Appendix A

Elements of Advanced Java Programming

Objectives

At the end of this appendix, you should be able to:

- Understand two-tier and three-tier architectures for distributed computing
- Understand the role of the Java programming language as a front-end for database applications
- Use the JDBC API
- Understand data interchange methodologies using object brokers
- Explain the JavaBeans Component Model
- Describe and use the `javadoc` and `jar` tools

Introduction to Two-Tier and Three-Tier Architectures

Client-server computing involves two or more computers sharing tasks related to a complete application. Ideally, each computer is performing logic appropriate to its design and stated function.

The most widely used form of client-server implementation is a two-tier client-server. This involves a front-end client application communicating with a back-end database engine running on a separate computer. Client programs send structured query language (SQL) statements to the database server. The server returns the appropriate results, and the client is responsible for handling the data.

The basic two-tier client-server model is used for applications that can run with many popular databases including ORACLE®, Sybase, and Informix.

A major performance penalty is paid in two-tier client-server. The client software ends up larger and more complex because most of the logic is handled there. The use of server-side logic is limited to database operations. The client here is referred to as a *thick client*.

Thick clients tend to produce frequent network traffic for remote database access. This works well for intranet and local area networks (LAN)-based network topologies but produces a large footprint on the desktop in terms of disk and memory requirements. Also, not all back-end database servers are the same in terms of server logic offered, and all of them have their own API sets that programmers must use to optimize and scale performance. Three-tier client-server, which is described next, takes care of scalability, performance, and logic partitioning in a more efficient manner.

Three-Tier Architecture

Three-tier is the most advanced type of client-server software architecture. A three-tier client-server demands a much steeper development curve initially, especially when you have to support a number of different platforms and network environments. The payback comes in the form of reduced network traffic, excellent Internet and intranet performance, and more control over system expansion and growth.

Three-Tier Client-Server Definition

Figure A-1 shows a diagram of a generic three-tier architecture.

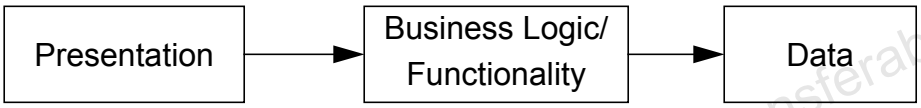

Figure A-1 Generic Three-Tier Architecture

The three components or tiers of a three-tier client-server environment are *presentation*, *business logic* or *functionality*, and *data*. They are separated such that the software for any one of the tiers can be replaced by a different implementation without affecting the other tiers. For example, if you wanted to replace a character-oriented screen (or screens) with a GUI (the presentation tier), you would write the GUI using an established API or interface to access the same functionality programs in the character-oriented screens. The business logic offers functionality in terms of defining all of the business rules through which the data can be manipulated. Changes to business policies can affect this layer without having any impact on the actual databases. The third tier or data tier, includes existing systems, applications, and data that has been encapsulated to take advantage of this architecture with minimal transitional programming effort.

A Database Front End

The Java programming language offers wide benefits to software engineers creating front-end applications for database-oriented systems. With its “Write Once, Run Anywhere™” language feature, the Java programming language offers immediate advantages in terms of its deployment on a wide range of hardware and operating systems. Programmers do not have to write platform-specific code for front-end applications, even in a multi-platform environment.

With Java technology’s rich set of supported front-end development classes, you can interact with databases through the JDBC API. The JDBC API provides a connectivity to back-end databases that can be queried with results being handled by the front end.

In a two-tier model, the database resides on a database server. The client executes a front-end application that opens a socket for communication over the network. The socket provides a communication path between the client application and the back-end server. In the following illustration, client programs send SQL database query requests to the database server. The server returns the results to the client, which formats the results for presentation. This architecture is shown in Figure A-2.

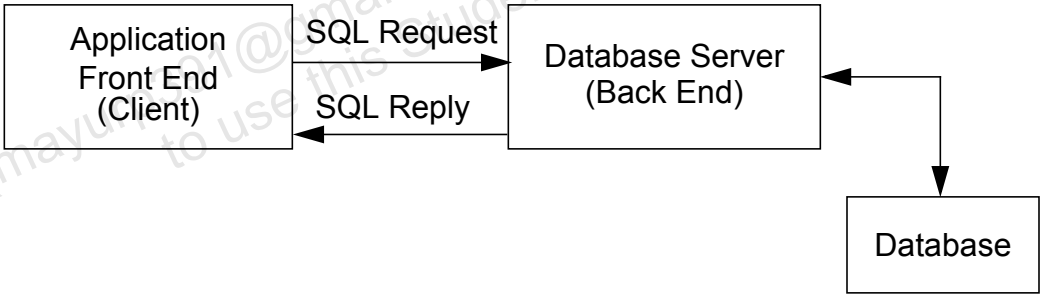

Figure A-2 Database-Centric Architecture

Frequently used mechanisms for data manipulations are often embedded as *stored procedures*. Triggers automatically execute stored procedures when certain conditions are activated during the course of manipulations on the database. The primary drawback of this model is that all business rules are implemented in the client application, creating large client-side runtimes and increased rewriting of the client’s code.

