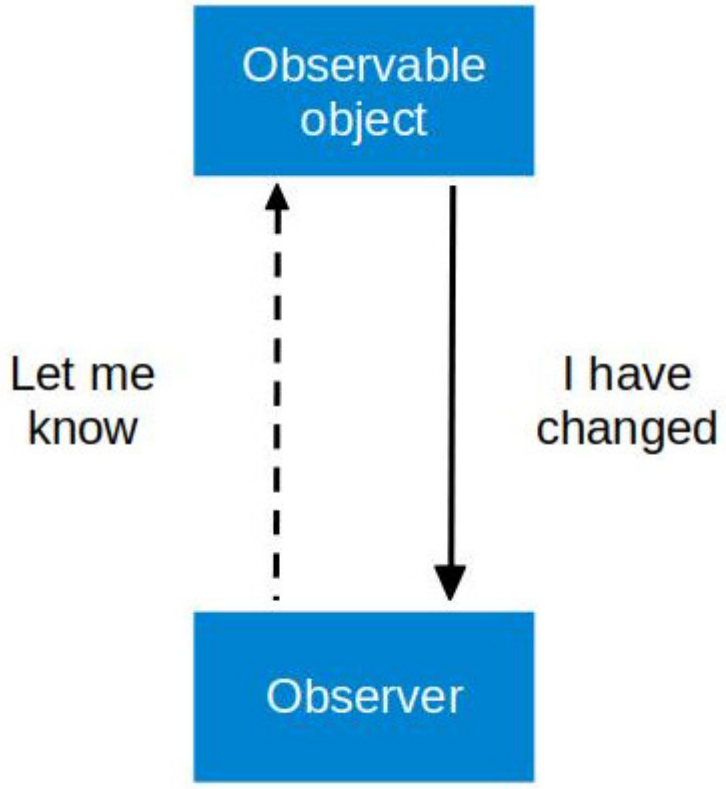
Another common confusion is the one that limits the development of non-graphical JavaBeans to distributed applications. Specifically in the form of **E**nterprise **J**ava**B**eans (**EJB**). The difficulty in developing distributed applications has led Java to make a great effort in specifying components that support them and named these specifications as EJB. The components of the first versions had very specific considerations, and were so inflexible that their use was confined exclusively to distributed applications, but the latest version has managed to break this link by incorporating components used in any type of application but also adapting them to distributed considerations.

These are components that have been developed independently of EJBs, but the success they have achieved has made it necessary to include them in the specification. We are referring, among others, to the data access components, also known as **DAO** (**D**ata **A**ccess **O**bject).

2.1. Events

A digital event is any change that occurs during the execution of a program. Event management attempts to take advantage of these changes to invoke specific procedures. Java has standard classes and interfaces to manage the capture of events as they occur in order to be able to associate conditional processes to certain events.

Event management is based on a software design **pattern** called **Observer** (see figure below). This pattern defines a scenario with two elements, the object to be observed and the observer. The object to be observed is also called the event source because it is there that changes occur and events originate.



How does it work?

In order to observe, the observer will have to signal the observable object to alert it when there is a change. The observer remains passive until it is alerted, at which point it will activate the necessary processes. In practice, this pattern needs a third element, the **event**. This is the element in charge of notifying the observers what change has occurred.

Let us now see how we can implement such a system with Java. The JDK has the

java.util.EventObject class. It must be considered as the base class from which any type of event will derive. It is a class that expects to receive in its constructor the data source of the event, and thus the observer of the event, so that the observer can analyse the changes and act accordingly. If we need to send to each notification other complementary data in addition to the event source, we can implement classes derived from *EventObject* that receive the necessary data during the instantiation.

Implementation in Java

The source of events, or class of observable objects, will be a normal Java class with the particularity that it must have an operation to associate observers and another one to remove them. We will also need to define an interface with a single method that we will use to perform the notifications. Any class that has to act as an observer of the event source will have to implement the interface for compatibility. In order to be able to define a diverse hierarchy, Java has an interface without methods called ***java.util.EventListener***. Although it is not mandatory, it is advisable that all observers belong to the same hierarchy and therefore derive from it.

Event management is very useful for working with components, as it allows components to interact without having to know during coding exactly which components will interact. Event management allows for dynamic allocation. Notice that from the perspective of the data source, it is not necessary to know which class will be the observer. It is enough that it implements the expected interface.

2.2. Property types

JavaBeans identify properties with a name and represent those attributes of state that may have effects on the appearance or behaviour of their instances. There is no physical element in JavaBeans classes that identifies properties. In fact, they are recognised because they match those attributes that have read or write descriptors. Descriptors are usually defined by prefixing the attribute name with ***get*** or ***set***. This convention is very useful to use in environments that work with scripting languages such as Web environments, since knowing the name of the property deduces the name of the descriptors, which can be invoked using introspection.

