OROMIA STATE UNIVERSITY
COLLEGE OF SCIENCE AND TECHNOLOGY
DEPARTEMENTOFINFORMATI ON TECHNOLOGY

Network Programming
Chapter-4
Java Native Interface (JNI)
By: Yigermal S.(M.Tech)

As we know, one of the main strengths of Java is its portability – meaning that once we write and compile code, the result of this process is platform-independent bytecode. Simply put, this can run on any machine or device capable of running a Java Virtual Machine, and it will work as seamlessly as we could expect. Dear students! What is JNI?

The Java Native Interface (JNI) is a framework that allows your Java code to call native applications and libraries written in languages such as C, C++ and Objective-C. To be honest, if you have any other choice besides using JNI, do that other thing. Dealing with native libraries is a huge source of pain when supporting Java applications. With that said, sometimes you just can't avoid using JNI. This article outlays the relatively simple steps needed to write a JNI bridge. However, sometimes **we do actually need to use code that's natively-compiled for a specific architecture**.

There could be some reasons for needing to use native code:

- The need to handle some hardware
- Performance improvement for a very demanding process
- An existing library that we want to reuse instead of rewriting it in Java.

To achieve this, the JDK introduces a bridge between the bytecode running in our JVM and the native code (usually written in C or C++).

The tool is called Java Native Interface. In this Lecture, we'll see how it is to write some code with it.

2. How It Works

2.1. Native Methods: The JVM Meets Compiled Code

Java provides the *native* keyword that's used to indicate that the method implementation will be provided by a native code.

Normally, when making a native executable program, we can choose to use static or shared libs:

- Static libs – all library binaries will be included as part of our executable during the linking process. Thus, we won't need the libs anymore, but it'll increase the size of our executable file.
- Shared libs – the final executable only has references to the libs, not the code itself. It requires that the environment in which we run our executable has access to all the files of the libs used by our program.

The latter is what makes sense for JNI as we can't mix bytecode and natively compiled code into the same binary file.

Therefore, our shared lib will keep the native code separately within its *.so*/*.dll*/*.dylib* file (depending on which Operating System we're using) instead of being part of our classes.

The *native* keyword transforms our method into a sort of abstract method:

```
1    private native void aNativeMethod();
```

With the main difference that **instead of being implemented by another Java class, it will be implemented in a separated native shared library**.

A table with pointers in memory to the implementation of all of our native methods will be constructed so they can be called from our Java code.

2.2. Components Needed

Here's a brief description of the key components that we need to take into account. We'll explain them further later in this article

- Java Code – our classes. They will include at least one *native* method.
- Native Code – the actual logic of our native methods, usually coded in C or C++.
- JNI header file – this header file for C/C++ (*include/jni.h* into the JDK directory) includes all definitions of JNI elements that we may use into our native programs.
- C/C++ Compiler – we can choose between GCC, Clang, Visual Studio, or any other we like as far as it's able to generate a native shared library for our platform.

2.3. JNI Elements in Code (Java And C/C++)

Java elements:

- “native” keyword – as we've already covered, any method marked as native must be implemented in a native, shared lib.
- *System. loadLibrary(String libname)* – a static method that loads a shared library from the file system into memory and makes its exported functions available for our Java code.

C/C++ elements (many of them defined within *jni.h*)

- JNIEXPORT- marks the function into the shared lib as exportable so it will be included in the function table, and thus JNI can find it
- JNICALL – combined with *JNIEXPORT*, it ensures that our methods are available for the JNI framework

- JNIEnv – a structure containing methods that we can use our native code to access Java elements
- JavaVM – a structure that lets us manipulate a running JVM (or even start a new one) adding threads to it, destroying it, etc…

3. Hello World JNI

Next, **let's look at how JNI works in practice.**

In this tutorial, we'll use C++ as the native language and G++ as compiler and linker.

We can use any other compiler of our preference, but here's how to install G++ on Ubuntu, Windows, and MacOS:

- Ubuntu Linux – run command *"sudo apt-get install build-essential"* in a terminal
- Windows – Install MinGW
- MacOS – run command *"g++"* in a terminal and if it's not yet present, it will install it.

3.1. Creating the Java Class

Let's start creating our first JNI program by implementing a classic "Hello World".