In a three-tier model, the presentation and control logic is embedded in the client (front-end) tier. It communicates with an intermediate server that provides a layer of abstraction from the back-end applications. This middle tier manages the business rules that manipulate the data per the governing conditions of the applications. It can also accept connections from several clients to one or more database servers on a variety of communications protocols. The middle tier provides a database-independent interface for applications and makes the front-end robust. Figure A-3 shows this architecture.

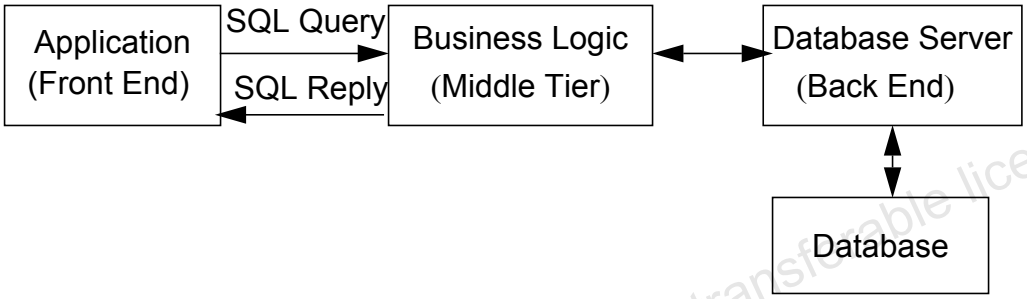

Figure A-3 Application-Centric Architecture

Introduction to the JDBC™ API

The ability to create robust, platform-independent applications and web-based applets prompted developers to create front-end connectivity solutions. JavaSoft™ technology worked with database and database-tool vendors to create a database management system-independent (DBMS-independent) mechanism that would enable developers to write client-side applications that worked with all databases. This effort resulted in the *Java Database Connectivity Application Programming Interface* (JDBC API).

JDBC, An Overview

The JDBC provides a standard interface for accessing a relational database. Modeled after the *open database connectivity* (ODBC) specification, the JDBC package contains a set of classes and methods for issuing SQL statements, table updates, and calls to stored procedures.

Figure A-4 shows a Java programming language front-end application uses the JDBC API to interact with the JDBC Driver Manager. The JDBC Driver Manager uses the JDBC Driver API to load the appropriate JDBC driver. JDBC drivers, which are available from different database vendors, communicate with the underlying DBMS.

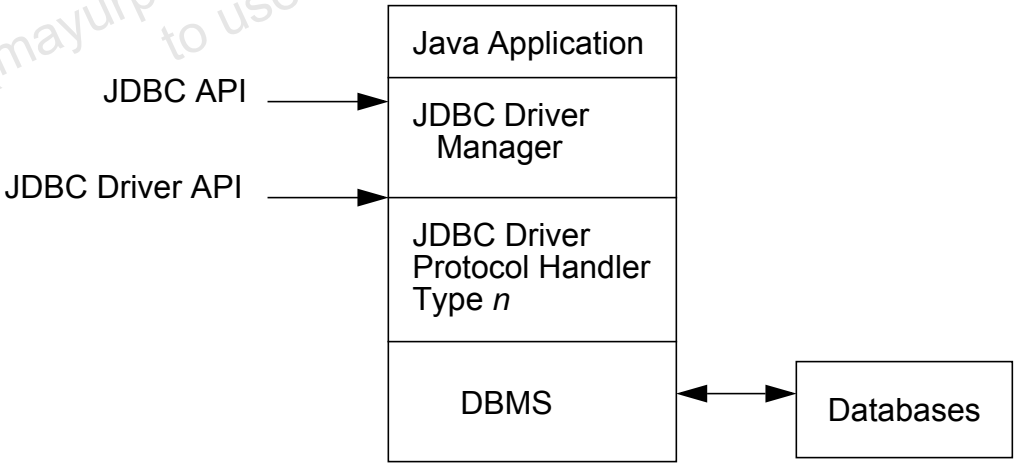

Figure A-4 Layers in a JDBC Application

JDBC Drivers

Java applications use the JDBC API to connect with a database through a database driver. Most database engines have different types of JDBC drivers associated with them. JavaSoft has defined four types of drivers. For more information, refer to `http://java.sun.com/products/jdbc/jdbc.drivers.html`.

