Java™ Tutorial For The Real World

Yakov Fain

Smart Data Processing, Inc.
Table Of Contents

Preface viii

Thanks ix

Lesson 1. You First Java Program 1

What’s So Good About Java? 1
Getting Started 2

Lesson 2. Object-Oriented Programming 7

Classes And Objects 7
Data Types 8
The Tax Calculation Program 10
Conditional Statement if 13
Switch Statement 14
Variable Scopes 14
Method Overriding 16

Lesson 3. Methods and Constructors 17

Method Arguments 17
Program Comments 17
Special Methods: Constructors 17
The Keyword this 18
Method Overloading 19
Arrays 20
Loops 21
Command-Line Arguments 22
The Keyword super 23
The Keyword final 24
Access Levels 24
Resources 25

Lesson 4. Graphic User Interface With AWT 26

Java Applets 26
The HTML Tag <Applet> 27
Java Archives – JARs 28
<table>
<thead>
<tr>
<th>Lesson 5. Event Handling. Interfaces. Inner Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces 42</td>
</tr>
<tr>
<td>AWT Event Listeners and Adapters 45</td>
</tr>
<tr>
<td>How to Use AWT Listeners 45</td>
</tr>
<tr>
<td>Independent Java GUI Applications 48</td>
</tr>
<tr>
<td>Inner Classes 49</td>
</tr>
<tr>
<td>How to Use AWT Adapters 50</td>
</tr>
<tr>
<td>Resources 51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lesson 6. Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading the Stack Trace 54</td>
</tr>
<tr>
<td>Exception Hierarchy 55</td>
</tr>
<tr>
<td>Try/Catch Block 55</td>
</tr>
<tr>
<td>Clause <em>throws</em> 57</td>
</tr>
<tr>
<td>Clause <em>finally</em> 57</td>
</tr>
<tr>
<td>Clause <em>throw</em> 59</td>
</tr>
<tr>
<td>User-Defined Exceptions 60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lesson 7. Java Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction 62</td>
</tr>
<tr>
<td>Byte Streams 63</td>
</tr>
<tr>
<td>Buffered Streams 64</td>
</tr>
<tr>
<td>Character Streams 65</td>
</tr>
<tr>
<td>Data Streams 68</td>
</tr>
<tr>
<td>Class <em>StreamTokenizer</em> 69</td>
</tr>
<tr>
<td>Class <em>StringTokenizer</em> 71</td>
</tr>
<tr>
<td>Class <em>File</em> 72</td>
</tr>
<tr>
<td>Java Serialization - Object Streams 72</td>
</tr>
<tr>
<td>Class <em>ObjectOutputStream</em> 73</td>
</tr>
</tbody>
</table>
Lesson 8. Java Network Programming

Introduction to Networking
Reading Data From the Internet
Connecting through HTTP Proxy Servers
How to Download Files From the Internet
The Stock Quote Program
Socket Programming
The Stock Quote Server with Sockets
How to Run the Stock Quote Server
Introduction to JavaMail API
How to Run the MailMan Program
Resources

Lesson 9. Data Structures And Collections

Arrays Revisited
Classes Vector and ArrayList
Classes Hashtable and HashMap
Interface Enumeration
Class Properties
Class BitSet
Class LinkedList
Resources

Lesson 10. Multithreading in Java

Introduction
Class Thread
Interface Runnable
Thread States
Sleeping Threads
Thread Priorities
Thread Synchronization. Race Condition.
Wait and Notify
Deprecated Methods
My Brokerage Firm With Threads
Resources

Lesson 11. Working With Databases Using JDBC
Introduction 117
JDBC Driver Types 117
Installing Oracle Database Server 118
Sample JDBC Program 119
Processing Result Sets 121
Class ResultSetMetaData 122
Scrollable Result Sets 124
JDBC-ODBC Bridge 125
Class PreparedStatement 126
Class CallableStatement 127
Batch Updates 127
My Brokerage Firm With the Database 128
Resources 129

Lesson 12. Introduction to Swing 130
Swing Components and Containers 130
Using Threads with Swing 132
Using JTable – the Big Picture 133
Using Table Models 135
How to Save JTable Data in the Database 138
Working with TableCellRenderer Interface 139
Resources 140

Lesson 13. Miscellaneous Topics 141
Casting 141
Abstract Classes 142
Polymorphism 144
Reflection 146
Cloning 148
Resources 150

Lesson 14. Java Servlets 151
Java 2 Enterprise Edition - J2EE 151
Introduction to Java Servlets 153
Thin Client – the Front Tier 154
Servlet – the Middle Tier 154
Browser-Servlet Data Flow 155
Your First Servlet 156
HTTP Get and Post Requests 158
Session Tracking With Servlets 158
Cookies 159
URL Rewriting 160
Hidden Fields 161
Lesson 15. Java Server Pages

What’s Wrong With Servlets
Embedding Java Code Into HTML
Major JSP Tags
Implicit JSP Objects
Error Pages
Java Beans
Using Java Beans in JSP
How Long Does a Bean Live?
Loading JSP From Servlets
Testing JSP in VisualAge/WebSphere
Stock Portfolio Project With JSP
 Deploying JSP in WebLogic
Tag Libraries
Resources

Lesson 16. Two Web Applications

Applet-Servlet Communications
The Online Store

Lesson 17. RMI And JNDI

Remote Method Invocation
Developing Applications With RMI
Defining Remote Interfaces
Implementing Remote Interfaces
Registering Remote Objects
Writing RMI Clients
Finding Remote Objects
Stubs And Skeletons
Starting the Server and the Client
Setting Up The Stock Server Application
Java Naming and Directory Interface
Naming And Directory Services
Setting Up JNDI Example
<table>
<thead>
<tr>
<th>Lightweight Directory Access Protocol</th>
<th>207</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td>208</td>
</tr>
</tbody>
</table>

**Lesson 18. Enterprise Java Beans**

- Application Server And EJB Container | 210  |
- Bean Types                            | 210  |
- Stateless Session Beans               | 211  |
- Home Interface                        | 212  |
- Remote Interface                      | 213  |
- Please Welcome – Mr. Bean             | 214  |
- Writing EJB Clients                   | 215  |
- EJB Deployment Descriptors            | 217  |
- Testing TradeOrder Session Bean in WebLogic | 218  |
- Stock Trading System Specification    | 219  |
- Stock Trading System Database         | 220  |
- Resources                             | 222  |

**Lesson 19. Session Beans (Cont.)**

- Stateful Session Beans                | 223  |
- Lifecycle of a Stateful Session Bean  | 223  |
- Creating a Stateful Bean              | 224  |
- Bean Activation, Passivation and Removal | 225  |
- Bean’s Session Context                | 226  |
- Transaction Attributes                | 227  |
- Bean and Container Managed Transactions | 229  |
- More on EJB Deployment Descriptors    | 229  |
- Deploying EJB in WebLogic Server      | 232  |
- Database Connection Pools             | 233  |
- JDBC Data Sources                     | 234  |
- Implementing STS (Session Beans Version) | 235  |
- How to Run the Stock Trading System   | 241  |

**Lesson 20. Entity Beans**

- Introduction                          | 243  |
- Types of Persistence                   | 244  |
- Primary Key of an Entity Bean          | 245  |
- Finding Entity Beans                   | 247  |
- Life Cycle of an Entity Bean           | 248  |
- Entity Beans With BMP                  | 249  |
- Entity Beans With CMP                  | 250  |
- Entity Beans Relations                 | 253  |
- Query Language – EJB-QL                | 254  |
Lesson 21. Java Messaging Service

Introduction
Two Modes of Message Delivery
JMS Classes and Terms
Types of Messages
How to Send a Message
How to Receive a Message
How to Publish a Message
How to Subscribe for a Topic
Message Selectors
Message-Driven Beans
Configuring WebLogic JMS Objects
How to Run JMS Samples
Resources

Appendix A. Java Technical Interviews

Rules of the Game
To Get or Not to Get Certified?
Technical Questions and Answers
Resources

Appendix B. Installing WebLogic Application Server

Index
Preface

The online bookstore Amazon.com has more than 1500 Java titles. There is also an overwhelming amount of Java documentation and articles on the Internet. But still, most programmers prefer training manuals because they are short and have lab instructions for building applications that actually work. While teaching Java-related courses, I was giving away lecture handouts and my students encouraged me to write my own Java tutorial. My main goal was not to write yet another fat book that covers everything, leaving you lost in details, but rather a compressed practical manual with clear instructions for understanding and trying out attached code samples.

This book should get you up to speed with Java as soon as possible, focusing on the features of the language that any decent Java developer should know. Each lesson ends with references to Internet resources that provide additional coverage of the related material.

The main features of the Java language are covered in the first half of the book and such advanced topics as working with databases, Java Servlets, JSP, EJB, and JMS are explained in the second half. Most of the lessons from this book come with working applications and setup instructions, if needed. The first lessons come with independent applications and the second half of the book leads you through development of a Stock Trading System, the final version of which is designed using Java servlets, JSP, EJB, and JMS. While Java is probably unique in the number of freely available compilers and application servers, we’ll be using commercial products in this book – the software that people use at work. That’s why you’ll have to download and install free evaluation copies of commercial products such as WebLogic and Oracle.

The source code of all book samples is available at www.smartdataprocessing.com.

While working as an architect-designer-consultant for large corporations I often have to conduct technical Java interviews. The technical questions and suggested answers are included at the end of the book and could be useful for both parties – interviewers and job applicants.

I would really appreciate any reader’s comments and suggestions regarding this book. Please e-mail me your comments or questions at the following address: yakov@smartdataprocessing.com.
Thanks

First of all, a big "Thank You" to my students for their support during all these years.

Also, a big thanks to my family:

my wife Natalia, a senior PowerBuilder programmer, for her support and valuable feedback;

my son Yuri, a student of New York School of Visual Arts, who created the cover of this book;

my son David, for his positive attitude towards life.

