

# Object - Oriented Programming & Design

## Part XVII - (Enterprise) Java Beans

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

# JavaBeans

---

A JavaBean, like an ActiveX control, is a fully tested, reusable component ...

- A *JavaBean* is an object (that comes bundled with other supporting objects in a `.jar` file) that adheres to the *JavaBean* standard. Note that the term, *JavaBean*, is used above with two different meanings.
- The official definition from Sun Microsystems: “A Java Bean is a reusable software component that can be manipulated visually in a builder tool.”
- JavaBeans are used primarily on the client side of a distributed application. They are often visual components, like, say, for a Spreadsheet or Calendar widget. But they can be “invisible” and still be quite useful, like, say for an OO proxy for a klunky old legacy system, or a PGP Bean, providing encryption services.
- Beans use the Java Reflection infrastructure to allow the Bean’s properties to be determined at “customization time” (A.K.A. design and deployment time) by a customization tool (such as JavaSoft’s “Bean Box”).

# JavaBeans

---

In order for a Java object to be a JavaBean, the JavaBean standard specifies:

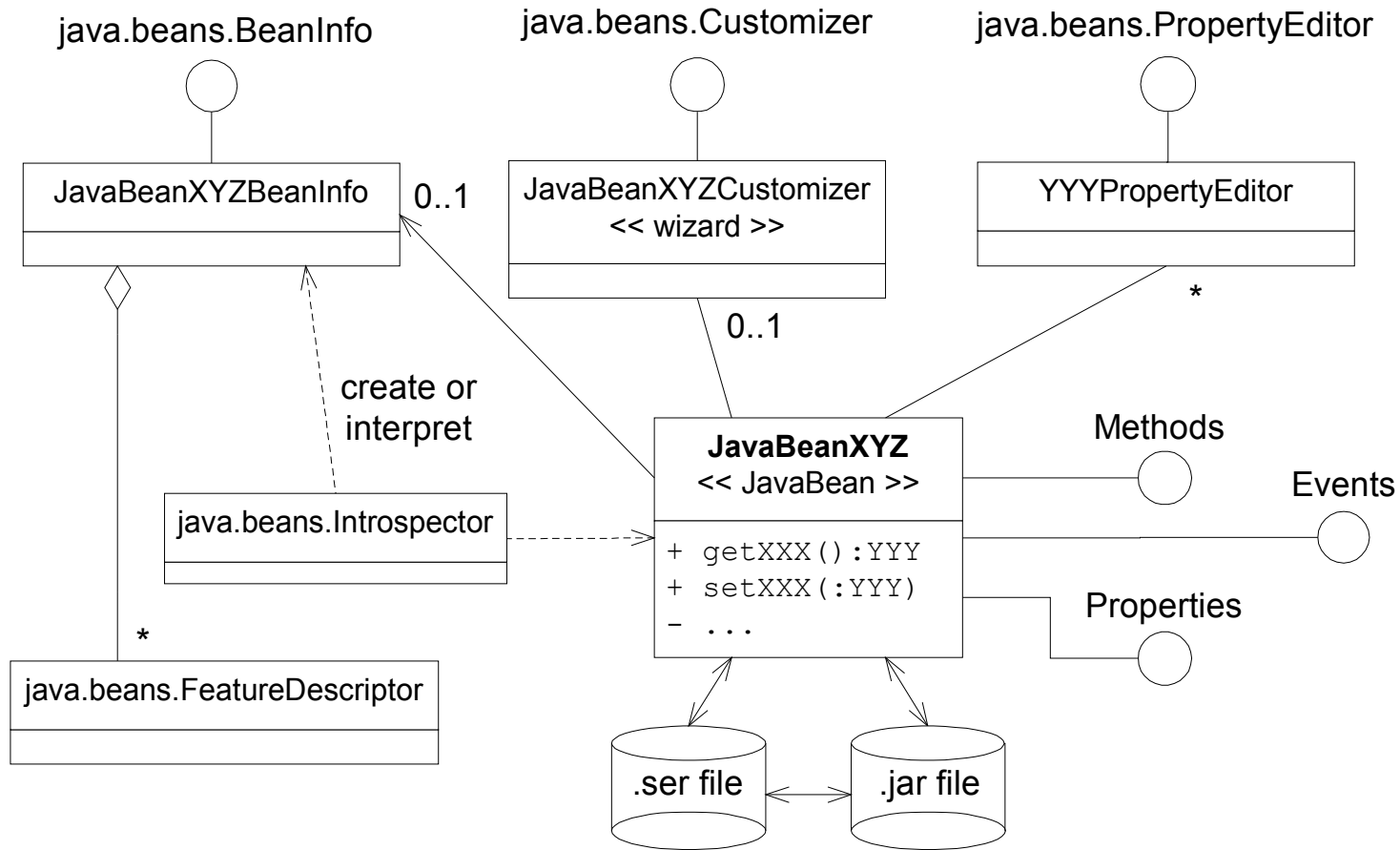
- The Bean's configurable properties have public “getter and setter” methods. For example, the property “String foo” must have the methods `getFoo()` and `setFoo()`. (This naming convention is referred to as “design patterns”).
- This Bean's companion *Introspector* determines the configurable properties using Java's Reflection API. The Introspector constructs a *property sheet* for the Bean, which can be edited using visual tools at design time.
- The Bean might have a companion *BeanInfo* object that describes the Bean's configurable properties; if a BeanInfo object exists, the Introspector will use it instead of using reflection.
- The Bean might have a companion *Customizer* object, which is Java code (adhering to a standard interface) designed to assist the user with Bean customization at design time. Within the Bean Box tool, the Customizer acts like a “wizard” to guide the bean configuration use case and/or constrain the range of possible values for the Bean's properties.