The JDBC-ODBC Bridge

The JDBC-ODBC bridge is a JDBC driver that translates JDBC calls to ODBC operations. This bridge enables all DBMS that support ODBC to interact with Java applications. Figure A-5 shows the layers in a JDBC application that uses the ODBC bridge.

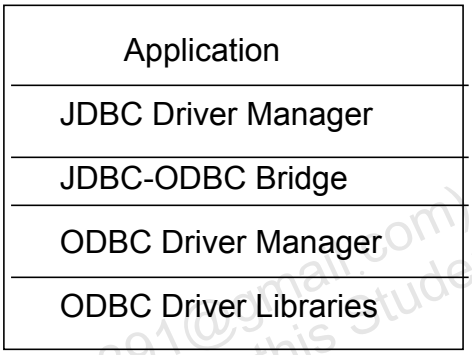

Figure A-5 A JDBC Application Using the ODBC Bridge

The JDBC-ODBC bridge interface is provided as a set of the C-shared dynamic libraries. ODBC provides a client side set of libraries and a driver specific to the client's operating system. These ODBC calls are made as C calls, and the client must have a local copy of the ODBC driver and associated client-side libraries. This places a restriction on its usage in web-based applications.

Distributed Computing

There are Java technologies available for creating distributed computing environments. Two popular technologies are the *remote method invocation* (RMI) and the *common object request broker architecture* (CORBA). RMI is analogous to the *remote procedure call* (RPC) and is preferred by programmers of the Java programming language. CORBA provides flexibility in heterogeneous development environments.

The RMI feature enables a program running on a client computer to make method calls on an object located on a remote server machine. It gives a programmer the ability to distribute computing across a networked environment. Object-oriented design requires that every task be executed by the object most appropriate to that task. RMI takes this concept one step further by allowing a task to be performed on the machine most appropriate to the task. RMI defines a set of remote interfaces that you can use to create remote objects. A client can invoke methods of a remote object with the same syntax that it uses to invoke methods on a local object. The RMI API provides classes that handle all of the underlying communication and parameter referencing requirements of accessing remote methods.

With all of the distributed computing architectures, an application process or *object server (daemon)* advertises itself to the world by registering with a naming service on the local machine (node). In the case of RMI, a naming service daemon called the RMI registry runs over an RMI port that by default listens over IP Port 1099 on that host. The RMI registry contains an internal table of remote object references. For each remote object, the table contains a registry name and a reference to that object. You can store multiple instances for the same object by instantiating and binding it multiple times to the registry, using different names.

RMI

When an RMI client binds a remote object through the registry, it receives a local reference to the remote instantiated object through its interface and communicates with the object through that reference. Local references to the same remote object can exist on multiple clients; any variables and methods contained within the remote object are shared.

The applet begins by importing the appropriate RMI packages and creating a reference to the remote object. After the applet establishes this link, it can call the remote object's methods as if they were locally available to the applet.

RMI Architecture

The RMI architecture provides three layers: Transport layers, Remote Reference layer, and Stubs / Skeleton layer. Figure A-6 shows these layers.