I am also grateful to excellent programmers, Yuri Goncharov from Toronto, and Andrey Postoyanets from New York, who read through the drafts of the books, made valuable comments, and have helped with some of the book samples.

Special thanks to my colleagues - members of the Zeus team working on a complex and interesting distributed real-time trading system – I'm learning from you guys every day.
Lesson 1

Your First Java Program

What’s So Good About Java?

What makes this language different from many others?

1. Portability

This term means that you can write a program for one type of computer, but run it on many different platforms. Most likely you’ve had a chance to see it cause some Internet pages besides HTML contain so called applets, which are Java programs that work under control of your browser. Say you went to an auto-dealer's web page that has a car loan calculator written in Java. People use different Operational Systems: Windows, Unix, etc., but all of them have downloaded the same Java program. Java programs run in a shell called Java Virtual Machine (JVM) that shields them from the Operational System and its hardware. Java programs do not even know in which environment they run!

2. Built-in Security

What are the chances that some bad guy will decide to write an applet in Java that will delete some files from your hard disk? No way, because Java applets can not access the hard disks of user computers unless special files (digital signatures) were created.

3. Error Processing.

It is mandatory in Java to handle possible run-time errors. Java has a mechanism called “Exceptions” and if you are calling a method (function or subroutine) that may give a run-time error, you will not be able to compile this program until you have taken care of these possible errors. This feature lowers the number of potential bugs in Java programs, while in other languages, programmers have to decide what kind of errors should be processed.

4. Internationalization

Internationalization is the ability to translate you programs to other languages. Amazon.com sells books in Germany (www.amazon.de) and Sun
Microsystems sells their products in Russia ([www.sun.ru](http://www.sun.ru)). It’s not difficult to translate an English site to a Russian one because these languages have alphabets consisting of a limited number of letters. Each letter takes one byte of memory space in most programming languages. One letter in the English language takes up the same exact amount of memory as one letter in Russian, but this is not the case in the Chinese language. One byte can not store thousands of Chinese symbols. Java reserves two bytes of memory for each character, which allows storing of over 65,000 different characters as opposed to only 255 in case of one byte characters.

### 5. Multithreading

Multithreading allows one running program to start multiple executing lightweight processes, which run in parallel. This can speed up the performance of your programs tremendously, especially if the computer is equipped with multiple processors. It also saves a lot of memory space when multiple users are working with the same computer at the same time. Instead of starting a new copy of a program for each user, the same program just starts a new thread, which needs a lot less resources. Other languages may also have a multithreading feature, but in Java, threads are a lot easier to program. A mid-level Java programmer could write programs with multithreading, while in C++ it has to be done by a guru.

### 6. Java is easier to learn than C++

While Java is a very rich and powerful object-oriented programming language based on C++, some of the most complex elements of the C++ language have been removed from Java.

### Getting Started

The Java Development Kit (JDK) could be downloaded from the Sun Microsystems’ Internet site at [http://java.sun.com/j2se/1.4/](http://java.sun.com/j2se/1.4/).

The installation process is pretty simple – just run the downloaded executable file and it’ll install it on your disk (the default directory for Java under Microsoft Windows is `c:\jdk1.4`).

To start writing a Java program you could use any plain text editor. In Windows, it could be an editor called Notepad. In UNIX, it could be the `vi` editor. The files with Java programs must be saved in a plain text format and must have names ending in `.java`. For example, if you want to write a program called `HelloWorld`, enter its code in Notepad and save it in a class named `HelloWorld.java`. 
Keep in mind that Java is a case sensitive language, which means that if you named the program HelloWorld with a capital H and a capital W, do not try to start the program helloworld. Your first dozen of syntax errors will be caused by improper capitalization.

Here is the infamous program that prints the words **Hello World** on the screen:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Now you need to compile this program. We’ll be using the *javac* compiler, which is a part of JDK.

Let’s say you’ve saved your program in the directory called `c:\practice`. Open a command window on your PC, change the current directory to `c:\practice` and compile the program:

```
c:\>cd \practice

c:\practice>javac HelloWorld.java
```

If your environment is set properly and your program does not have syntax errors, it will create a new file called `HelloWorld.class` in the same directory.

If an error message is displayed saying something like “javac is not found”, or “bad command/file name” make sure that the directory `c:\jdk1.4\bin` is included to the search path of your environment.

- If you are using Windows 98, open the file `c:\autoexec.bat` and add the directory where your JDK is installed to the environment variable *PATH*, for example

  ```
c:\jdk1.4\bin;
  ```

- In Windows 2000 or XP set the *PATH* using the menu *Settings | Control Panel | System | Environment Variables*.

- In Unix – add it to the shell’s *PATH* environment variable.
You won’t see any confirmation of a successful compilation, just type `dir` in Windows or `ls` in Unix, and a new file named `HelloWorld.class` has to be there. This proves that your program has been successfully compiled.

If the program has some syntax errors, the compiler will print error messages. In this case you’d need to fix the errors, and recompile the program again. You may need to do it more than once until the file `HelloWorld.class` is created.

Now let’s run the program - enter the following:

```
c:\practice> java HelloWorld
```

Please note that we do not start `javac`, but `java`, which is called the Java run-time environment or the Java Virtual Machine (JVM).

This time the error message may say that the `HelloWorld.class` is not found. Even though you have the `.class` file in the same directory as your `.java` file, JVM is not going to look for it in the current directory unless the current directory is listed in the so-called `CLASSPATH` variable. Don’t confuse this with the variable `PATH`, that’s been discussed earlier.

The variable `CLASSPATH` variable is used by the JVM to find compiled classes. Let’s do a procedure similar to what you’ve done with the `PATH`.

For example, in Windows 98, open the file `autoexec.bat` and add the following line to it:

```
set CLASSPATH=.;
```

The dot above represents the current directory. If you already had the `CLASSPATH` variables set in your machine, just add the dot and semicolon to the end of its value.
Give your Java class and its file the same name. There could be exceptions to this rule, but not in this simple program.

While writing Java programs, you create classes which represent objects from real life. You'll learn more about classes in the lesson called “Introduction to Object-Oriented Programming in Java”.

Our HelloWorld program is also a class and it contains a method main(). Methods in Java classes represent actions that the class could perform. The method main() calls the method println() to display the text “Hello World” on the screen.

Here is the method signature of the method main():

```java
public static void main(String[] args)
```

The method signature includes the access level - public, instructions on usage - static, return value type - void, name of the method - main, and the argument list - String[] args.

The keyword public means that the method main() could be accessed by any other Java class.

The keyword static means that you don't have to create an instance of this class to use this method.

The keyword void says that the method main() doesn’t return any value to the calling program.

The keyword String[] args tells us that this method will receive an array of Strings as the argument (some values could be passed to this method from a command line).

The main() method is the starting point of your program. You can have a program that consists of more than one class, but at least one of them usually has the method main(), otherwise the program will not start. A Java class can have more than one method. For example, a class Employee can have the methods updateAddress(), raiseSalary(), changeName(), etc.

The body of the method main() contains the following line:

```java
System.out.println("Hello World");
```

The println() is a method that is used to print data on the system console (command window). Java’s method names are always followed by parentheses.
System and out are not methods, but names that represent other Java classes.

System.out means that the variable out is defined inside the class System.

The out.println() tells us that there is an object represented by a variable called out and it has a method called println().

We will be using this so-called dot notation to access class methods or variables. Say you have a class Employee that has a method changeAddress(). Here is an example:

Employee.changeAddress(“25 Broadway”)

Let’s review the steps you would perform to create and run the HelloWorld program:

Step 1. Set the values for the PATH and CLASSPATH system variables.

Step 2. Create a new directory called practice.

Step 3. Using a text editor, enter the code of the class HelloWorld and save it in the file c:\practice\HelloWorld.java.

Step 4. Compile and run the program:

   c:\practice> javac HelloWorld.java
   c:\practice> java HelloWorld

Assignment. Write a program to print your address using more than one statement println().

After trying your first couple of Java programs you may want to move from a simple text editor to an Integrated Development Environment (IDE) with Java-oriented editor, debugger, help and other conveniences. I like the following two products: IDEA from JetBrains (www.intellij.com) and JBuilder from Borland (www.borland.com). Some super-duper Java programmers prefer working with plain text editors and compilers, and you also will make your choice after mastering this tutorial. If you prefer free software, get the NetBeans at www.netbeans.org.
Lesson 2

Object-Oriented Programming

Classes And Objects

Java is an object-oriented language. As I've mentioned earlier, Java programs consist of classes that represent objects in the real world.

Classes in Java may have methods and attributes.

Let's create and discuss a class named Car. This class may have one or more methods, which can tell what the objects of this class can do: start the car, stop it, accelerate, lock the doors, and so on.

This class also may have some attributes or properties: color of the car, number of doors, size of engine, and so on.

Our class Car may represent some common features for many different cars: all cars have such properties as color and the number of doors, and all of them perform similar actions. We can be more specific and create another Java class called ToyotaCorolla. It's still a car, but with some properties specific to the model Toyota Corolla.

We will be using such terms as an object, which is an instance of a class. The phrase “to create an instance of a class” means to create a copy of the object in the computer’s memory, based on the class definition.

Factory specifications of a Toyota Corolla plays a similar role as the Java classes. The process of building real cars based on these specs is equivalent to creating instances of this class in Java.

In many cases, a program can’t use the Java class until its instance has been created. Obviously, you can create thousands of cars based on the same Toyota Corolla specifications. Even though they all represent the same class, they may have different values in their properties - some of them are red, some of them have two doors, while others have four, etc. In other words, we may create multiple instances of the class Toyota Corolla:
class ToyotaCorolla{
    String color;
    int numberOfDoors;

    void startEngine {
        ...
    }
    void stopEngine {
        ...
    }
}

Data Types

Java variables have to be declared before usage – they must have an assigned data type.