The use of descriptors makes it possible to control the scope of access regardless of whether it is read access or write access. In addition, by using descriptor methods it is easy to create an event management to control changes to properties.

Before going into detail on the change management of properties using events, we analyse some singularities of properties according to the types of values they store or whether they are simple or indexed values (lists and data vectors).

Simple property

As mentioned above, attribute descriptor names with single values shall be generated by prefixing the attribute name with get or set.

The former represents the method of reading the attribute value:

```
typeAttribute get<NameAttribute>()
```

And the second one, the method of writing the attribute:

```
void set(typeAttribute value)
```

However, if the attribute is specifically of Boolean type, the Java convention indicates that its read descriptor may be prefixed with the particle is instead of get.

Example:

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

    public int getMouthWidth() {
        return mMouthWidth;
    }

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

Indexed property

The convention also allows for a specific generation of multivalent attribute descriptors (arrays or lists), in addition to the classical forms, to support access to each of the elements stored in the multivalent attribute. Indexed access is really important to define when we need individual access to the data in the collection, as it normalises the different types of access. It must be understood that the concept of property is a broad and abstract one, and does not only refer to attribute values. We can also define calculated properties without the need for a direct correspondence with any attribute.

Example:

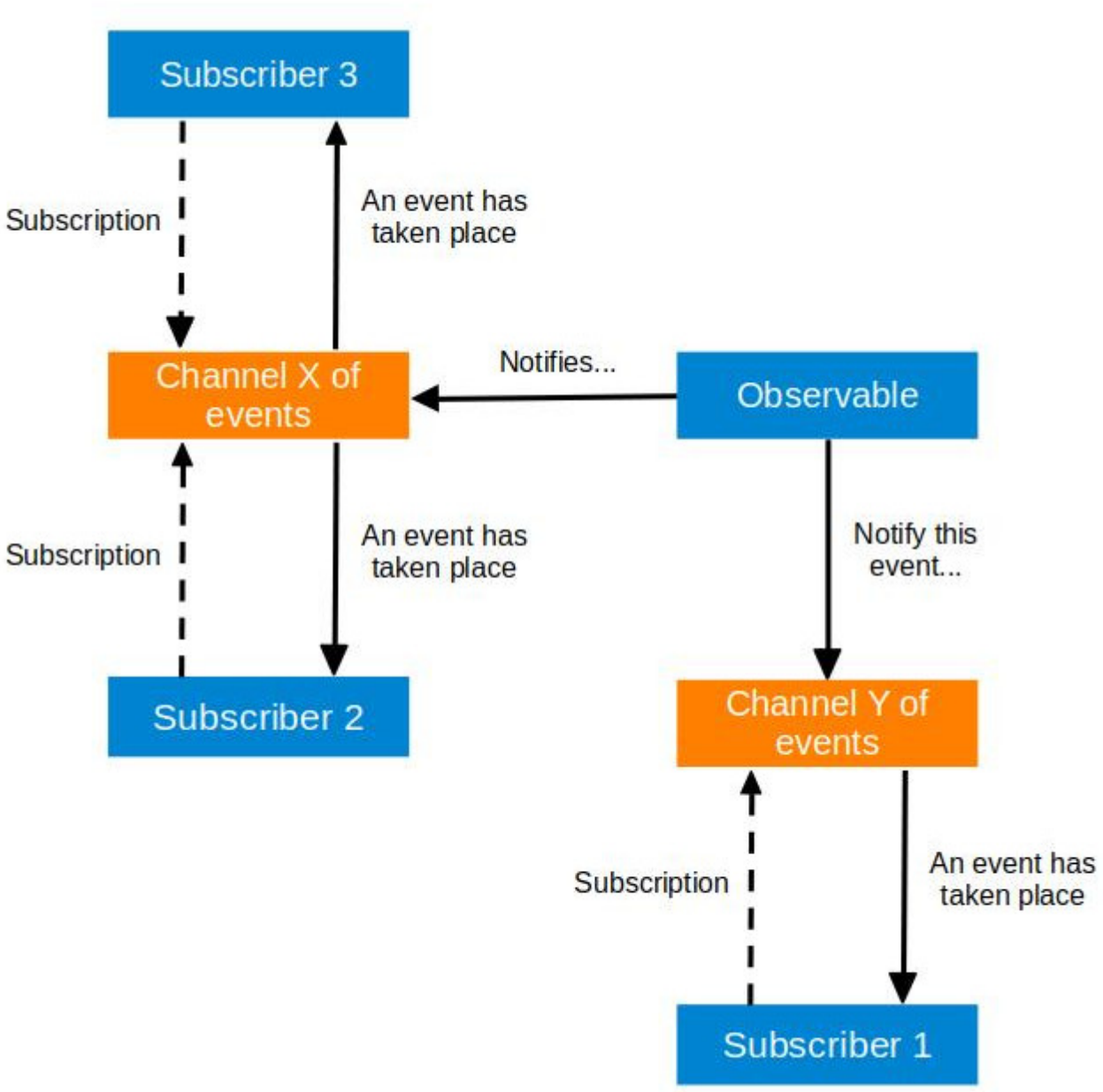
```
public int[] getTestGrades() {
    return mTestGrades;
}

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

2.3. Properties as a source of events

It is important to note the use of descriptor methods in order to define an event management system linked to changes in property values. The Observer pattern can be used, but the standard version of the Java language provides a series of tools that facilitate the construction of event management systems in JavaBeans.

Instead of strictly applying the Observer pattern, a variant called **Publication-subscription** is applied that facilitates the management of multiple events and multiple observers within a single class (see figure below). The Observer pattern is slightly modified by adding a fourth element, the event channel. Thus, the observable object takes the role of event publisher. Observers take the role of subscribers, who subscribe to notifications that arrive exclusively from certain events and not others (event channels).