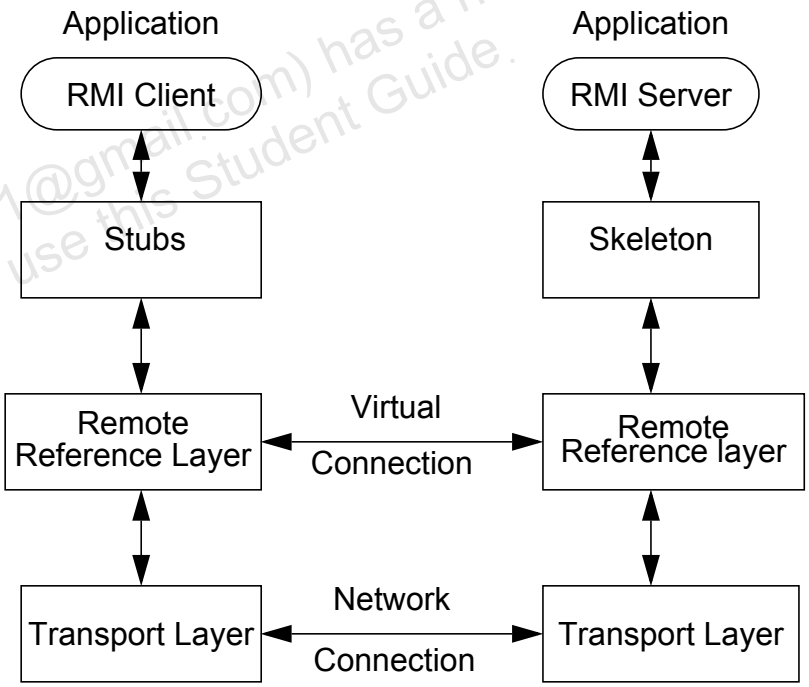

Figure A-6 Layers of an RMI Architecture

The Transport layer creates and maintains physical connections between the client and server. It handles the data stream passing through the Remote / Reference layers (RRLs) on the client and server side.

The Remote Reference layer provides an independent reference protocol for establishing a virtual network between the client and server. It establishes interfaces to the lower Transport layer and the upper Stub / Skeleton layer.

A Stub is a client-side proxy representing the remote object. The client interacts with the Stub through interfaces. The Stub appears as a local object to the client. The Skeleton on the server side acts as an interface between the RRL and the object implemented on the server side.

Creating an RMI Application

This section guides you through the steps for creating, compiling, and running an RMI application. The following steps illustrate the process:

1. Define interfaces for remote classes.
2. Create and compile implementation classes for the remote classes.
3. Create stub and skeleton classes using the `rmic` command.
4. Create and compile the server application.
5. Start the RMI Registry and the server application.
6. Create and compile a client program to access the remote objects.
7. Test the client.

CORBA

CORBA is a *specification* that defines how distributed objects can interoperate. The CORBA specification is controlled by the Object Management Group (OMG), an open consortium of more than 700 companies that work together to define open standards for distributed computing. For more information, refer to the following URL:

`http://www.omg.org`

You can write CORBA objects in almost any programming language, including C and C++. These objects can also exist on almost any platform, including the Solaris OS, Microsoft Windows, openVMS, Digital UNIX, HP-UX, and many others. This means a Java application running on a Microsoft Windows platform can interact over a network with C++ objects on a UNIX system.

Language independence is possible using the construction of interfaces to objects using the *Interface Definition Language* (IDL). IDL allows all CORBA objects to be described in the same manner; the only requirement is a *bridge* between the native language (C/C++, COBOL, Java) and IDL.

At the core of CORBA is the *object request broker* (ORB). The ORB is the principal component for the transmission of information between the client and the server of the CORBA application. The ORB manages marshalling requests, establishes a connection to the server, sends the data, and executes the requests on the server side. The same process occurs when the server wants to return the results of the operation. ORBs from different vendors communicate over a network (often, but not necessarily using TCP/IP) using the *Internet Inter ORB Protocol (IIOP)*, which is a part of the CORBA 2.0 standard.

The Java IDL

Java IDL adds CORBA capability to the Java programming language, providing standards-based interoperability and connectivity. Java IDL enables distributed Java applications to transparently invoke operations on remote network services, using the industry standard IDL and IIOP.

Java IDL is not an *implementation* of OMG's IDL. It is, in fact, a CORBA ORB that uses IDL to define interfaces. The `idltojava` compiler generates portable client stubs and server skeletons. The CORBA client interacts with another object running on a remote server by accessing a reference object through its *naming service*. Like the RMI Registry, the naming service is an application that runs as a background process on a remote server. It holds a table of named services and remote object references used to resolve client requests.

The steps involved in setting up a CORBA object can be summarized as follows:

1. Create the object's interface using the IDL.
2. Convert the interface into stub and skeleton objects using the `javatoidl` compiler.
3. Implement the skeleton object, creating the CORBA server object.
4. Compile and execute the server object, binding it to the naming service.
5. Create and compile a client object, which invokes the methods within the server object.
6. Execute the client object, accessing the server object through the CORBA naming service.

RMI Compared With CORBA

RMI's biggest advantage stems from the fact that it fully object oriented. By contrast, CORBA provides a largely-procedural mechanism for connecting distributed objects. Consider the command pattern. This pattern provides excellent flexibility and maintainability, but cannot be implemented properly between two CORBA systems because it requires that objects (both state and *behavior*) be moved from the client to the server. RMI can do this as a direct consequence of the platform independent bytecode.

One of the key benefits often cited for CORBA is its language independence. In fact, RMI can provide this too by using the Java Native Interface.

CORBA services are available for a significant variety of both vertical and horizontal problems, these services are often well-understood and mature. Examples of potentially important features of services are transactions and security.