There are 8 primitive data types in Java: 4 data types are for integer values, 2 are for values with a decimal point, 1 char, and 1 boolean (allows only the values true or false).

All of these primitives have corresponding wrapper classes that contain useful methods dealing with respective data types.

The data type char allows you to store only one character, while classes String or StringBuffer are used for holding a longer text, i.e.

String lastName="Smith";
char grade = 'A';

Please note that the char data type uses 2 bytes of memory to store the data.

Below are some examples of variable declarations and initializations.

int chairs = 12;
boolean cancelJob = false;
double nationalIncome = 23863494965745.78;
float hourlyRate = 12.50f;   // add an f at the end of float literals
long totalCars  = 4637283648392l;  // add an l at the end of long literals
The table below contains some characteristics of the Java data types.

<table>
<thead>
<tr>
<th>Prim. Type</th>
<th>Size</th>
<th>Min Value</th>
<th>Max Value</th>
<th>Wrapper class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte</strong></td>
<td>8 bits</td>
<td>-128</td>
<td>127</td>
<td>Byte</td>
</tr>
<tr>
<td><strong>short</strong></td>
<td>16 bits</td>
<td>-32,768</td>
<td>32,767</td>
<td>Short</td>
</tr>
<tr>
<td><strong>int</strong></td>
<td>32 bits</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
<td>Integer</td>
</tr>
<tr>
<td><strong>float</strong></td>
<td>32 bits</td>
<td>Single precision floating point (IEEE754 document)</td>
<td>Single precision floating point (IEEE754 document)</td>
<td>Float</td>
</tr>
<tr>
<td><strong>double</strong></td>
<td>64 bits</td>
<td>Double precision floating point (IEEE754 document)</td>
<td>Double precision floating point (IEEE754 document)</td>
<td>Double</td>
</tr>
<tr>
<td><strong>char</strong></td>
<td>16 bits</td>
<td>Unicode 0</td>
<td>Unicode 2 in a power of 16 value</td>
<td>Character</td>
</tr>
<tr>
<td><strong>boolean</strong></td>
<td>-</td>
<td>false (not a min.)</td>
<td>true (not a max.)</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

The Tax Calculation Program

Let’s design and write a program that will calculate the state tax.

First, we need to decide what Java class(es) we need to write. Second, we’ll think about properties and methods that our class(es) should have.

Start your text editor and enter the following:

```java
class Tax{
}
```

The open and close curly brackets are used in various contexts to enclose the body of a class, a method, and some other Java elements. Every open curly brace must have the closing one.
Save your class in the file called Tax.java. You can't go wrong with that. Let’s think about the data that this class would need to perform the state tax calculation.

Obviously, you will need to know the gross income of a person for the last year. This is a good candidate for a property of this class. Properties in Java are represented by variables. Before declaring a variable for the gross income, we need to decide what types of data we'll be storing in it. These are some of the Java data types:

- int, double, float, char, String...

Let’s add the variable grossIncome of the data type double to our class, because this type allows numbers with a decimal point:

```java
class Tax{
    double grossIncome;
}
```

We also need to know what state the person lives in - different states may have different taxation rules. Possible values of this variable are: “NY”, “NJ”, “CT”, etc. The String data type represents text data:

```java
class Tax{
    double grossIncome;
    String state;
}
```

Let’s add one more property for dependents. This will be an integer variable, since a person can not have two and a half dependents.

```java
class Tax{
    double grossIncome;
    String state;
    int    dependents;
}
```

It’s time to add some methods to our class. We definitely need to be able to calculate the state tax based on the values of gross income, number of dependents, and state. Let’s create a method called calcTax():

```java
class Tax{
    double grossIncome;
    String state;
    int    dependents;
    // Method to calculate the state tax
    public double calcTax() {
        // Implementation
    }
}
```
public double calcTax() {
    ...  
    return 234.55;  // returning a hard-coded value
}

This method's signature tells us the following:

- Any external class could access this method (public).
- This method will return a value of type double.
- The name of the method is calcTax.
- This method does not need any values from outside – empty parentheses mean that the method does not have any arguments.

How do we decide if a method should return a value? If your method performs some calculations and has to give this value back to a calling program, it has to return a value. If a method just prints some information, it may not need to return any value, but you still need to mention this in a method signature by using a special keyword void:

public void printCustomers() {...}

Quiz. Select the proper answer for the following question:
A method increaseSalary (int newSalary) should return:

- integer
- String
- void
- boolean
- none of the above

I'd answer boolean, because after some business processing, we need to notify the calling program if we succeeded (by returning true), or failed (by returning false). Strictly speaking, it's up to the programmer to decide what to return from a method, but boolean sounds right to me in this case.

Java has a return statement and this is how a method returns data contained in a variable myResult to a calling program:

return myResult;
Our class `Tax` has to be **instantiated** before we start using it. Let’s create one more class called `TestTax`. This class will just instantiate and use our class `Tax`. Here’s what the class `TestTax` should do:

1. Create an instance of the class `Tax`.
2. Assign some values (gross income, state...) to the variables of the class `Tax`.
3. Call the method `calcTax()`.
4. Print the result on the screen.

The class `TestTax` will be stored in a separate file named `TestTax.java`.

```java
class TestTax{
    public static void main(String[] args){
        Tax t = new Tax(); // creating an instance
        t.grossIncome = 50000;  // assigning the values
        t.dependents = 2;
        t.state = "NJ";

        double yourTax = t.calcTax(); //calculating tax

        // Printing the result
        System.out.println("Your tax is " + yourTax);
    }
}
```

In the code above, we’ve declared a variable `t` of type `Tax`.

The method `main()` is an entry point to our program. This method creates an instance of the class `Tax` using the Java operator `new`. The variable `t` points to a place in the memory where the `Tax` object was created. From now on, if we want to work with the class, we will be using the variable `t`.

The following three lines assign values to the properties of the object `Tax`. 
After that, we'll calculate tax by calling the method `calcTax()` and the result will be assigned to our variable `yourTax`

The last line of method `main()` just displays the result on the system console.

At this point we already have two classes communicating with each other (the `TestTax` and the `Tax`).

**Quiz:** We've declared the variable `yourTax` as `double` because:

a) Type `double` can store numbers with decimal points.

b) Calculated tax could be a large number, and the `float` type is not big enough.

c) The method `calcTax` returns a double value.

The correct answer is c because the data type of the “receiving” variable has to match the return type of a method.

### Conditional Statement `if`

We always make decisions in our life: “If she'll say this – I'll answer with that, otherwise I'll do something else”. Java has an `if` statement that checks if a particular condition is true or false. This allows you to program different reactions based on the result of the check.

If the condition expression returns `true`, the code between the first curly braces will be executed, otherwise the code after the `else` statement will take place. For example,

```java
if (totalOrderPrice > 100)
   System.out.println("You’ll get a 20% discount");
else
   System.out.println("Order books for more than a" +
                     " $100 to get a 20% discount");
```

To make our example more realistic, let’s add some code to perform tax calculations in the method `calcTax()`.

Let’s say if the gross income was less that $30,000, we will take 5% for state tax. If it’s greater than $30,000, we will take 6%.

```java
public double calcTax() {
    double stateTax=0;
    if (grossIncome < 30000) {
        stateTax=grossIncome*0.05;
    }
    // Add state tax calculation for grossIncome >= 30000
}
```
14 The Java Tutorial For The Real World

```java
else{
    stateTax= grossIncome*0.06;
}
return stateTax;
```

### Switch Statement

The `switch` statement is an alternative to `if`. The variable in the `switch` condition is evaluated and the program goes to one of the `case` clauses:

```java
int taxCode=someObject.getTaxCode(grossIncome);
switch (taxCode){
    case 0:
        System.out.println("Tax Exempt");
        break;
    case 1:
        System.out.println("Low Tax Bracket");
        break;
    case 2:
        System.out.println("High Tax Bracket");
        break;
}
...
```

Do not forget to put the `break` at the end of each `case` to jump out of the `switch` statement, otherwise the code above may print more than one line even though a `taxCode` could have only one value.

### Variable Scopes

If you declare a variable inside any method, the variable has a local scope (the variable `stateTax` is local). This means that it’s visible only for the code within this method (calcTax). When the method is finished, the variable automatically gets destroyed.

If a variable has to be visible through more than one method in a class, you should declare it on a class level. In the class `Tax`, `grossIncome`, `dependents`, and `state` are **class variables**. These variables are “alive” when the class exists in memory. They could be shared and reused by all methods within the class and they can even be visible from external classes, for example `TestTax` class is accessing them.
In the previous examples we were using **hard-coded** values for calculations (values that didn’t change). Later on, we’ll make sure that our program can perform tax calculations for any income, state, and number of dependents.

Let’s introduce some new object-oriented terms. One of them is called **inheritance**, which is an ability to create a new object, based on an existing object.

We were planning to use the class `Tax` for all states, but what if the state of New Jersey introduces educational tax deductions? If you have a kid in college, this makes you eligible for an additional $500 deduction from your taxes. In this case, we have to either change the method `calcTax()`, or create another class that will be based on our class `Tax` plus have this new functionality to calculate educational deductions.

In real life, every person inherits some features from his or her parents. This similar process exists in Java. The special keyword `extends` is used to indicate that one class has been inherited from another:

```java
class NJTax extends Tax{
    ...
}
```

The class `NJTax` will have all features the class `Tax` has, plus you can add some new properties and methods to it. The class `Tax` is called a **superclass**, and the `NJTax` is called a **subclass**. You could also use such terms as **ancestor** and **descendent** respectively. This new class will have access to variables and methods of its superclass (unless they have a `private` access level, but let’s worry about that later).