# JavaBeans

---

- The fact that the Bean can be customized at design /deployment time implies that the Bean must have a means to “remember” the values for all of the customizable properties. This is accomplished using *Serialization*. A persistent Bean is saved to a `.ser` (Serialized) file, deployed with the Bean inside of a `.jar` (Java Archive) file. This is an example of the *Memento* design pattern.
- Individual Bean properties can have their own custom PropertyEditors. For example, a Color object is often described by three integers representing red, blue, and green; a visual ColorEditor can be provided.
- Bean properties may be “indexed”, “bound” or “constrained”... *Indexed* properties have array values, and it is expected that the Bean provides methods to get and set individual elements of the array. *Bound* properties, when updated, will cause the appropriate Bean *Observers* to get notified. These observers have the right to “veto” proposed updates of *constrained* properties.

# JavaBeans



# JavaBeans

---

To summarize, JavaBeans have the following core features:

- Ability to be manipulated visually in a builder tool.
- Introspection, to allow the Bean's properties to be determined by the tool.
- Customization, of both appearance and behavior.
- Event handling, using the *Observer* design pattern (listener model).
- Properties, for customization and other programmatic uses.
- Persistence, allowing the Bean's state to be stored for later retrieval.

Design-time customization code is NOT deployed with a run-time Bean.

# Enterprise Java Beans Framework

---

- Powerful *Framework* for building fast, scalable, and secure servers.
- Configured with an XML deployment descriptor.
- Part of the Java *Enterprise Edition*, along with
  - Remote Method Invocation (RMI)
  - Servlets / Java Server Pages (JSP) / Java Server Faces (JSF)
  - Java Naming and Directory Interface (JNDI)
  
- EJB 1.0 was klunky and not well liked.
- EJB 2.0 was klunky but significantly better.
- EJB 3.0 is good ☺

# EJB Framework

---

- Good for creating server-side applications that are:
  - Scalable
  - Transactional
  - Multi-user
  - Secure
  - Distributed
- Services provided by an EJB container:
  - Persistence
  - Security
  - Transactions
  - Threading
  - Exception Handling
- Provides a consistent component architecture for creating distributed n-tier middleware and applications.

# Enterprise Java Beans (EJBs)

---

EJBs are similar to regular Java Beans in that they are designed to be configurable components.

- Java Beans are components that can be used on any tier of an application.
- EJBs are server-based components.

There are 3 types of EJBs:

- Session
- Entity
- Message Driven

# Session Beans

---

- Beans that represent *mediators, commands, or controllers*
- Define the scope of transactions involving multiple entity beans
- Two basic types:
  - **Stateless**
    - » Have no internal state
    - » Because of the fact that they are stateless, they can be pooled to service multiple clients
  - **Stateful**
    - » Possess internal states
    - » There can be only one Stateful Session Bean per EJB Client
    - » Must be thread safe

# Entity Beans

---

- Entity Beans are business domain objects
- Can be a POJO (Plain Old Java Object) *annotated* with `@Entity`
- Stateful
- Persistent
  - can survive system shutdowns
  - can be shared by multiple EJB Clients
- There are two types of entity bean persistence
  - Container-Managed Persistence (CMP)
  - Bean-managed persistence – the Bean has code to map itself to some persistent storage, usually using JDBC.

`javax.persistence.EntityManager` has an interface to find, create and remove entity beans from the data store.