Event channels play a key role in sorting subscribers. This avoids having to always inform all subscribers, limiting notifications to subscribers associated exclusively with a given channel. The classification into channels basically responds to the type of role that subscribers will play with respect to the observed component. JavaBeans contemplates two types of notifications:

- **Informative**. They have the exclusive mission of informing about the change of a property.
- **Control**. In addition to informing, they allow to decide if the change is coherent and therefore feasible, or on the contrary, if it is incoherent and cannot be carried out.

On the other hand, if the JavaBean has many properties defined, we may be interested in having at least one channel for each property. However, it should be borne in mind that this is not a mandatory condition, since if the component has few properties, it may be easier to have a single channel and make a selection during the launch of the event itself that allows us to decide, depending on the affected property, which processes should be activated.

2.3.1. Types of channels

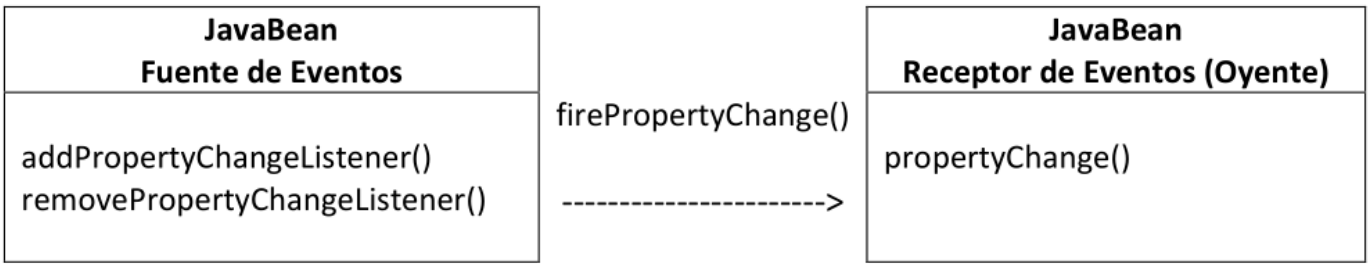
Java standardly contemplates two types of channels linked to JavaBeans. Each channel has a specific type of interface associated with it that subscribers will have to implement.

a) Bound property

One of the channels is implemented by the **PropertyChangeSupport class**. The purpose of this class is to **notify subscribers that there has been a change in one of the properties** so that they can decide what actions to take. The **notification is purely informative**, and actions taken by subscribers will not affect the assignment of the new value.

The associated subscribers in this channel have to implement the *PropertyChangeListener* interface, which has a single method called *propertyChange*, which receives an event of type *PropertyChangeEvent*.

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);
    }
}
```

b) Constrained property

The other type of channel is implemented by the **VetoableChangeSupport class**. This is a channel intended to **support subscribers that can veto** (hence its name) the allocation of the new value, **preventing it from being carried out**.

The subscribers associated in this channel have to implement the *VetoableChangeListener* interface, which has a method called *vetoableChange* that receives an event of type *PropertyChangeEvent*, but unlike the previous one, this method is expected to throw an exception of type *PropertyVetoException* in case of not accepting the change of value of the property.

Example:

```
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);
    }
}
```

2.3.2. Interaction between classes and interfaces

Let's now see how each of these classes and interfaces relate to each other.