It’s time to create a method called `adjustForStudents()` in the class `NJTax`.

```java
class NJTax extends Tax{
    double adjustForStudents (double stateTax){
        double adjustedTax = stateTax - 500;
        return adjustedTax;
    }
}
```

How will the class `TestTax` create an instance of the class `NJTax`? Here you go:

```java
NJTax  t= new NJTax();
```

Now you can call methods defined in the class `Tax` as well as in the `NJTax` using the variable `t`, for example:
NJTax t = new NJTax();
double yourTax = t.calcTax();

double totalTax = t.adjustForStudents(yourTax);

Please note that we’ve added a new functionality to the tax calculation program without changing the code of the class Tax. Another important note is that even though the variable t refers to an instance of the class NJTax, we are calling the method defined in its superclass:

t.calcTax();

The above code fragment also shows how you can pass a value calculated from one method to another one. We are passing the value of the variable yourTax to the method adjustForStudents() as an argument.

Method Overriding

The next important term of object-oriented programming is method overriding. Imagine that a superclass has 20 methods. Most of them are generic for all states, but there is one method that is not valid for the state of New Jersey. Instead of modifying this method in the superclass, we could create another method in the subclass with the same name and argument list.

It’s time to modify our previous example. The goal is to change the calcTax() method without modifying the superclass. We’ll declare the method calcTax() in the subclass – NJTax. By doing that, we’re suppressing the method of the superclass with the same name and argument list.

Let’s try a simple test - create a method calcTax() in NJTax and only write one line:

```java
public double calcTax() {
    return 1000;
}
```

Compile the code and run the TestTax program – it will prints the number 1000 as a result, which proves that only the calcTax() from the subclass has been called.

The advantages of using method overriding as opposed to direct code modification are:

1. The source code of the superclass may not be available, but you still need to change its functionality.
2. Somebody else may still need the original version, that’s why you can’t touch it.
Lesson 3

Methods and Constructors

Method Arguments

JDK comes with a variety of classes and each of them may contain methods. Some methods require arguments – values to be processed. For example, here is the code to convert a String value to a number of the type of integer:

```java
String testStr = "5000";
int test = Integer.parseInt(testStr);
```

It is said that the method `parseInt()` has one argument of type `String`.

The method `parseInt()` is defined in the Java class `Integer`. Programmers could also define methods that take arguments, for example:

```java
double calcTax(double grossIncome, String state, int dependents) {
    // The code implementing business logic goes here
}
```

Arguments are used in the method body the same way as local variables. The difference is that you do not assign the values to these variables, but assume that the values will be provided by the caller of the method, for example:

```java
double myTax = calcTax(45000.00, "NJ", 2);
```

Program Comments

You can add any text (comments) to your program to explain its functionality.

- Single line comments are represented by two slashes:

  ```
  // This method calculates tax
  ```

- Block or multi-line comments are enclosed in `/* */`, for example:

  ```
  /* This method calculates tax. */
  ```
It takes three arguments: gross income, state, and the number of dependents. */

- Javadoc comments are enclosed in /** */ and help write documentation. Javadoc is a very convenient utility that reads your Java programs and automatically creates their description in the form of an html file. It includes the text placed between /** and */ into documentation files.

## Special Methods: Constructors

Java uses the operator `new` to instantiate classes, for example:

```java
Tax t = new Tax();
```

Parentheses after the word `Tax` mean that we are calling a method from this class. It's a special kind of method called a **constructor**.

Constructors have the following characteristics:

- They are called **only once** when the class is being instantiated.
- They must have the same name as the class itself.
- They do not return a value and you do not have to specify the keyword `void`.

A class can have more than one constructor (see the section “Method Overloading” below).

If you do not create a constructor for the class, Java helps you by using a so called default no-argument constructor. That’s why the Java compiler does not complain about the statement `new Tax()`, even though we have not written a constructor for the class `Tax`.

Constructors are usually used to assign initial values to member variables of the class, for example:

```java
class Tax {
    double grossIncome;   // member variables
    String state;
    int dependents;

    Tax (double gi, String st, int depen){
        grossIncome = gi;   // member variable initialization
        state = st;
        dependents=depen;
    }
}
```
If a constructor with arguments has been defined in a class, you can no longer use a default no-argument constructor – you have to write one.

The **Keyword this**

The keyword **this** is useful when you need to refer to an instance of the class from its method. Let's consider an example:

```java
class Tax {
    double grossIncome;
    Tax(double grossIncome) {
        this.grossIncome = grossIncome;
    }
}
```

The keyword **this** helps avoid name conflicts, for example **this.grossIncome** refers to a member variable **grossIncome**, while the **grossIncome** on the right refers to the argument's value.

Let's look at another example - say you have some class with a method **verifyTax(Tax t)** that needs an instance of class **Tax** as an argument. This is how you can call it from the class **Tax**:

```java
class Tax {
    void myMethod() {
        ...
        SomeOtherClass s = new SomeOtherClass();
        s.verifyTax(this);
    }
}
```

**Method Overloading**

If a class has more than one method with the same name, but with different argument lists, it's called method overloading. For example, the method **print()** could be called with different types of arguments. Actually there are multiple overloaded versions of the method **print()**, it's just easier to remember one method **print()**, than **printString()**, **printInt()**, etc.
Constructors also can be overloaded.

The next version of our class *Tax* will have two overloaded constructors: one with 3 arguments (income, state and dependents), and another one with 2 (income and state). If the 2-argument constructor will be used when the class is instantiated, the default value of 1 dependent is assumed.

```java
class Tax {
    double grossIncome;
    String state;
    int dependents;

    Tax (double gi, String st, int depen){
        grossIncome = gi;
        state = st;
        dependents = depen;
    }

    Tax (double gi, String st){
        grossIncome = gi;
        state = st;
        dependents = 1;  // Default value
    }
    ...
}
```

Only one constructor will be used when a class is instantiated, based on the provided argument list. The example below uses the 3-arguments constructor:

```java
Tax t = new Tax( 50000.00, "NJ", 2);
```

The following example uses 2-argument constructor:

```java
Tax t = new Tax( 50000.00, "NJ");
```

### Arrays

Let's say you have to store names of 20 different girls, such as “Masha”, “Matilda”, “Rosa” etc. Instead of declaring 20 different *String* variables, you can declare one *String* array capable of storing 20 elements:

```java
String [] friends = new String [20];
friends[0] = "Masha";
friends[1] = "Matilda";
friends[2] = "Rosa";
...
The first element of an array in Java always has an index of 0. You can either place brackets after the data type, or after the variable name. Both declarations below are correct:

```java
String friends[];
String [] friends;
```

You must know the size of the array **before** assigning values to its elements. If you do not know this, consider other Java collection classes from the `package java.util` such as `Vector`, `ArrayList`, `Hashtable`, etc.

Arrays have a property `length` that stores its number of elements:

```java
int totalElements = friends.length;
```

If you know all the values that will be stored in the array at the time of its declaration, initialize an array in this manner:

```java
String [] friends = {"Masha", "Matilda", "Rosa", "Sharon"};
```

Say you need to choose a girl for a concert. Just roll the die – you’ve got number 5!

```java
String girlfriend = friends[4]; // Why not 5?
```

Our Array `friends` has one problem – it does not store girl’s phone numbers. Luckily, Java has multidimensional arrays:

```java
String  friends [][] = new String [20][2];
friends[0][0] = "Masha";
friends[0][1] = "732 456-7834";
friends[1][0] = "Matilda";
friends[1][1] = "718 456-7834";
...
friends[19][0] = "Sharon";
friends[19][1] = "212 456-7834"
```

**Loops**

Loops are used to repeat the same action multiple times. When you know in advance how many time you want to repeat this action use the loop `for`, and if you just know the condition of exit from the loop use the loop `while`. Let’s print the names from the one-dimensional array `friends`.

```java
int totalElements = friends.length;
```
int i;
for (i=0; i<totalElements;i++){
    System.out.println("I love " + friends[i]);
}

The code above reads “Print the value of the element $i$ of the array friends starting from $i=0$, and incrementing $i$ by one ($i++$) until $i$ is equal to totalElements”. The same result could have been achieved by using the while loop:

```
int totalElements = friends.length;
int i=0;
while (i<totalElements){
    System.out.println("I love " + friends[i]);
    i++;    // you could replace this line with  i=i+1;
}
```

Actually, the while loops are usually used when you do not know in advance how many times it has to be repeated, for example when reading lines from a file or the database records (see Lessons 8 and 11).

Use the keyword break to prematurely jump out of the loop the line below the ending curly brace. The keyword continue allows to jump up to the first line of the loop (condition checking line).

### Command-Line Arguments

To start a program called TestTax from the command window you have to enter the following:

```
c:\practice>java TestTax
```

If you need to pass some data to this program during the start-up, i.e. gross income, state, and dependents – use the command line arguments:

```
c:\practice>java TestTax 50000 NJ 2
```

The method `main(String[] args)` of the class TestTax receives this data as a `String` array called `args`. The above command line causes the automatic creation of the following array:

```
args[0] = “50000”;
args[1] = “NJ”;
args[2] = “2”;
```
Command-line arguments are always being passed to a program as Strings. It’s the responsibility of the programmer to convert the data to the appropriate data type.

Let’s modify the TestTax class to get rid of hard-coded values and switch to command-line arguments instead:

```java
class TestTax{
    public static void main(String args[]) {
        double grossIncome = Double.parseDouble(args[0]);
        String state = args[1];
        int dependents = Integer.parseInt(args[2]);

        // Let's use the 3-argument constructor of the class Tax
        Tax t = new Tax(grossIncome, state, dependents);

        System.out.println("Your tax is ") + t.calcTax();
    }
}
```

It’s a good idea to put in the beginning of the method `main` the code that checks if the command line contains the correct number of arguments:

```java
public static void main(String args[]) {
    if (args.length != 3) {
        System.out.println("Sample usage of the program:");
        System.out.println(" java TestTax 50000 NJ 2");
        System.exit(0);
    }
}
```

**Assignment.** Re-write the class Tax replacing the 3-argument constructor with the three-argument method `calcTax()`.

### The keyword `super`

The keyword `super` is used to refer to the superclass of an object, for example it could be used to call a constructor of the superclass:

```java
class NJTax extends Tax{
    NJTax(double income, String state) {
        super(income, state, 1);
        ...
    }
}
```
The keyword `super` could also be used to call any method in the superclass. You do not usually use it, because methods of the superclass are available just by specifying their names. Having an overridden method could come in handy - one in the current class and the other one in the superclass. If for any reason you want to call a particular method from a superclass, even though there is a method with the same signature in the subclass, do it as follows:

```java
super.myMethod();
```

**The keyword `final`**

The keyword `final` could have different meanings depending on its location in the Java class:

- If you use it in the front of the variable declaration, its value can not be changed. They are called **final variables**.