# Container-Managed Persistence (CMP)

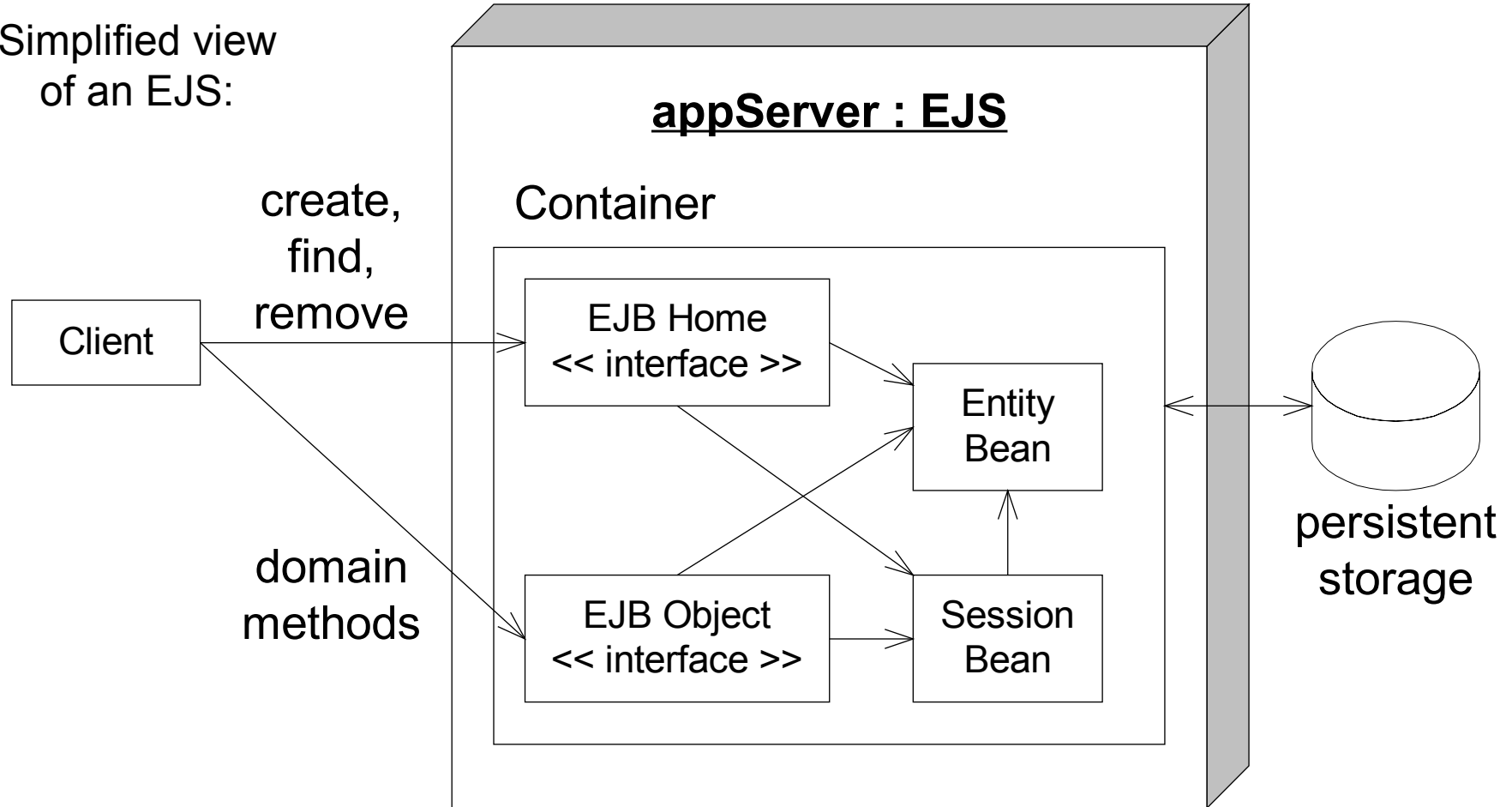
---

The EJB container is responsible for saving the Bean's state.

- The container generates all the database calls (or however it chooses to manage the bean's persistence).
  - The implementation is independent of the data source.
- 
- The EJB 3.0 CMP implementation is nearly identical to work done on the *Hibernate* project. See also the *Spring* framework.
  - May be configured using XML or annotations (JDK 1.5+).
  - CMP using annotations is a huge leap forward in simplifying the code required to implement object to relational mapping.
  - Further detail is beyond the scope of this course.

# Basic EJB 2.0 Architecture

Simplified view  
of an EJS:



# The Spring Framework

---

The Spring Framework integrates with EJB 3.0. Check it out.

**Spring** provides the following sub-frameworks (and more):

1. **Inversion of Control (IoC)** container – configures application components (using a technique known as *dependency injection*), and manages the lifecycles of Java objects (called beans).
2. **Aspect-oriented programming (AOP)** – for cross-cutting concerns.
3. Data access – significant support for **object-relational mapping**, including transaction management (look into *Hibernate*).
4. HTTP, Servlet, RPC, RMI, Web Services, SOAP.
5. Message queues using JMS (Java Messaging Service).
6. Facilitates testing.

# Aspect-Oriented Programming (AOP)

---

AOP provides strict separation between business logic and nonfunctional code.

The business classes do not know that they are wrapped by additional functionality. Think of an aspect as a flexible and configurable *decorator* that can be applied to methods in a declarative manner (with XML or annotations).

EJB3 containers come with built-in, highly reusable aspects.

Example cross-cutting concerns: Transactions, exception handling, logging, security, and persistence. “Remoting” can also be an aspect.

Session beans are the natural place for the use of AOP.

Java EE aspects are called interceptors.

# Example EJB3 interceptor

---

```
public class MethodMonitorInterceptor {
    @AroundInvoke
    public Object trace( InvocationContext invocationContext )
                        throws Exception {
        System.out.println( "" + Clock.time() + " : " +
                            invocationContext.getMethod() );
        return invocationContext.proceed();
    }
}

// In another file, a bean:

@Stateless
@Interceptors( MethodMonitorInterceptor.class )
public class HelloWorldBean {
    public String hi() { return "Hello World"; }
}
```