We start with the PropertyChangeSupport, PropertyChangeListener and PropertyChangeEvent. The channel constructor has to receive an instance of the event source as a parameter. Since channels are normally instantiated by an event source, it will be necessary to instantiate this type of channel by making a call similar to this one:

```
propertySupport = new PropertyChangeSupport(this);
```

The PropertyChangeSupport channels have a method to associate subscribers. The method is called addPropertyChangeListener and follows the following syntax:

```
void addPropertyChangeListener(PropertyChangeListener listener)
```

It also has a method to unlink them once they are associated. The syntax is similar:

```
void removePropertyChangeListener(PropertyChangeListener listener)
```

The method called *firePropertyChange* shall be used in this channel to indicate that a change has occurred in one of the properties of the JavaBean. The invocation of this method shall force the creation of a PropertyChangeEvent with the data passed as a parameter, followed by the creation of an event of type PropertyChangeEvent with the data passed by parameter, and then the propertyChange method of each subscriber shall be invoked to notify them of the change. No notifications need not be sent by the programmer.

```
propertySupport.firePropertyChange(propertyName, oldValue, newValue);
```

3. 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.

3.1. 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).

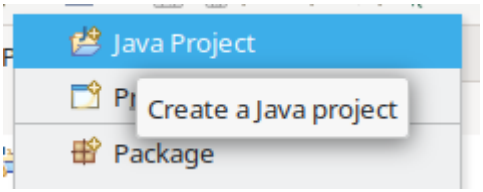
3.2. Part 1. Creating the beans

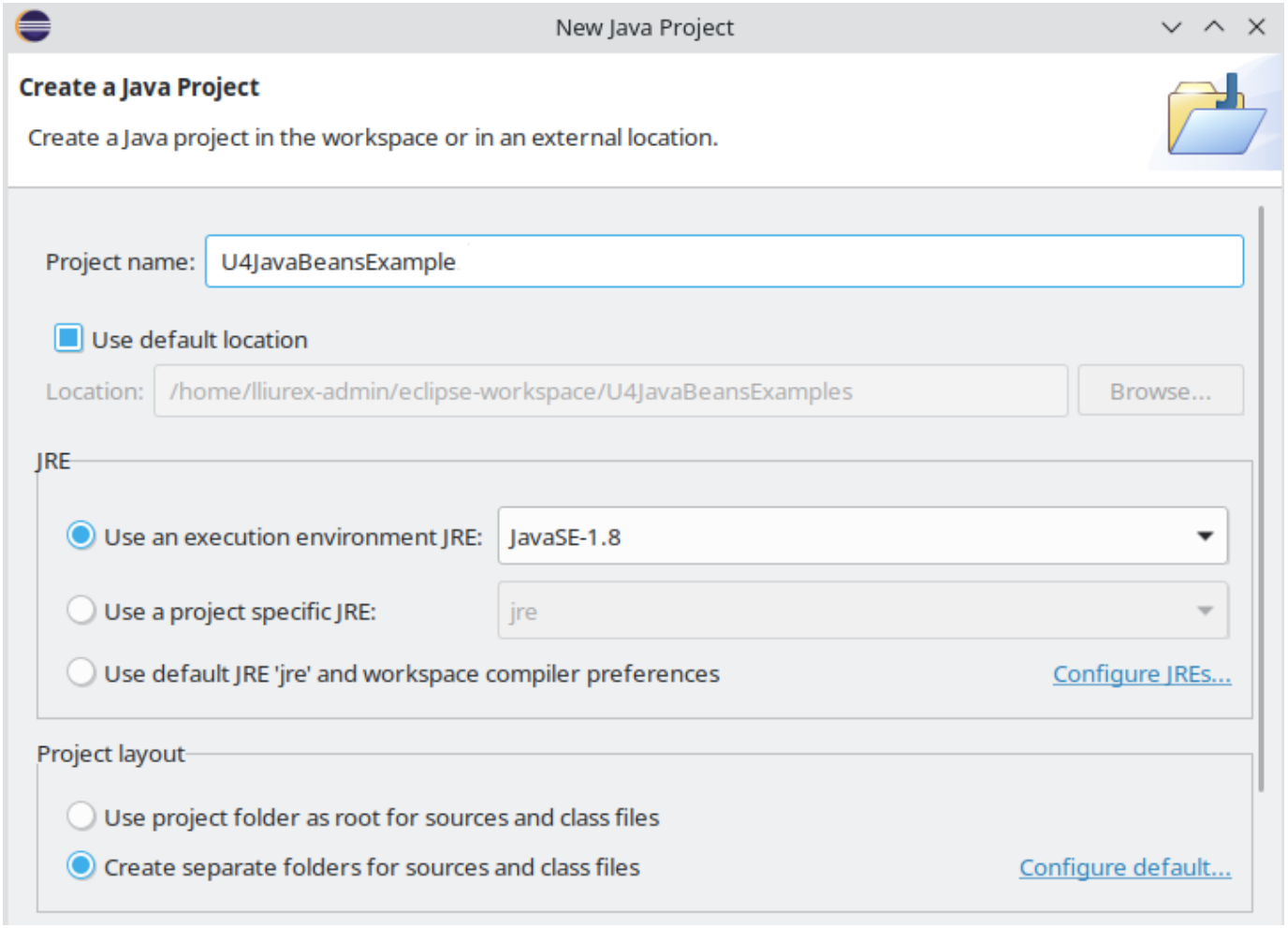
3.2.1. STEP 1.1: Create a project and two classes (beans)