  ```java
  final String state = "NJ";
  ```

- If you use it in the front of the class declaration, it can not be subclassed (have descendents).

  ```java
  final class Tax {
      ...
  }
  ```

- If you use it in the front of a method declaration, the method can not be overridden:

  ```java
  final double calcTax(){
      ...
  }
  ```

When the project is finished, it make sense to make all your classes and methods `final` - it could give improve the performance of your application.

**Access Levels**

Java classes, methods and member variables could have **public**, **private**, **protected** and **package** access levels, for example:

```java
public class Tax {
    private double grossIncome;
```
private String state;
private int dependents;
    protected double calcTax(...) {...}
}

The keyword public means that this element (class, method or a variable) could be accessed from any other Java class.

The keyword protected makes the element “visible” either from the current class, or from its subclasses.

The keyword private is the most restrictive one and makes the member variable or a method available only within the current class. For example, our class Tax may need some additional internal method that could be called from the method calcTax(). The users of the class Tax do not need to know about it and this method should be declared as private one.

If you do not specify any access level, the default package access level is used.

One of the main features of object-oriented languages is encapsulation, which is an ability to hide and protect its elements. The classes should expose to their possible users only the necessary methods, i.e. calcTax(). The methods that the class Tax exposes to its prospective clients could be called Application Program Interface (API).

Do you really know what exactly happens under the hood when you start your car’s engine? Do you really want to know? That ignition key slot and the “Check oil” signal are example of your car’s API.

If you are not sure which access level to give to methods or variables, just make them all private. If at the later development stage another class needs to access them, you can always change it, but this is a simple way to protect all the internal of your application from misuse. Think of it this way: “I want to sell my class Tax to various accounting firms across the country. If their programmers will incorporate this class into existing systems - what’s would be methods that they must know about to be able to calculate the tax?” If car designers would not ask themselves a similar question, you’d need to press hundreds of buttons just to start the engine.

Resources

1. The javadoc utility:
http://java.sun.com/j2se/javadoc/
Lesson 4

Graphic User Interface with AWT

Java Applets

Java Applets are programs that could be included in HTML pages and run under control of Web browsers. For example, you want to add a calculator to a Web page. Since HTML is only a mark-up language, it can't create executable programs. Java Applets fill this gap and allow you to create more advanced Web pages.

Java applets are downloaded to the user's computer from the specified URL along with the Web page. Since all Java programs need a JVM to run, Java-capable Web browsers automatically start their own JVM after finding the tag `<Applet>` in the Web page.

Users browse the Internet without knowing if the web page contains a Java applet, but they want to be sure that their disks will not be harmed by some bad guys who added a nasty applet to the page. That’s why the applets were designed with the following security restrictions:

- Applets can not access files on the user’s machine.
- Applets can only connect to the computer they where downloaded from.
- Applets can not execute any of the programs located on the user’s machine.

If you want to use Applets only within your organization, security restrictions could be eliminated by the use of so called digital signatures, which are encrypted files containing information about the applet. JDK comes with tools called jarsigner and keytool that help you with the creation and the administering of the digital signatures. These tools replace the older program called javakey. You could find the Internet links to the articles explaining the whole process in detail at the end of this lesson.
The HTML Tag <Applet>

In this section, I'll explain how to add a Java applet contained in the Calculator.class to an HTML page. Here's a simple Web page called Calculator.html:

```html
<HTML>
<BODY>
Here is my Calculator:
<APPLET code="Calculator.class" width=300 height=100>
</APPLET>
</BODY>
</HTML>
```

The above code fragment will work if the file Calculator.class and the HTML file are located in the same directory.

These are some attributes of the tag <APPLET>:

- **code** - the name of the applet’s Java class.

- **width** - the width (in pixels) of the area on the screen to be used by the applet.

- **height** - the height (in pixels) of the area to be used by the applet.

- **codebase** - location of the applet’s class. This could be either a directory on the same web server, or a remote site.

```html
<applet code="Calculator.class" codebase=AppletsDir width=300 height=100>
</applet>
```

- **archive** - the name of a jar file that contains the applet.

```html
<applet code="Calculator.class" archive=utilities.jar width=300 height=100>
</applet>
```

A Java applet may use some other Java classes, and it makes a lot of sense to put all of them into one archive file using the jar program that comes with JDK. This way, your browser will only need to connect to the remote site once to download the jar with all needed classes.
Java Archives - JARs

Java archives could contain multiple files and have the file name extension .jar. The internal format of jar files is the same as in .zip files, which means that you can use such utilities as winzip from WinZip Computing to see the content of the archive.

The following two commands illustrate the use of the jar utility:

To create a jar that will contain all .class file, enter this in the command window:

c:\practice>jar cvf myClasses.jar *.class

To extract all files from the myClasses.jar, type the following command:

c:\practice>jar xvf myClasses.jar

Writing Applets

Java applets do not need the main() method – Web browsers know how to control its creation and destruction. A browser also sends signals to applets when important events happen (applet is starting, re-painting, etc.). If an applet needs to react to these events, a programmer has to program the applet’s methods such as start(), paint(), etc.

Java applets have to be inherited from the class java.applet.Applet:

class TaxApplet extends java.applet.Applet {
}

The class Applet has five so called callback methods, which means that these methods are not being called by any class that you write, but rather by the environment that the applet operates in – in this case, it is by the Web browser’s JVM. The programmer needs to understand when these methods are called.

- init() - is called when the applet is loaded by the browser. It’s called only once, so it has a usage similar to constructors in regular Java classes.

- start() - is called right after the init(). It is also called if a user returns to the Web page with the applet after visiting another page.
- **paint()** - is called when the applet’s window needs to be displayed or refreshed after some activity on the screen, i.e. the applet is overlapped with some other window. To repaint the applet’s window programmatically, you should call the method `repaint()`. This method internally calls the methods `update()`, and then `paint()`.

- **stop()** - is called when a user leaves the Web page containing the applet.

- **destroy()** – is called when the browser destroys the applet. It’s only used if an applet was using some other resources that need to be removed from memory. The usage of this method is similar to the usage of the `finalize()`.

Here’s a code of the applet that displays the words “Hello World”:

```java
public class HelloWorld extends java.applet.Applet {
    public void paint(java.awt.Graphics graphics) {
        graphics.drawString("Hello World!", 70, 40);
    }
}
```

Perform the following steps to test it:

Step 1. Enter the above code and save it in the file `HelloWorld.java` and compile the class:

```
  c:\practice> javac HelloWorld.java
```

Step 2. Create the file `HelloWorld.html` with the following content:

```html
<HTML><BODY>
Here is my first applet:<P>
<APPLET code="HelloWorld.class" width=200 height=100>
</APPLET>
</BODY></HTML>
```

Step 3. Start your browser and open the file `HelloWorld.html` using the menu File | Open.

Here’s what the screen will look like:
Java also provides a tool to test applets called **appletviewer**. If the applet works in the applet viewer, but does not work in your Web browser, it means that the browser has an older version of JVM.

This applet could also be tested this way:

```
c:\practice>appletviewer HelloWorld.html
```

The output will look like this:
Abstract Windowing Toolkit

Whenever you need to use Graphic User Interface (GUI) components such as buttons, text fields, and others, you could use classes from the java.awt package.
You add components to the containers such as windows, panels, frames, etc. To arrange components inside a container, use layout managers.

Every Abstract Windowing Toolkit (AWT) component is a Java class, and has to be instantiated and added to the container.

```
import java.awt.*;
public class TaxCalcApplet extends java.applet.Applet {
    Label lblGrIncome = new Label("Gross Income: ");
    TextField txtGrossIncome = new TextField(15);
    Label lblDependents = new Label("Number of Dependents: ");
    TextField txtDependents = new TextField(2);
    Label lblState = new Label("State: ");
    Choice chState = new Choice();
    Label lblTax = new Label("State Tax: ");
    TextField txtStateTax = new TextField(10);
    public void init() {
        // Add all components to the Applet
        this.add(lblGrIncome);
        this.add(txtGrossIncome);
        this.add(lblDependents);
        this.add(txtDependents);
        this.add(lblState);
        this.add(chState);
        this.add(lblTax);
        this.add(txtStateTax);

        // Populate the Choice component with 2 states
        chState.addItem("NY");
        chState.addItem("NJ");

        // Make the state tax field non-editable
        txtStateTax.setEditable(false);
    }
}
```

In the sample above, we did not assign any layout manager, so Java used a default one, which is a FlowLayout for applets (see below). Try to compile and test it with your Web browser, but the output screen will look a little ugly:
Besides AWT Java has another set of more advanced Swing components (see Lesson 12), that has similar classes such as JButton, JLabel, JTextField and a lot more. Not all Web browsers have JVM that support Swing.

Layout Managers

Some old-fashioned programming languages force you to assign coordinates to every button, text field and other components that you put on the screen. This works fine if you know what the resolution of your user's monitors is. Layout Managers in Java help you arrange components on the screen without assigning pre-set positions, and will ensure that the screen will look fine in different operational environments, screen sizes, and screen resolutions.

Java AWT offers five layout managers:

- FlowLayout
- GridLayout
- BorderLayout
- CardLayout
- GridBagLayout

To use layout managers, you would first need to create an instance of the selected class:

GridLayout gr = new GridLayout(2,3);
Now assign the layout manager to your Applet, Frame, Dialog or Panel.