To begin, we create the following Java class that includes the native method that will perform the work:

```
1  package com.baeldung.jni;
2
3  public class HelloWorldJNI {
4
5      static {
6          System.loadLibrary("native");
      }
```

```
7
8           public static void main(String[] args) {
9               new HelloWorldJNI().sayHello();
10          }
11
12          // Declare a native method sayHello() that receives no arguments and returns v
13          private native void sayHello();
14      }
15
```

As we can see, **we load the shared library in a static block**. This ensures that it will be ready when we need it and from wherever we need it.Alternatively, in this trivial program, we could instead load the library just before calling our native method because we're not using the native library anywhere else.

4. Disadvantages Of Using JNI

JNI bridging does have its pitfalls.

The main downside being the dependency on the underlying platform; **we essentially lose the "write once, run anywhere"** feature of Java. This means that we'll have to build a new lib for each new combination of platform and architecture we want to support. Imagine the impact that this could have on the build process if we supported Windows, Linux, Android, MacOS…

JNI not only adds a layer of complexity to our program. **It also adds a costly layer of communication** between the code running into the JVM and our native code: we need to convert the data exchanged in both ways between Java and C++ in a marshaling/unmarshaling process.

Sometimes there isn't even a direct conversion between types so we'll have to write our equivalent.

JavaBean

A JavaBean is a Java class that should follow the following conventions:

- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods.

Why use JavaBean?

According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Simple example of JavaBean class

```
1. //Employee.java
2.
3. package mypack;
4. public class Employee implements java.io.Serializable{
5. private int id;
6. private String name;
7. public Employee(){}
8. public void setId(int id){this.id=id;}
9. public int getId(){return id;}
10. public void setName(String name){this.name=name;}
11. public String getName(){return name;}
12. }
```

How to access the JavaBean class?

To access the JavaBean class, we should use getter and setter methods.

```
1. package mypack;
2. public class Test{
3. public static void main(String args[]){
4. Employee e=new Employee();//object is created
5. e.setName("Arjun");//setting value to the object
```

```
6. System.out.println(e.getName());
7. }}
```

Note: There are two ways to provide values to the object. One way is by constructor and second is by setter method.

JavaBean Properties

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define.

A JavaBean property may be read, write, read-only, or write-only. JavaBean features are accessed through two methods in the JavaBean's implementation class:

1. getPropertyName ()

For example, if the property name is firstName, the method name would be getFirstName() to read that property. This method is called the accessor.

2. setPropertyName ()

For example, if the property name is firstName, the method name would be setFirstName() to write that property. This method is called the mutator.

Advantages of JavaBean

The following are the advantages of JavaBean:/p>

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

Disadvantages of JavaBean

The following are the disadvantages of JavaBean:

- JavaBeans are mutable. So, it can't take advantages of immutable objects.
- Creating the setter and getter method for each property separately may lead to the boilerplate code.

jsp:useBean action tag

The jsp:useBean action tag is used to locate or instantiate a bean class. If bean object of the Bean class is already created, it doesn't create the bean depending on the scope. But if object of bean is not created, it instantiates the bean.

Syntax of jsp:useBean action tag

1. <jsp:useBean id= "instanceName" scope= "page | request | session | appli cation"
2. **class**= "packageName.className" type= "packageName.className"
3. beanName="packageName.className | <%= expression >" >
4. </jsp:useBean>

Attributes and Usage of jsp:useBean action tag

1. **id:** is used to identify the bean in the specified scope.
2. **scope:** represents the scope of the bean. It may be page, request, session or application. The default scope is page.
    - **page:** specifies that you can use this bean within the JSP page. The default scope is page.
    - **request:** specifies that you can use this bean from any JSP page that processes the same request. It has wider scope than page.
    - **session:** specifies that you can use this bean from any JSP page in the same session whether processes the same request or not. It has wider scope than request.
    - **application:** specifies that you can use this bean from any JSP page in the same application. It has wider scope than session.
3. **class:** instantiates the specified bean class (i.e. creates an object of the bean class) but it must have no-arg or no constructor and must not be abstract.
4. **type:** provides the bean a data type if the bean already exists in the scope. It is mainly used with class or beanName attribute. If you use it without class or beanName, no bean is instantiated.
5. **beanName:** instantiates the bean using the java.beans.Beans.instantiate() method.