Create a Project and two classes (JavaBeans):

1. Java Project

- Menu **File** →**New** →**Java Project** or press the icon.

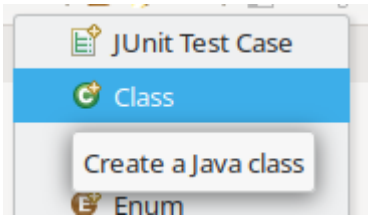


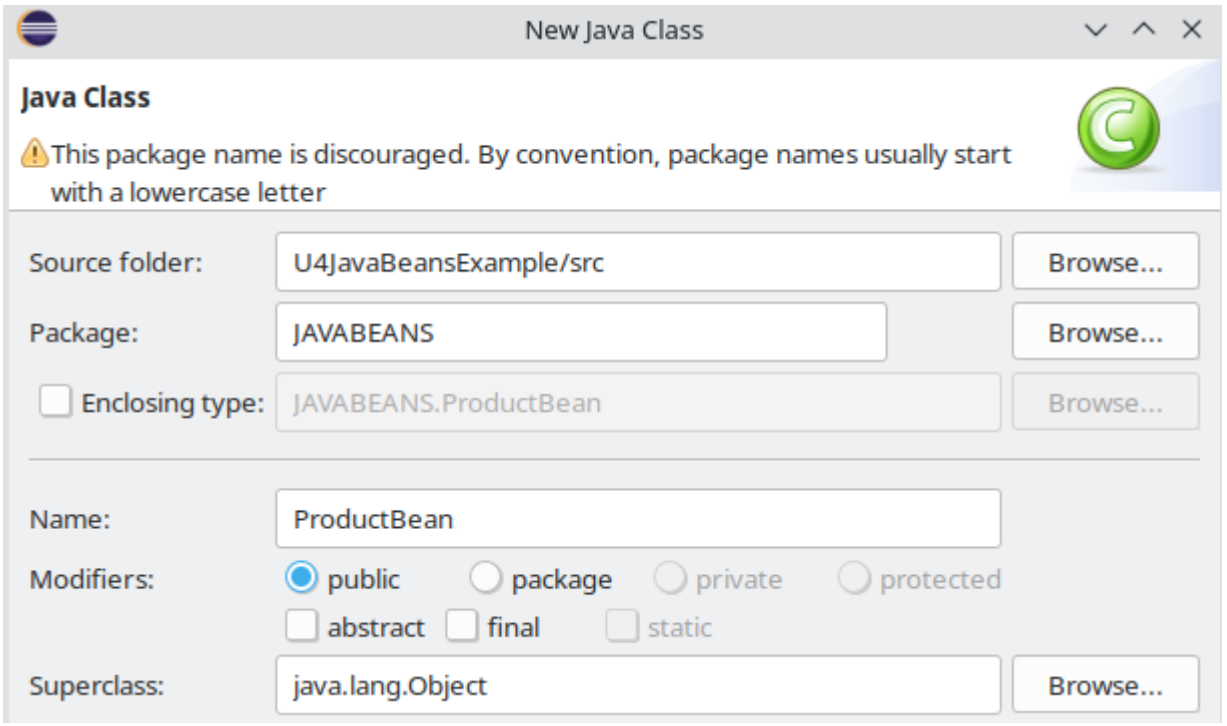


- Click button Finish.