CORBA's language independence adds significant complexity to the development cycle and precludes garbage collection features that RMI supports.

In many cases, the choice between RMI and CORBA is most likely to be made largely on political and environmental grounds, rather than on purely technical ones. A company that already has CORBA would need a compelling reason to introduce a new technology, especially if the change would render a substantial investment in services redundant. New systems, however, are likely to benefit from the use of RMI, provided that higher management do not perceive this as an immature technology.

The JavaBeans™ Component Model

The JavaBeans architecture is an integration technology, a component framework that allows reusable component objects (called *beans*) to communicate with one another and with the framework.

A Java bean is an independent and reusable software component that you can manipulate visually in a builder tool. Beans can be visible objects, such as AWT components, or invisible objects, such as queues and stacks. A builder or integration tool manipulates beans to create applets and applications. The component model specified by the JavaBeans 1.00-A specification defines five major services:

- Introspection – This process exposes the properties, methods, and events that a JavaBean component supports. It is used at runtime while the bean is being created with a visual development tool.
- Communication – This event-handling mechanism creates an event that serves as a message to other components.
- Persistence – Persistence is a means of storing the state of a component. The simplest way to support persistence is to take advantage of Java object serialization, but it is up to the individual browsers or the applications that use the bean to actually save the state of the bean.
- Properties – Properties are attributes of a bean that are referenced by name. These properties are usually read and written by calling methods on the bean created specifically for that purpose. Some property types affect neighboring beans as well as the one in which the property originates.
- Customization – One of the primary characteristics of a bean is its reusability. The beans framework provides several ways of customizing existing beans into new ones.

Bean Architecture

A bean is represented by an interface that is seen by the users. The environment must connect to this interface, if it wants to interact with this bean. Beans consist of three general-purpose interfaces: events, properties, and methods. Because beans rely on their state, they must be persistent over time.

Events

Bean events are the mechanism for sending asynchronous messages between beans, and between beans and containers. A bean uses an event to notify another bean to take an action or to inform the bean that a state change has occurred. An event allows your beans to communicate when something interesting happens; to do this, they make use of the event model introduced in JDK version 1.1. The event model used in Java 2 SDK is the same event model that was introduced in JDK version 1.1. There are three parts to this communication: event object, event listener, and event source.

The JavaBeans architecture communicates primarily using event listener interfaces that extend `EventListener`.

Bean developers can design their own event types and event listener interfaces and make their beans act as a source by implementing the `add`*`XXX`*`Listener(EventObject e)` and `remove`*`XXX`*`Listener(EventObject e)` methods, where *`XXX`* is the name of the event type. Then, the developers can make other beans act as event targets by implementing the *`XXX`*`Listener` interface. The `sourceBean` and the `targetBean` are brought together by calling `sourceBean.add`*`XXX`*`Listener(targetBean)`.

Properties

Properties define the characteristics of the bean. They can be changed at runtime through their `get` and `set` methods.

You can use properties to send two-way synchronous communications between beans. Beans also support asynchronous property changes between beans using special event communication.

Methods

Methods are operations through which you can interact with a bean. Beans receive notification of events by having the appropriate event source method call them. Some methods are special and deal with properties and events. These methods must follow special naming conventions outlined in the beans specification. Other methods might be unrelated to an event or property. All public methods of a bean are accessible to the beans framework and can be used to connect a bean to other beans.

Bean Introspection

The JavaBean introspection process exposes the properties, methods, and events of a bean. Bean classes are assumed to have properties if there are methods that either set or get a property type.

The `BeanInfo` interface, provided by the JavaBeans API, enables bean designers to expose properties, events, methods, and any global information about a bean. The `BeanInfo` interface provides a series of methods to access bean information, but a bean developer can also include private description files that the `BeanInfo` class uses to define bean information. By default, a `BeanInfo` object is created when introspection is run on the bean (Figure A-7).