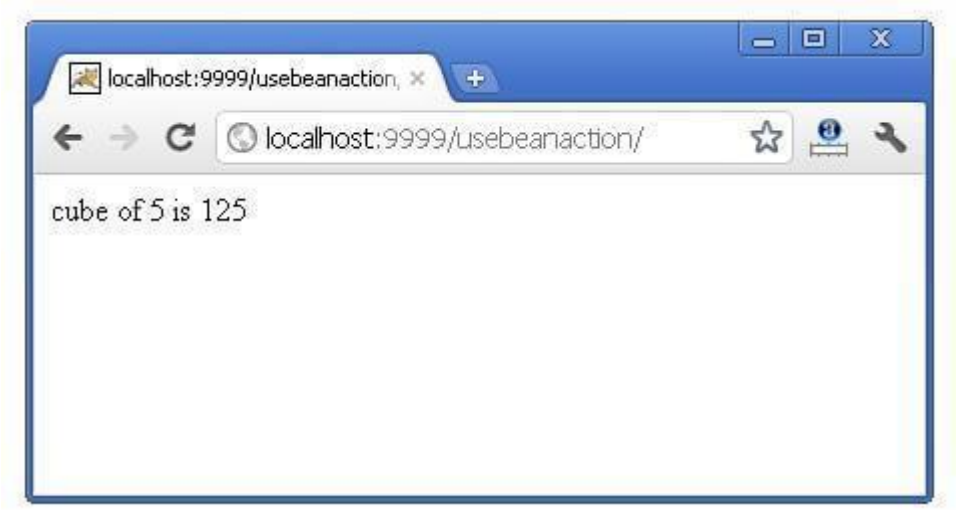
alculator.java (a simple Bean class)

```
1. package com.javatpoint;
2. public class Calculator{
3.
4. public int cube(int n){return n*n*n;}
5.
6. }
```

index.jsp file

```
1. <jsp:useBean id="obj" class="com.Calculator"/>
2.
3. <%
4. int m=obj.cube(5);
5. out.print("cube of 5 is "+m);
6. %>
```

download this example



jsp:setProperty and jsp:getProperty action tags

The setProperty and getProperty action tags are used for developing web application with Java Bean. In web devlopment, bean class is mostly used because it is a reusable software component that represents data.

The jsp:setProperty action tag sets a property value or values in a bean using the setter method.

Syntax of jsp:setProperty action tag

```
1. <jsp:setProperty name="instanceOfBean" property= "*"   |
2. property="propertyName" param="parameterName"  |
3. property="propertyName" value="{ string | <%= expression %>}"
4. />
```

Example of jsp:setProperty action tag if you have to set all the values of incoming request in the bean

```
<jsp:setProperty name="bean" property="*" />
```

Example of jsp:setProperty action tag if you have to set value of the incoming specific property

```
<jsp:setProperty name="bean" property="username" />
```

Example of jsp:setProperty action tag if you have to set a specific value in the property

```
<jsp:setProperty name="bean" property="username" value="Kumar" />
```

---

jsp:getProperty action tag

The jsp:getProperty action tag returns the value of the property.

Syntax of jsp:getProperty action tag

```
1. <jsp:getProperty name="instanceOfBean" property="propertyName" />
```

Simple example of jsp:getProperty action tag

```
1. <jsp:getProperty name="obj" property="name" />
```

Example of bean development in JSP

In this example there are 3 pages:

- index.html for input of values
- welocme.jsp file that sets the incoming values to the bean object and prints the one value

- User.java bean class that have setter and getter methods

index.html

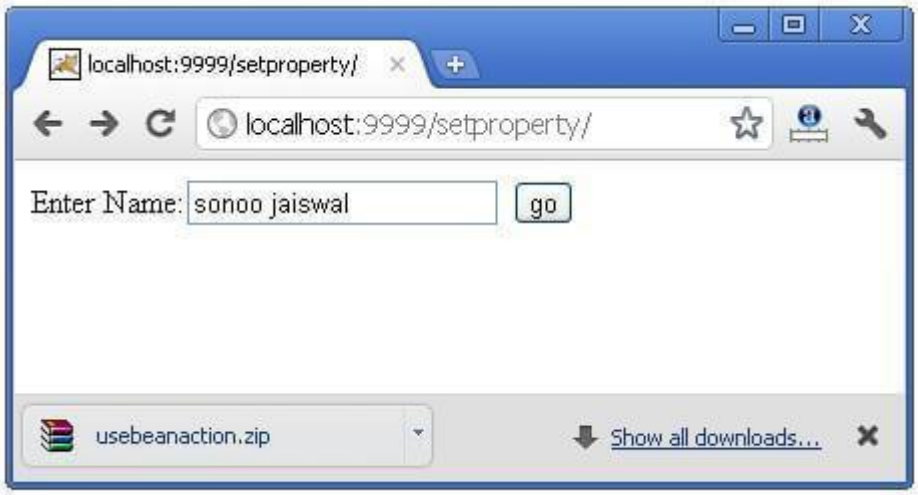
```
<form action="process.jsp" method="post">
Name:<input type="text" name="name"><br>
Password:<input type="password" name="password"><br>
Email:<input type="text" name="email"><br>
<input type="submit" value="register">
</form>
```