```java
this.setLayoutManager(gr);
```

### Flow Layout

This layout places components on the window row by row. Say, buttons will be placed on the first imaginary row till there is not enough room left. When the current row is filled, it'll be moved to the next one, and so on. If the user changes the size of the screen, it may mess up the picture.

In the next sample the keyword `this` represents your GUI class.

```java
FlowLayout fl = new FlowLayout();
this.setLayoutManager(fl);
...
Button b1 = new Button("Hello");
this.add(b1);
```

### Grid Layout

The `GridLayout` allows you to arrange components as rows and columns like a grid. You'll be adding components to imaginary cells of this grid. If the screen will be resized, cells may become larger, but it will not change the relative positions of the components. For example, if you need to place 8 components on the screen, you may consider a grid of 4 rows and 2 columns:

```java
GridLayout gr = new GridLayout(4,2);
```

You can also assign some horizontal and vertical space gaps between the cells, for example 5 pixels:

```java
GridLayout gr = new GridLayout(4,2,5,5);
```

Add the following 2 lines to the beginning of the method `init()` of the `TaxCalcApplet`:

```java
GridLayout gr = new GridLayout(4,2,5,5);
this.setLayout(gr);
```
Now the output screen looks differently:

```
Applet Viewer: TaxCalcApplet2.class

Gross Income:  |
Number of Dependents:  |
State: NY |
State Tax:  |
```

Try to resize the applet by changing the width/size attribute of the tag `<Applet>` - all components keep their relative positions.

If you specify 0 as the number of rows in a grid, Java will use as many rows as needed to accommodate all of the components. The same statement is true for the column number.

Here’s the program that produced the above screen:

```
// TaxCalcApplet2.java - example of GridLayout usage.

import java.awt.*;

public class TaxCalcApplet2 extends java.applet.Applet {
    Label lblGrIncome = new Label("Gross Income: ");
    TextField txtGrossIncome = new TextField(15);
    Label lblDependents = new Label("Number of Dependents: ");
    TextField txtDependents = new TextField(2);
    Label lblState = new Label("State: ");
    Choice chState = new Choice();
    Label lblTax = new Label("State Tax: ");
    TextField txtStateTax = new TextField(10);

    public void init() {
        // Create a layout for the 4 rows and 2 columns
        // with 5 pixel spacing

        GridLayout gr = new GridLayout(4,2,5,5);
        this.setLayout(gr);

        add(lblGrIncome);
        add(txtGrossIncome);
    }

    // Other methods...

    // Other methods...
}
```
add(lblDependents);
add(txtDependents);

add(lblState);
add(chState);

add(lblTax);
add(txtStateTax);

// Populate states
chState.addItem("NY");
chState.addItem("NJ");

// Make the resulting state tax field non-editable
txtStateTax.setEditable(false);
}
}

The next program SampleFrame.java shows various AWT components in a grid layout. Each component has been added with the position specifications in the grid. The grid cell numbers start from the top left corner. This is not an applet, but an independent Java program that could be compiled and started from a command window:

c:\practice>javac SampleFrame.java
c:\practice>java SampleFrame

Do not try to close the window by clicking on the little cross on the top right corner - we did not program it yet. Just press buttons Ctrl and C on your keyboard to close the window. Below is the code and the output screen.

import java.awt.*;
public class SampleFrame extends Frame {
    Label label1 = new Label("A Button:");
    Button button1 = new Button("OK");
    Label label2 = new Label("Checkboxes:");
    Label label3 = new Label("Checkboxes in a group:");
    Checkbox checkboxLife = new Checkbox("Life Insurance",true);
    Checkbox checkboxMed = new Checkbox("Medical Insurance",false);
    CheckboxGroup cbgGender = new CheckboxGroup();
    Checkbox checkboxMale = new Checkbox("Male",cbgGender,true);
    Checkbox checkboxFemale = new Checkbox("Female",cbgGender, false);
36  The Java Tutorial For The Real World

Label label4 = new Label("Choice and List:");
Choice  choiceState = new Choice();
List   listState = new List();
Label label5 = new Label("TextField and TextArea:");
TextField txtField =
    new TextField("Some Long Text", 5);  // 5 char wide
TextArea txtArea = new TextArea(
    "Some very Long Text",5,3,TextArea.SCROLLBARS_NONE);

public SampleFrame(String title) {
    super(title);
    GridLayout gl = new GridLayout(0,3,5,5);
    this.setLayout(gl);
    // First row with labe and button
    this.add(label1,0);
    this.add(button1,1);
    this.add(new Label(""),2);

    // Insurance checkboxes
    this.add(label2,3);
    this.add(checkboxLife,4);
    this.add(checkboxMed,5);

    // Checkboxes in the cbgGender group -
    // mutually exclusive radiobuttons
    this.add(label3,6);
    this.add(checkboxMale,7);
    this.add(checkboxFemale,8);

    // A row with Choice and List components
    choiceState.add("New York");
    choiceState.add("New Jersey");
    this.add(label4,9);
    this.add(choiceState,10);
    listState.add("Delaware");
    listState.add("Connecticut");
    this.add(listState,11);

    // A row with textField and TextArea
    this.add(label5,12);
    this.add(txtField,13);
    this.add(txtArea,14);
}

public static void main(String[] args) {
    SampleFrame myFrame = new SampleFrame("Sample AWT Components");
    myFrame.setSize(500,230);
    myFrame.setVisible(true);
Border Layout

The BorderLayout divides the screen into a South, West, North, East, and Center area. For example, the text field that displays the numbers belongs to the North area.

```java
BorderLayout bl = new BorderLayout();
this.setLayoutManager(bl);
...
TextField txtDisplay = new TextField();
this.add("North", txtDisplay);
```

It's not a must to have all 5 areas on the screen. If you only need the North, Center, and South areas, the Center area will become wider - it will occupy all of the space that is available since you are not using the East and the West sides.

The BorderLayout is a default layout for Frame windows.

Combining Layout Managers

Do you think that the GridLayout will allow you to create a Calculator window that looks like the one that comes with Microsoft Windows (see the picture below)? Unfortunately not, because cells have different sizes (the display field is much wider than the buttons). You could combine layout managers using so-called panels. A panel is an invisible container that holds
other components. It could have its own layout manager. This is how you can start:

Step 1. Assign the border layout to your window.

Step 2. Add your display `TextField` to the North area of the screen.

Step 3. Add components to the `Panel` with the `GridLayout` inside, and then add the `Panel` to your window:

```java
GridLayout gr = new GridLayout(4, 5);
Panel p1 = new Panel();
p1.setLayout(gr);

Button b1 = new Button("1");
Button b2 = new Button("2");
...
p1.add(b1);
p1.add(b2);
...
this.add("East", p1);
```

Write a similar script for the West area.

![Calculator](image)

The complete code of a class `Calculator.java`, is included in the code samples for this lesson.

**GridBag Layout**

Let's re-design the calculator to use the `GridBagLayout` manager instead of combined layouts and panels.
Calculator has a grid structure because it has rows and columns, but in a grid layout, all of the components must have the same size. This does not work for our calculator because there is a text field on the top that has to be wider than the others.

The GridBagLayout is a more advanced grid, that allows you to assign different properties to its cells. One additional class called GridBagConstraints should be used for that.

All the constraints for a cell have to be set before placing a component into it. One of the constraint’s properties is gridwidth. It allows you, for example, to make a cell as wide as 6 other cells. Create an instance of this constraint object first, and then assign the values to its properties. Add your component to the specified cell in your container:

```java
GridBagLayout gb = new GridBagLayout();
this.setLayout(gb);

GridBagConstraints constr = new GridBagConstraints();

// setting the constraints for the Calculator’s text field:
constr.x=0; // x coordinate in the grid
constr.y=0; // y coordinate in the grid
constr.gridheight =1; // this cell has the same height
    // as other cells
constr.gridwidth= 6; //this cell is as wide as 6 other ones
constr.fill= constr.BOTH; // fill all space in the cell
constr.weightx = 1.0;    // proportion of horizontal space
                       // taken by this component
constr.weighty = 1.0;  // proportion of vertical space
                       // taken by this component
constr.anchor= constr.CENTER;  // position of the component
                               // within the cell

Button b1=new Button("1"); //create a button with label "1"

gb.setConstraints(b1, constr); // assign constraints to
     // this button
this.add(b1);     // add the button to the window
Card Layout

Think of a deck of cards laying on top of each other. You can only see the top one. The CardLayout could be used if you need to create a component that looks like a tab folder. Many Properties screens are designed this way. When you click on a tab, the content of the screen changes. In fact, all of the panels needed for this screen are already pre-loaded and laying on top of each other. When the user clicks on a tab, the program just “brings this card” on top and the rest of the cards become invisible.

Most likely you will not be using this layout, cause the Java Swing library includes a more advanced component called JTabbedPane that is used for screens with tabs.

What If I Don’t Like Layout Managers?

If you know exactly the screen sizes/resolutions of the user's monitors, your may assign specific coordinates to the GUI components. In this case, your class has to explicitly reject the use of layout managers:

```java
this.setLayout(null);
```

For each component, the program has to assign the coordinates of its left upper corner, the width, and the height. For example, the following button’s width is 40, height is 20, and it’s located 100 pixels down and 200 pixels to the right from the top left corner of the window:
Version Compatibility Issues

The fact that Web browsers use their own JVM introduces the issue of the Java version compatibility. What if the applet was developed using Java 1.4, but a user has an old Web browser that only supports the Java 1.1 programs? To resolve this issue, Sun Microsystems provides free downloadable Java plug-in files and an HTML Converter to modify HTML files, forcing them to use the plug-ins. The HTML Converter will replace the tag `<Applet>` with the tags `<Object>` and `<Embed>` (see the web link below).