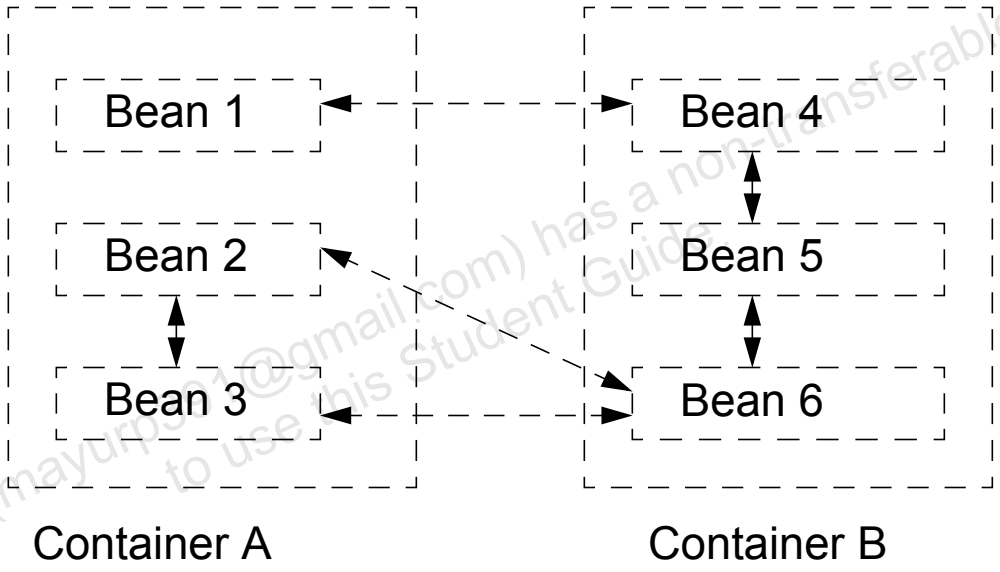

Figure A-7 A Sample Bean Interaction

A Sample Bean Interaction

In Figure A-7 on page A-16, Container A and Container B contain six beans. A bean can interact with other beans that are present in the same container and with beans that are in a separate container. In this example, Bean 1 interacts with Bean 4. It does not communicate with Bean 2 and Bean 3, which reside in the same container. This illustrates the point that a bean can communicate with any other bean and is not restricted to communicating with a bean in the same container. However, Bean 4 communicates with Bean 5, which resides in the same container. Source Bean 1 sends an event to the target Bean 4, which causes it to listen for messages on its event listener. All other inter-container and intra-container bean interactions can be explained in a similar manner.

The Beans Development Kit (BDK)

The BDK is a Java application developed by JavaSoft that enables Java technology developers to create reusable components that use the bean model. It is a complete system that contains source code for all examples and documentation. The BDK comes with a sample bean builder and customizer application called BeanBox. The BeanBox is a test container that you can use to do the following:

- Resize and move beans
- Alter beans with property sheets
- Customize beans with a customizer application
- Connect beans together
- Drop beans onto a composition window
- Save beans through serialization
- Restore beans

The BDK comes with a set of 12 sample beans, which cover all of the aspects of the JavaBeans API.

JAR Files

JAR (Java archive) is a platform-independent file format that aggregates many files into one. You can bundle multiple Java applets and their requisite components (`.class` files, images, and sounds) in a JAR file and subsequently download to a browser in a single Hypertext Transfer Protocol (HTTP) transaction, greatly improving the download speed. The JAR format also supports compression, which reduces the file size, further improving the download time. In addition, the applet author can digitally sign individual entries in a JAR file to authenticate their origin. It is fully backward-compatible with existing applet code and can be extended.

Changing the `applet` tag in your HTML page to accommodate a JAR file is easy. The JAR file on the server is identified by the `archive` parameter, which contains the directory location of the JAR file relative to the location of the HTML page; for example:

```
<applet code="Animator.class"
        archive="jars/animator.jar"
        width="460" height="160" >
  <param name="foo" value="bar">
</applet>
```

Using the javadoc Tool

Documentation of your code is important to the success of future maintenance efforts for standalone applications and is critical to the use of APIs. In this section, the javadoc tool, comment tags, and how to use the tool are briefly described.

Hopefully you already have some experience in using the Java 2 SDK documentation. Look at how you can generate documentation HTML pages for your projects.

```
> javadoc -private -d ../doc/api banking banking.domain banking.reports
```

Typically if you are generating documentation for an API, you would use the -public option (or leave it out because it is the default). However, it is common to use the -private option to generate documentation that is shared among an application project team.

Note – Read the online documentation for more details.

Documentation Tags

The javadoc tool parses the specified source files for comment lines that start with /** and end with */. The tool uses these to document the declaration that the comment immediately precedes.

The first sentence of the comment is called the *summary sentence*, and it should be a complete, concise description of the declaration. You can use the text following the summary sentence to give details about the declaration, including usage information. HTML tags can be included in any portion of the text, such as using the <P> tag to separate paragraphs, <UL> to generate lists, <B> (and so on) to format the text.

Also within the comment block, javadoc uses tags to identify special elements of the declaration, such as the return value of a method. The table in the preceding overhead shows a set of the most common javadoc tags, their meaning, and with which declarations they may be used.