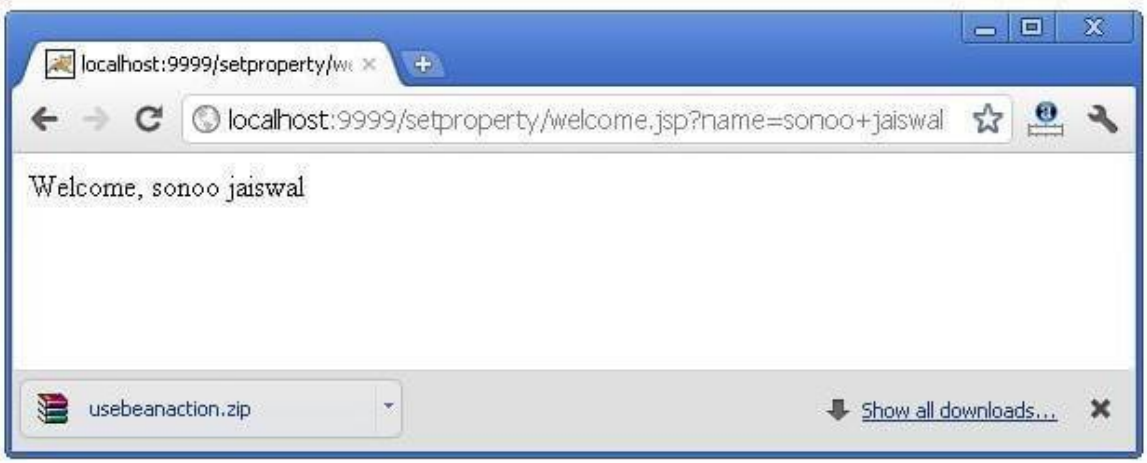
process.jsp

```
<jsp:useBean id="u" class="org.sssit.User"></jsp:useBean>
<jsp:setProperty property="*" name="u"/>

Record:<br>
<jsp:getProperty property="name" name="u"/><br>
<jsp:getProperty property="password" name="u"/><br>
<jsp:getProperty property="email" name="u" /><br>
```

User.java

```
package org.sssit;

public class User {
private String name,password,email;
//setters and getters
}
```





Reusing Bean in Multiple Jsp Pages

Let's see the simple example, that prints the data of bean object in two jsp pages.

index.jsp

Same as above.

User.java

Same as above.

---

process.jsp

```
1. <jsp:useBean id="u" class="org.sssit.User" scope="session"></jsp:useBean>
2. <jsp:setProperty property="*" name="u"/>
3. 
4. Record:<br>
5. <jsp:getProperty property="name" name="u"/><br>
6. <jsp:getProperty property="password" name="u"/><br>
7. <jsp:getProperty property="email" name="u" /><br>
8. 
9. <a href="second.jsp">Visit Page</a>
```

---

second.jsp

```
1. <jsp:useBean id="u" class="org.sssit.User" scope="session"></jsp:useBean>
2. Record:<br>
3. <jsp:getProperty property="name" name="u"/><br>
4. <jsp:getProperty property="password" name="u"/><br>
5. <jsp:getProperty property="email" name="u" /><br>
```

Using variable value in setProperty tag

In some case, you may get some value from the database, that is to be set in the bean object, in such case, you need to use expression tag. For example:

process.jsp

```
1. <jsp:useBean id="u" class="org.sssit.User"></jsp:useBean>
2. <%
3. String name="osu";
4. %>
5. <jsp:setProperty property="name" name="u" value="<%=name %>"/>
6. 
7. Record:<br>
8. <jsp:getProperty property="name" name="u"/><br>
```