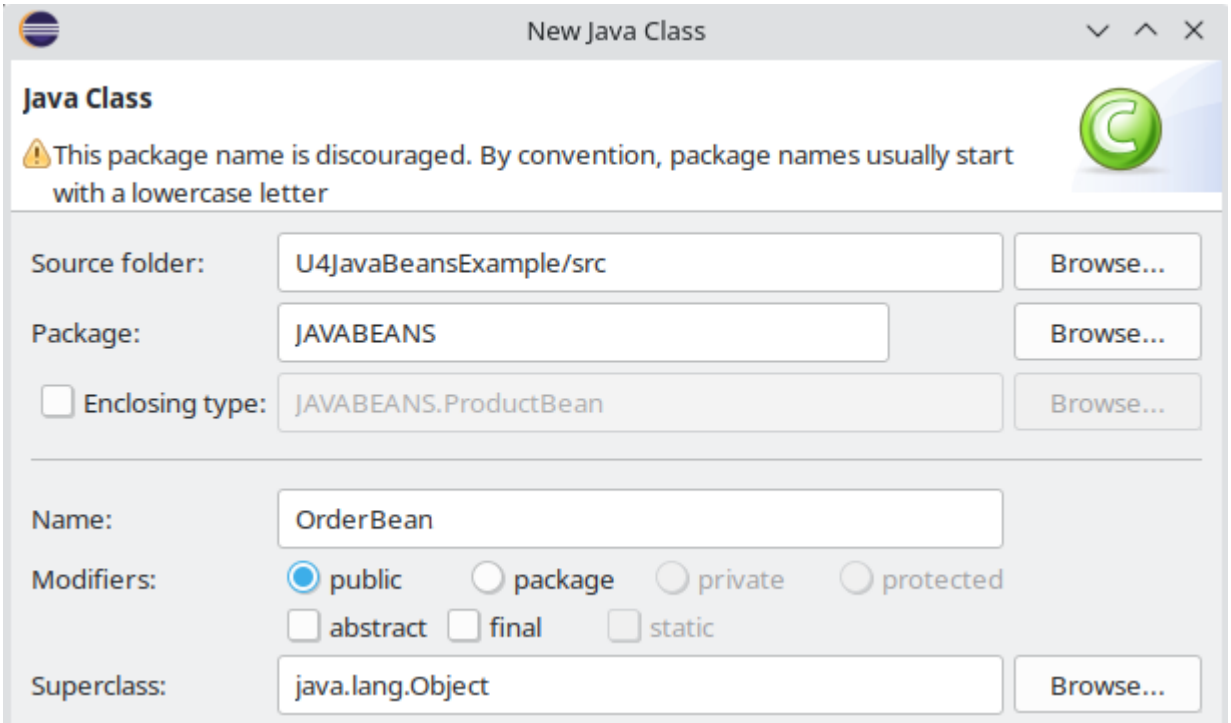
2. Java Class x 2

- Class **ProductBean**. Menu **File** →**New** →**Class** or press the icon.



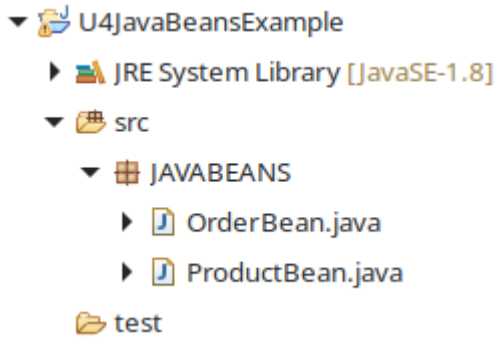


- Click button Finish.
- Class **OrderBean**.



3. Result

- And the final result:



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

The **source bean** must implement the **Serializable** interface.

```
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;
```

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

```
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;
```

3.2.3. 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 current and min stock**.

```
package JAVABEANS;

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

/**
 * ===========================================================================
 * Object ProductBean. SOURCE
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ===========================================================================
 */

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;

    private PropertyChangeSupport propertySupport;

    /*
     * ----------------------------------------
     * METHODS
     * ----------------------------------------
     */
    /*
     * Empty constructor
     */
```

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

    /*
     * Constructor with all fields
     */
    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;
    }

    /*
     * -----------------------------------------
     * GETTERS
     * -----------------------------------------
     */
    public int getiProductID() {
        return iProductID;
    }

    public String getstDescription() {
        return stDescription;
    }

    public float getfPrice() {
        return fPrice;
    }

    /*
     * -----------------------------------------
     * SETTERS
     * -----------------------------------------
     */
    public void setiProductID(int iProductID) {
        this.iProductID = iProductID;
    }

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

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

    /*
     * --------------------------
     * LISTENERS. PROPERTYCHANGE
     * --------------------------
     */
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

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

}
```

3.2.4. 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);
        }
    }
```