Example

The following example is used in the tests of javadoc:

```
1   /*
2    * This is an example using javadoc tags.
3    */
4
5   package mypack;
6
7   import java.util.List;
8
9   /**
10   * This class contains a bunch of documentation tags.
11   * @author Bryan Basham
12   * @version 0.5(beta)
13   */
14  public class DocExample {
15
16    /** A simple attribute tag. */
17    private int x;
18
19    /**
20     * This variable a list of stuff.
21     * @see #getStuff()
22     */
23    private List stuff;
24
25    /**
26     * This constructor initializes the x attribute.
27     * @param x_value  the value of x
28     */
29    public DocExample(int x_value) {
30      this.x = x_value;
31    }
32
33    /**
34     * This method return some stuff.
35     * @throws IllegalStateException if no stuff is found
36     * @return List the list of stuff
37     */
38    public List getStuff()
39         throws IllegalStateException {
40      if ( stuff == null ) {
41        throw new java.lang.IllegalStateException("ugh, no stuff");
42      }
```

```
43        return stuff;
44      }
45    }
```

The following command results in the display shown in Figure A-8:

```
> javadoc -d doc/api/public DocExample.java
```

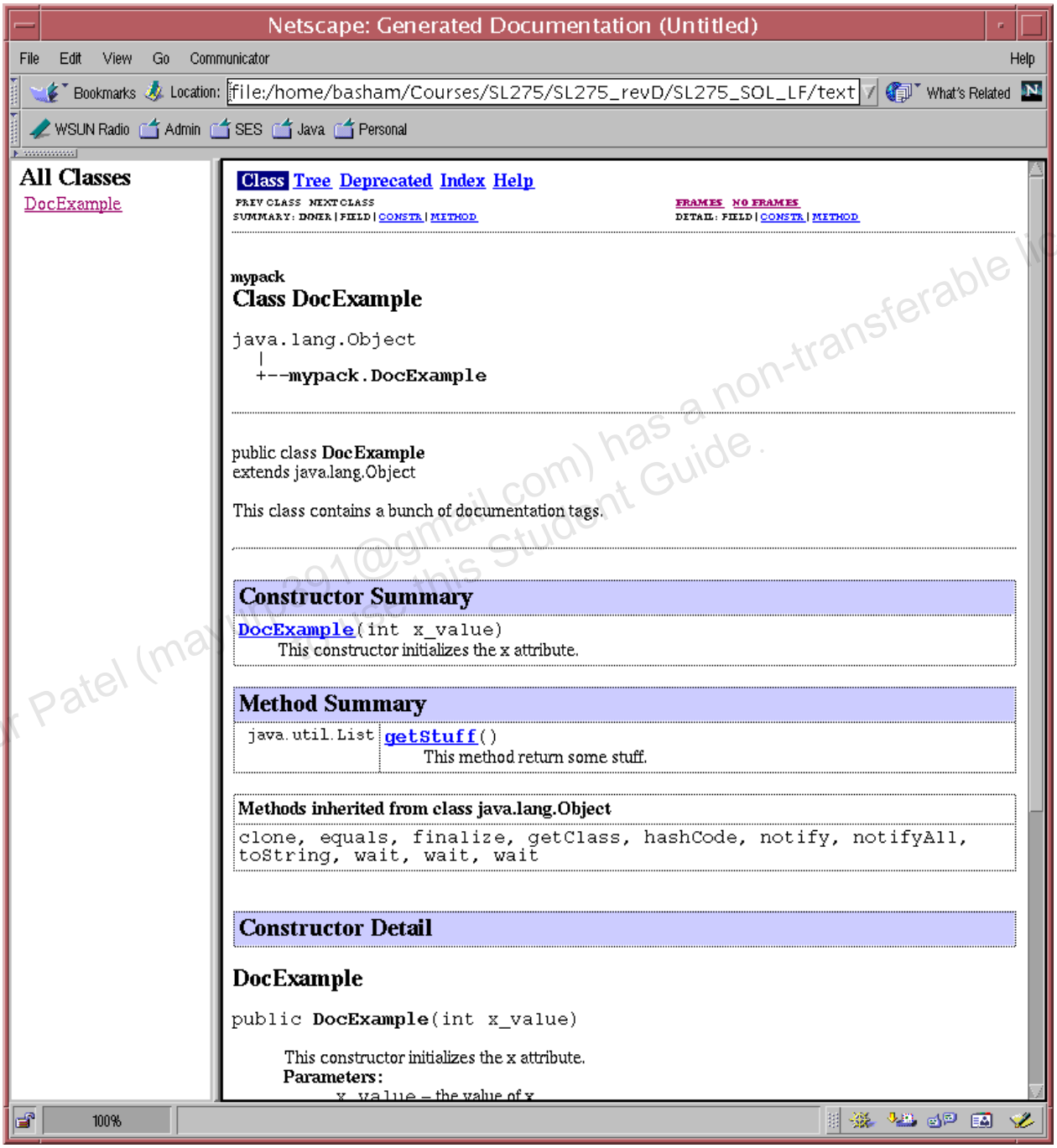

Figure A-8 Public Example

The following command results in the display shown in Figure A-9:

```
> javadoc -private -d doc/api/private DocExample.java
```

