JavaBeans Component Technology in Java

Introduction

JavaBeans makes it easy to reuse software components. Developers can use software components written by others without having to understand their inner workings. To understand why software components are useful, think of a worker assembling a car. Instead of building a radio from scratch, for example, she simply obtains a radio and hooks it up with the rest of the car.

In Object Oriented Programming an object represents a person, place or thing that has attributes and functions. These objects are implemented in Java as Java Beans. A Java Bean acts as a container for all of the related attributes and functions associated with the object. Java Beans make it easy to pass all of the related data associated with the object in single argument when calling function. This is particularly important when making a Remote Procedure Call (RPC) to a function on a different server over the network.

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between an application client or applet and components running on the Java EE server, or between server components and a database. JavaBeans components are not considered Java EE components by the Java EE specification.

JavaBeans components have properties and have get and set methods for accessing the properties. JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture. Let's start with an initial definition and then refine it:

> "***A Java Bean is a reusable software component that can be created and manipulated in a software development tool.***"

Individual Java Beans will vary in the functionality they support, but the **typical unifying features of a Java Bean are**:

- Support for *introspection* so that a software development tool can analyze how a bean works.
- Support for *customization* so that when using a software development tool a user can customize the appearance and behavior of a bean.
- Support for *events* as a simple communication metaphor than can be used to connect up beans.
- Support for *properties*, both for customization and for programmatic use.
- Support for *persistence*, so that a bean can be customized in a software development tool and then have its customized state saved away and reloaded later.

Writing JavaBeans

Writing beans assumes following certain coding conventions, which enables tools that use beans to recognize and use your beans. **Java Beans should follow the following rules**:

1) They should implement the java.io.Serializable interface, which allows sending it over the network, or reading and writing to a file.
2) They should implement a default constructor.
3) All variables should be accessed via public "get" and "set" methods. All variables should be declared with private modifier, enabling encapsulation of data items and making Java Bean more modular. In this way Java Bean maintains the integrity of the data.
4) Override the default constructor with a Constructor that sets the initial values of variables, which is used for instantiating and initializing a Java Bean object.
5) Override the **`equals()`** method for comparing the contents of two objects for equality.
6) Override the **`hashCode()`** function to support sorting and searching of Java Bean objects stored in collections.

Serialization of JavaBeans

Serializability of a class is enabled by the class implementing the **`java.io.Serializable`** interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

Serialization in java is implemented by **`ObjectInputStream`** and **`ObjectOutputStream`**, so all we need is a wrapper over them to either save it to file or send it over the network. Static variable values are not serialized since they belong to class and not object. The serialization mechanism automatically detects references to other objects. As long as the "sub-objects" are also serializable, **`ObjectOutputStream`** serializes them and includes them in the stream.

Java serialization process is done automatically. Sometimes we want to obscure the object data to maintain its integrity. We can do this by implementing **`java.io.Externalizable`** interface and provide implementation of **`writeExternal()`** and **`readExternal()`** methods to be used in serialization process. Notice that order of writing and reading the extra data to the stream should be same. We can put some logic in reading and writing data to make it secure.

Literature and Links

[1] **Trail: JavaBeans(TM)**. https://docs.oracle.com/javase/tutorial/javabeans/index.html

[2] **JavaBeans. Version 1.01-A**. Sun Microsystems. 1997.

[3] **Class Object**. https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

[4] **public interface Serializable**. https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

[5] **public interface Externalizable extends Serializable**. https://docs.oracle.com/javase/7/docs/api/java/io/Externalizable.html

[6] **Serialization in Java – Java Serialization**. https://www.journaldev.com/2452/serialization-in-java