3.2.5. 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.

```
package JAVABEANS;

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

/**
 * ============================================================================
 * Object OrderBean. LISTENER
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ============================================================================
 */

/*
* ----------------------------------------
* CLASS DEFINITION, PROPERTIES AND METHODS
* ----------------------------------------
*/
public class OrderBean implements Serializable, PropertyChangeListener {

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

    /*
     * ----------------------------------------
     * METHODS
     * ----------------------------------------
     */
    /*
     * Empty constructor
```

```
     */
    public OrderBean() {

    }

    /*
     * Constructor with all fields
     */
    public OrderBean(int iOrdernumber, int iAmount, ProductBean objProductBean) {
        this.iOrdernumber = iOrdernumber;
        this.objProductBean = objProductBean;
        this.iAmount = iAmount;
    }

    /*
     * ----------------------------------------
     * GETTERS
     * ----------------------------------------
     */
    public int getiOrderNumber() {
        return iOrdernumber;
    }

    public int getiAmount() {
        return iAmount;
    }

    public ProductBean getobjProductBean() {
        return this.objProductBean;
    }

    /*
     * ----------------------------------------
     * SETTERS
     * ----------------------------------------
     */
    public void setiOrderNumber(int iOrdernumber) {
        this.iOrdernumber = iOrdernumber;
    }

    public void setiAmount(int iAmount) {
        this.iAmount = iAmount;
    }

    public void setobjProductBean(ProductBean objProductBean) {
        this.objProductBean = objProductBean;
```

```
    }
```

3.2.6. 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.

```
/*
     * -------------------------
     * LISTENERS. EVENTS RAISED
     * -------------------------
     */
    public void propertyChange(PropertyChangeEvent pceEvent) {
        if (pceEvent.getPropertyName().equals("currentStockBelowMinStock"))
        {
            // Current stock is below minimum
            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", pceEvent.getOldValue());
            System.out.printf("=> New current Stock: %d%n", pceEvent.getNewValue());
            System.out.printf("It will place an order for this product: %s%n",
                objProductBean.getstDescription());
        }
        if (pceEvent.getPropertyName().equals("minStockRaisedOverCurrentStock"))
        {
            // MIN stock has been raised over current stock
            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", pceEvent.getOldValue());
            System.out.printf("New minstock Stock: %d%n", pceEvent.getNewValue());
            System.out.printf("It will place an order for this product: %s%n",
                objProductBean.getstDescription());
        }
    }
```

3.3. Part 2. Running the beans

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

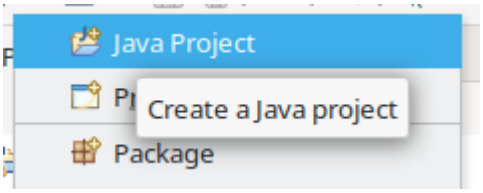
1. First, we're saving our beans in a jar file (**clean & build**).
2. Then, we're importing that JAR file (library) in a **new class**.
3. Finally, we're **running a piece of code** to watch both beans interact.

3.3.1. STEP 2.1: Create and import JAR file

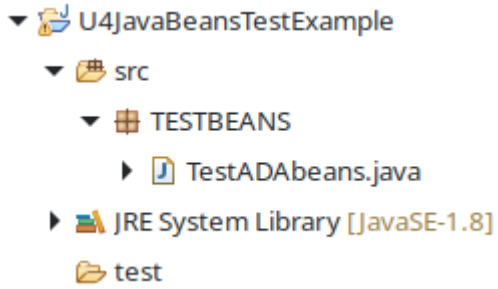
Create a Project, a Java Class and import our beans:

1. Create new Java Project

- Menu **File** →**New** →**Java Project** or press the icon.

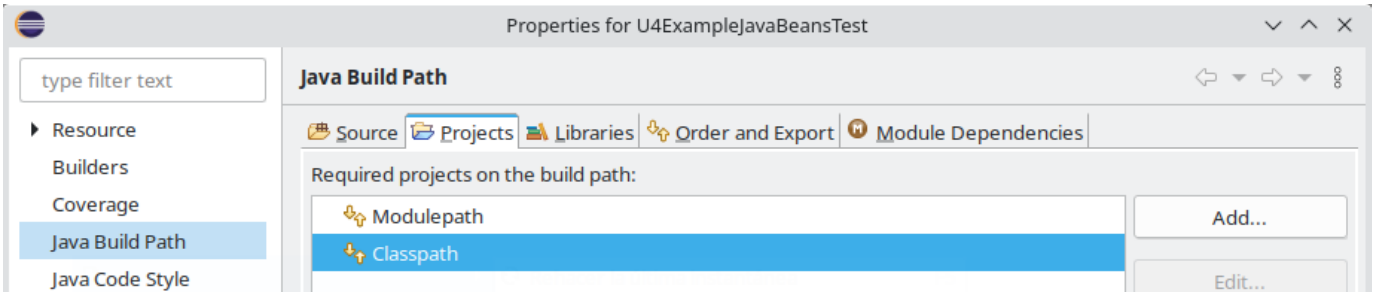


- Click button Finish.
- Class **TestADAbeans**. Menu **File** →**New** →**Class** or press the icon. Click button Finish.