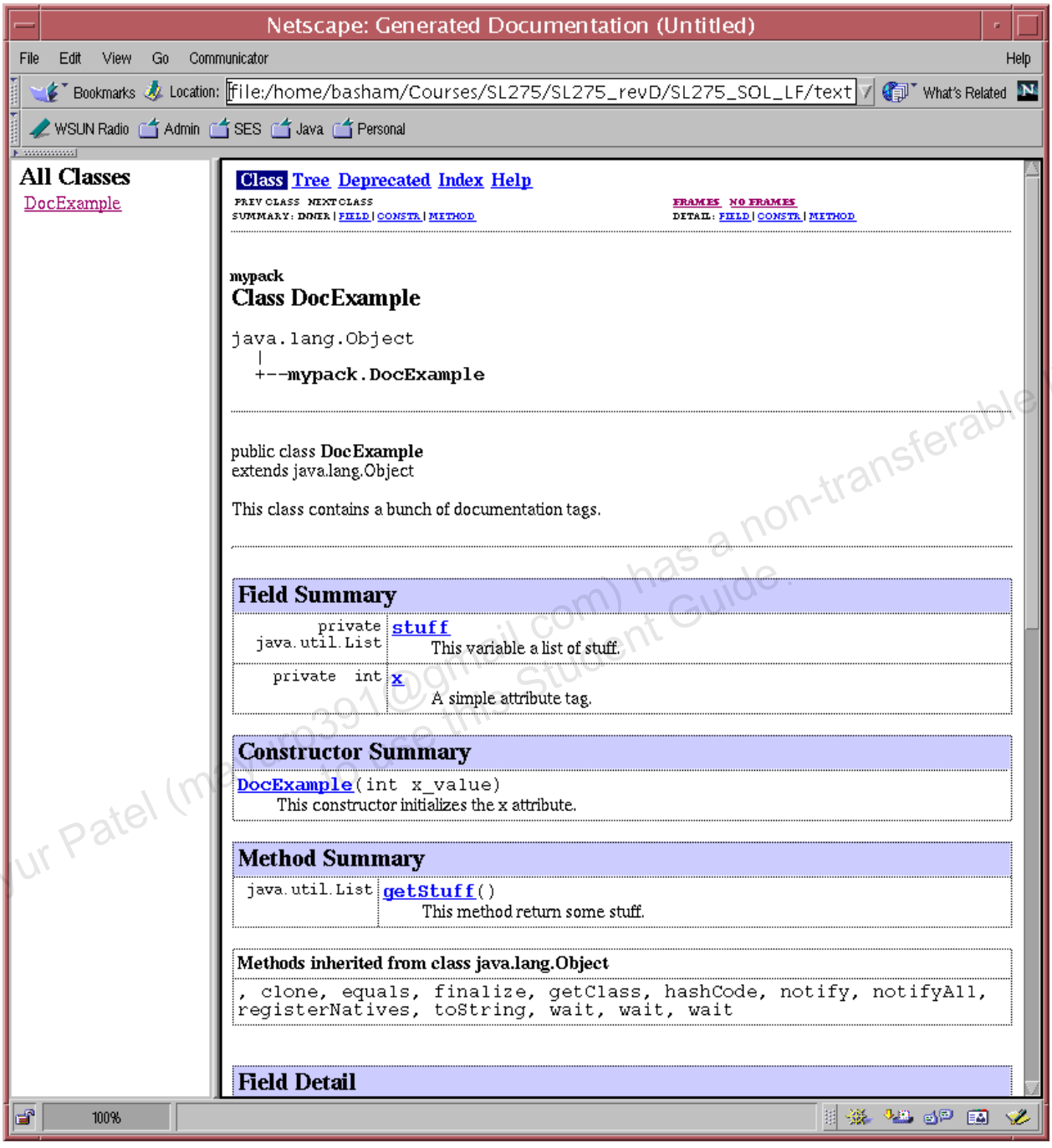

Figure A-9 Private Example