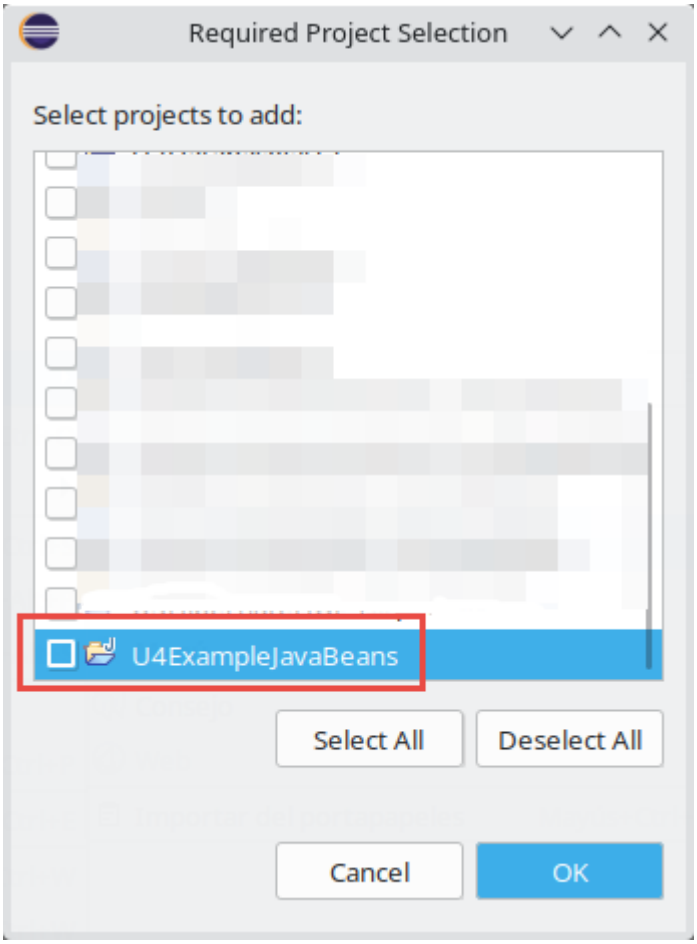
2. Configure build path

- Right button on the project's name and option **Build path** →**Configure Build path**.

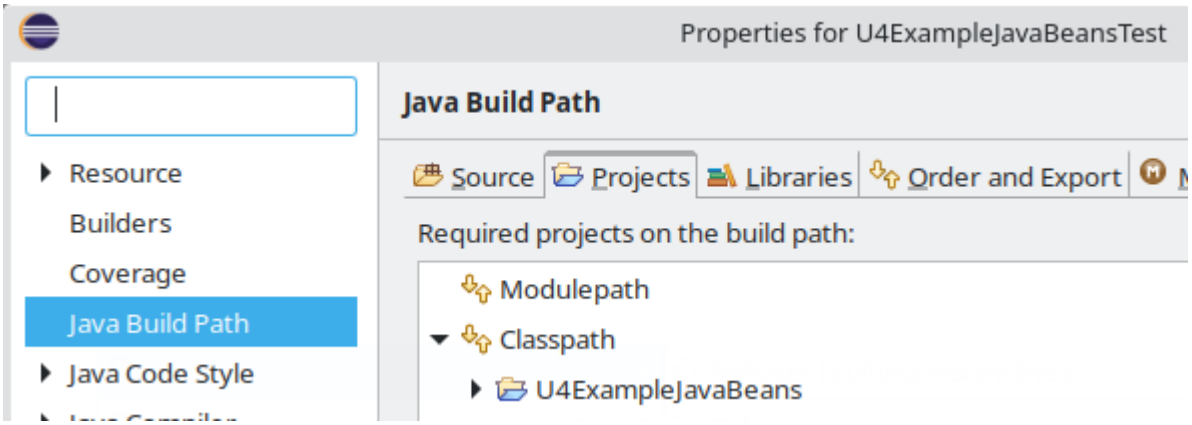


3. Import JAR file

- Click button **Add** and select the previous project:



- Click button OK.



- Click button **Apply and close**.

3.3.2. 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).

```
package TESTBEANS;

import JAVABEANS.*;


/**
 * ====================================================================
 * Main programme to interact among JavaBeans
 * @author Abelardo Martínez. Based and modified from Sergio Badal
 * ====================================================================
 */

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);
    }
}
```

And the 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. Bibliography

Sources

- 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
- Perforce. What Is Component Based Development? https://www.perforce.com/blog/vcs/component-based-development
- 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/
- JavaTPoint. POJO. https://www.javatpoint.com/pojo-in-java