In fact, there is a question during the installation of JSDK 1.4 asking if you’d like to use Java plug-in with your Web browser.

Resources

1. JAR Signing and Verification Tool:
   http://java.sun.com/j2se/1.3/docs/tooldocs/win32/jarsigner.html

2. Sun’s Java Tutorial – Securities Features Overview:
   http://java.sun.com/docs/books/tutorial/security1.2/overview/index.html

3. JAR - The Java Archive Tool:
   http://java.sun.com/j2se/1.4/docs/tooldocs/win32/jar.html

4. The JarIndex tool that improves the efficiency of search in jars:
   http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR Index

5. Using the AWT To Create A GUI:
   http://java.sun.com/docs/books/tutorial/applet/practical/gui.html

6. Using Layout Managers:
   http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html

7. Java Plug-in Download:
   http://java.sun.com/products/plugin/index-1.4.html
Lesson 5

Event Handling. Interfaces. Inner Classes.

Various events may happen to a running program: the user clicks on a button in a window, a program receives a message from another program, a Web browser decides to re-paint the applet’s window, etc.

Originally, Java did not handle these events too efficiently (all events were first sent to the underlying OS and then back to JVM). The events would be delivered to a component even if it did not need to process them. Let’s say a user moves the cursor over a button – the important event is the ActionEvent (click on the button) and not the MouseEvent that would help to find out coordinates of the mouse cursor.

Starting from Java 1.02, a better event-handling model was introduced – now components could only register themselves to the events they are interested in. AWT has so called listeners that allow you to specify, that a particular button “will listen” to mouse clicks (action events) and that some list box is interested in listening to clicks as well as selection changed events, and so on.

In the old event model, GUI components would use a method handleEvent().

The current event model is using special Java classes called interfaces such as ActionListener, WindowListener, KeyListener and others. This event model is also known as the delegation model (see explanation below).

Interfaces

Let’s say we’ve defined several types of possible object behaviors such as ‘will react to button clicks’, ‘will react to mouse movements’, etc. Can our Calculator applet inherit these behaviors? If these additional behaviors would be defined in some Java classes, the answer would be no, cause Java does not allow multiple inheritance and the class Calculator is already inherited from the class Applet.

The good news is that besides regular classes, Java also has interfaces.
A Java class can not be a descendent of multiple classes, but it can **implement** multiple interfaces.

A Java interface is a class with the following restrictions:

- It can contain method signatures, but no method implementations.
- It can only contain declarations of **static final** variables.

Here’s an example of the `ActionListener` interface that defines one method called `actionPerfomed()`. The class `Calculator` guarantees that it will **implement** the `ActionListener` interface.

```java
interface ActionListener{  
    public void actionPerformed(ActionEvent e);  
}
class Calculator extends Applet implements ActionListener{  
...
}
```

My first reaction was that methods are useless if they have no bodies! This is not true. When a class declares that it implements some interface(s), it actually takes as obligation to write code for all methods declared in this interface(s). It’s like a **contract** between the class and the interface – the class promises to write an implementation for the behavior(s) declared in the interface. In the code sample above, the class `Calculator` guarantees to have the method `actionPerformed()`.

If the above paragraph does not make too much sense after the first reading, do not get frustrated – keep reading, or I should say, keep programming.

Java comes with a number of pre-defined interfaces for different needs including AWT event processing (see the section AWT Event Listeners and Adapters below). Programmers can also define their own interfaces. Say employees and consultants work in a firm. Here’s one of the ways to assure that each person in the firm receives payments despite the differences in payroll calculation:

```java
interface Payment{  
    boolean raiseSalary(int employeeID,  
                        double proposedSalary);  
    int setTaxationType(int taxcode);  
}
```

More than one class could implement the `Payment` interface:
class Employee implements Payment{
    ...
    boolean raiseSalary(int employeeID,
                        double proposedSalary){
        // Your code goes here
    }
    int setTaxationType (int taxcode){
        // Your code goes here
    }
}

Consultants usually have hourly rates and increases work differently for them, but the method signature has to be the same as the one declared in the interface Payment. Consultants may also have different taxation rules.

class Consultant implements Payment{
    boolean raiseSalary(int employeeID,
                        double proposedSalary){
        // Your code goes here
    }
    int setTaxationType (int taxcode){
        // Your code goes here
    }
}

As I’ve mentioned above, you could declare constants that are needed in your application in an interface to make them available for any class just by using this interface:

interface MyConstants{
    final String ADDRESS ="123 Main St., New York, NY";
    ...
}

class Tax implements MyConstants {
    public static void main(String args[]){
        ...
        System.out.println("Send payments to "+ ADDRESS);
    }
}

Java does not have global variables, but constant interfaces allow you to imitate them. The sample above could have been re-written using a class with static constants instead of the interface. In this case you’d have to fully qualify the constants, i.e. FirmConstants.ADDRESS.
### AWT Event Listeners and Adapters

The table below contains some of the AWT listeners and adapters followed by explanations of their usage. The listeners are Java interfaces and the adaptors are the predefined classes that implement them. Let's concentrate on the listeners first. The middle column contains method names declared in the corresponding interface (listener). Consult Java documentation for complete list of interfaces.

<table>
<thead>
<tr>
<th>Listener</th>
<th>Methods</th>
<th>Adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionListener</td>
<td>ActionPerfomed(ActionEvent)</td>
<td>None</td>
</tr>
<tr>
<td>AdjustmentListener</td>
<td>AdjustmentValueChanged(AdjustmentEvent)</td>
<td>None</td>
</tr>
<tr>
<td>FocusListener</td>
<td>FocusGained(FocusEvent)</td>
<td>FocusAdapter</td>
</tr>
<tr>
<td></td>
<td>FocusLost(FocusEvent)</td>
<td></td>
</tr>
<tr>
<td>ItemListener</td>
<td>ItemStateChanged(ItemEvent)</td>
<td>None</td>
</tr>
<tr>
<td>KeyListener</td>
<td>keyPressed(KeyEvent)</td>
<td>KeyAdapter</td>
</tr>
<tr>
<td></td>
<td>keyReleased(KeyEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>keyTyped(KeyEvent)</td>
<td>MouseAdapter</td>
</tr>
<tr>
<td>MouseListener</td>
<td>mouseClicked(MouseEvent)</td>
<td>MouseMotionAdapter</td>
</tr>
<tr>
<td></td>
<td>mouseEntered(MouseEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mouseExited(MouseEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mousePressed(MouseEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mouseReleased(MouseEvent)</td>
<td></td>
</tr>
<tr>
<td>MouseMotionListener</td>
<td>mouseDragged(MouseEvent)</td>
<td></td>
</tr>
<tr>
<td>TextListener</td>
<td>mouseMoved(MouseEvent)</td>
<td></td>
</tr>
<tr>
<td>WindowListener</td>
<td>TextValueChanged(TextEvent)</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>WindowActivated(WindowEvent)</td>
<td>WindowAdapter</td>
</tr>
<tr>
<td></td>
<td>WindowClosed(WindowEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WindowClosing(WindowEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WindowDeactivated(WindowEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WindowDeiconified(WindowEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WindowIconified(WindowEvent)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>windowOpened(WindowEvent)</td>
<td></td>
</tr>
</tbody>
</table>

### How To Use AWT Listeners

There are the 3 steps that have to be performed to use the AWT listeners:
• Add an appropriate clause `implements` to your class declaration, for example:

```java
class Calculator implements ActionListener{...}
```

• Let the GUI component know where the listener is by calling the proper method `add...Listener()`, for example:

```java
Button b1 = new Button("1");
b1.addActionListener(this);
```

• Write the implementation for the method(s) declared in the interface, for example:

```java
public void actionPerformed(ActionEvent e){
    Object src = e.getSource();
    if (src == b1){
        // do some processing for the button 1
    } else if (src == b2){
        // do some processing for the button 2
    }
}
```

The term **Delegation Model** means that you can delegate event processing to a different class (not the class with GUI). In the following code snippet, a class does not implement any listeners - it delegates processing to the instance of another class called `MyEventProcessor`:

```java
import java.awt.*;
import java.awt.event.*;

class MyEventProcessor implements ActionListener{
    public void actionPerformed(ActionEvent e){
        // Your code goes here
    }
}

class MyGUI {
    ...
    MyEventProcessor me = new MyEventProcessor();
    Button b1 = new Button("1");
    b1.addActionListener(me);
    ...
}
Below is yet another version of the tax calculation applet from Lesson 4 with added code to process button clicks and calculate the state tax using the class Tax from Lesson 3.

/*
This applet will get the user's input, calculate and display the state tax.
Event processing is done by using the ActionListener interface.
*/
import java.awt.event.*;
import java.awt.*;
public class TaxCalcApplet3 extends java.applet.Applet
    implements ActionListener {
    Label lblGrIncome = new Label("Gross Income: ");
    TextField txtGrossIncome = new TextField(15);
    Label lblDependents =
        new Label("Number of Dependents: ");
    TextField txtDependents = new TextField(2);
    Label lblState = new Label("State: ");
    Choice chState = new Choice();
    Label lblTax = new Label("State Tax: ");
    TextField txtStateTax = new TextField(10);
    Button bGo = new Button("Go");
    Button bReset = new Button("Reset");

    GridLayout gr = new GridLayout(5,2,1,1);

    public void init() {
        setLayout(gr);
        add(lblGrIncome);
        add(txtGrossIncome);

        add(lblDependents);
        add(txtDependents);

        add(lblState);
        add(chState);

        add(lblTax);
        add(txtStateTax);

        // Populate states
        chState.addItem(" ");
        chState.addItem("NY");
        chState.addItem("NJ");
add(bGo);
add(bReset);

// Make the state tax field non-editable
txtStateTax.setEditable(false);

bGo.addActionListener(this);
bReset.addActionListener(this);
}

public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    if (source == bGo ){
        // The Button Go has been clicked
        int grossInc =
            Integer.parseInt(txtGrossIncome.getText());
        String state = chState.getSelectedItem();
        int dependents=
            Integer.parseInt(txtDependents.getText());

        // Creating an instance of the class Tax
        // using the values entered on the screen
        Tax tax = new Tax(dependents,  state, grossInc);

        // Calculate the state tax and display it
        String sTax =
            Double.toString(tax.calcStateTax());
        txtStateTax.setText(sTax);
    }
    else if (source == bReset ){  
        // The Button Reset has been clicked

        txtGrossIncome.setText(" ");
        txtDependents.setText(" ");
        chState.select(" ");
        txtStateTax.setText(" ");
    }
}

Independent Java GUI Applications

Java AWT has such classes as Frame, Dialog, and FileDialog that could be used for independent applications that could be started from the command line.
Frames and dialog boxes look like regular windows. While Frames are really independent windows, dialogs must have a parent window and are usually displayed for a short time to request or display some data.

The code below creates and displays a Frame window:

```java
import java.awt.*;
class Calculator {
    public static void main(String[] args) {
        Frame myWin = new Frame("Calculator");
        BorderLayout bor = new BorderLayout();
        myWin.setLayout(bor);

        TextField txtDisplay = new TextField(20);
        myWin.add("North", txtDisplay);

        // set the window’s size just big enough to display all components
        myWin.pack();
        myWin.setVisible(true);   // Displays the window
    }
}
```

Frames do not have a size and are invisible until they are explicitly created in the program (see the last two method calls above).

**Inner Classes**

As you could have guessed, the inner classes are defined inside other classes, for example:

```java
class Tax{
    double grossIncome;
    int dependents;

    double calcStateTax(){
        TaxOptimizer tOpt = new TaxOptimizer();
        tOpt.optimize(grossIncome, dependents);
    }

    TaxOptimizer getTaxOptimizer(){
        Return new TaxOptimizer();
    }
}
```
class TaxOptimizer{
    int taxCode;
    float someValue2;
    void setTaxCode(int tCode){taxCode=tCode;}
    int optimize(double grossIncome, int dep){...}
}

The class TaxOptimizer is a member inner class and has access to all variables of the class Tax. The fact that the class TaxOptimizer is defined inside of the class Tax just provides better encapsulation of the classes – it's just a way of saying that the classes belong together. After compilation, the class Tax file will produce two output files: Tax.class and Tax$TaxOptimizer.class.

If the inner class has been defined as static, it'll only have access to static variables of the outer class.

The inner class could even be defined inside a method of an outer class. In this case, this local inner class is only available when the outer method is called and it can only access static variables of the top level class.

I've defined the method getTaxOptimizer() to return an instance of the inner class if someone needs it. For example, if a class TaxCalcApplet would need to access the method setTaxCode() from the inner class it could have done it in the following way:

    Tax t = new Tax(2, "NY", 50000);
    Tax.TaxOptimizer tOptimizer = t.getTaxOptimizer();
    tOptimizer.setTaxCode(12345);    ...

The sample above could have been re-written like this:

    Tax t = new Tax(2, "NY", 50000);
    Tax.TaxOptimizer tOptimizer = t.new TaxOptimizer();
    tOptimizer.setTaxCode(12345);

If an inner class does not have a name, it's called anonymous (see below).

How To Use AWT Adapters

Let's see how to provide closing of the frame when a user clicks on that little cross on the window's title bar. By now, we know that a class should
implement the WindowsListener interface that has 7 methods (see the table above). I'd need to write the implementations for all of them even though only the method windowClosing() will contain code.

Java AWT provides so called adapters, which are classes that have already implemented all required methods (these methods are empty-bodied). This means we can just override the methods we are interested in - the method windowClosing() in this case. We are delegating the processing of our frame's event to this adaptor. The rest is easy – just register this class as a listener of our frame's Window events.

```java
this.addWindowListener(thatClass);
```

To have even more fun, instead of doing the boring inheritance:

```java
class ThatClass extends WindowAdapter{...},
```

we'll kill 3 birds with one stone: define, instantiate, and register the listener using the anonymous inner class. Just look at this beauty:

```java
this.addWindowListener(
    new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
```

The `addWindowListener()` requires a subclass of the `WindowAdapter` and we define it with one overridden method `windowClosing()`. At the same time, we instantiate it using `new WindowAdapter() {...}`. This class does not have a name and will be defined inside some other class (TaxFrame in our case) – that's why we call it an anonymous inner class. Anonymous inner classes could not have constructors, because they do not have names.

The working example of the class TaxFrame is included in this lesson's code samples. I did it as a frame because applets can't be closed. Please note the additional file `TaxFrame$1.class` that will be created by the java compiler. This is a compiled version of our anonymous class. Should you have one more anonymous class, you'll find yet another file called `TaxFrame$2.class`, etc.

The AWT listeners that have only one method defined that does not have corresponding adapters since they would not make coding simpler.

**Resources**

1. AWT event listeners’ documentation:
http://java.sun.com/j2se/1.3/docs/api/java/awt/event/package-summary.html

2. AWT Documentation
http://java.sun.com/j2se/1.4/docs/guide/awt/index.html

3. Inner classes specification
Lesson 6
Exceptions

Every program should perform error processing. Let’s say a class reads a file with the customers’ data. What’s going to happen if someone deletes this file? Will the program crash with that scary multi-line error message, or will it stay alive displaying a user friendly message like this one: “Dear friend, for some reason I could not read the file customer.txt. Please make sure that the file exists”? In many programming languages, error processing depends on the programmer’s mood.

Java forces a programmer to include error processing code, otherwise the programs will not even compile. Error processing in the Java world is called Exceptions Handling.

You have to place code that may produce errors in a `try/catch` block:

```java
... try {
    fileCustomer.read();
} catch (IOException e) {
    System.out.println("Dear friend, I could not read the file customer.txt...");
}
```

The `IOException` is a class that contains information about input/output errors.

In case of an error, the method `throws an exception`. If the `catch` block exists for this type of error, the exception will be `caught` and the statements from a catch block will be executed. The program will not terminate and this exception is considered to be taken care of.

The print statement from the code above will be executed only in case of the file read error.
Reading the Stack Trace

If an unexpected exception occurs that’s not handled by the program, it prints multiple error messages on the screen. This is called **stack trace**. If a program performs a number of nested method calls to reach the problematic line, a stack trace can help you trace the workflow of the program, and localize the error.

Let’s write a program that will intentionally divide by zero:

```java
class TestStackTrace{
    TestStackTrace()
    {
        divideByZero();
    }

    int divideByZero()
    {
        return 25/0;
    }

    static void main(String[]args)
    {
        new TestStackTrace();
    }
}
```

Below is an output of the program – it traced what happened in the program stack before the error had occurred. Start reading it from the last line going up. It reads that the program was executing methods `main()`, `init()` (constructor), and `divideByZero()`. The line numbers 14, 4 and 9 indicate where in the program these methods were called. After that, the `ArithmeticException` had been thrown – the line number nine tried to divide by zero.

```
c:	emp>java TestStackTrace

Exception in thread "main"
    java.lang.ArithmeticException: / by zero
     at TestStackTrace.divideByZero(TestStackTrace.java:9)
     at TestStackTrace.<init>(TestStackTrace.java:4)
     at TestStackTrace.main(TestStackTrace.java:14)
```
Exception Hierarchy

Exceptions in Java are also classes with the following inheritance hierarchy:

Subclasses of the class Exception are called listed exceptions and have to be taken care of in your classes.

Subclasses of the class Error are fatal JVM errors and the running program can't fix them.

The BadSocialSecurityNumberException is an example of a user-defined exception.

How is a programmer supposed to know in advance if some Java method may throw a particular exception and the try/catch block should be used? Don't worry – if a method throws an exception, the Java compiler will print an error message similar to this one:

"Tax.java": unreported exception: java.io.IOException; must be caught or declared to be thrown at line 57

Try/Catch Block

There are 5 Java keywords that could be used for exceptions handling: try, catch, finally, throw, and throws.

By placing a code in a try/catch block, a program says to a JVM: “Try to execute this line of code, and if something goes wrong, and this method
throws exceptions, please catch them, so that I could report this situation to a user.
One try block can have multiple catch blocks, if more than one problem occurs. For example, when a program tries to read a file, the file may not be there - FileNotFoundException, or it's there, but the code has reached the end of the file - EOFException, etc.

```java
public void getCustomers(){
    try{
        fileCustomers.read();
        }catch(FileNotFoundException e){
            System.out.println("Can not find file Customers");
        }catch(EOFException e1){
            System.out.println("Done with file read");
        }catch(IOException e2){
            System.out.println("Problem reading file "+
            e2.getMessage());
        }
    }
}
```

If multiple catch blocks are handling exceptions that have a subclass-superclass relationship (i.e. EOFException is a subclass of the IOException), you have to put the catch block for the subclass first.

A lazy programmer would write the above code in the following way:

```java
public void getCustomers(){
    try{
        fileCustomers.read();
        }catch(Exception e){
            System.out.println("Problem reading file "+
            e.getMessage());
        }
    }
}
```

Catch blocks receive an instance of the Exception object that contains a short explanation of a problem, and its method `getMessage()` will return this info. If the description of an error is not clear, try the method `toString()` instead.

If you need more detailed information about the exception, use the method `printStackTrace()`. It will print all internal method calls that lead to this exception (see the section “Reading Stack Trace” above).
Clause throws

In some cases, it makes more sense to handle an exception not in the method where it happened, but in the calling one. Let’s use the same example that reads a file. Since the method read() may throw an IOException, you should either handle it or declare it:

```java
class CustomerList{
    void getAllCustomers() throws IOException{
        ...
        file.read(); // Do not use try/catch if you are not handling exceptions here
    }

    public static void main(String[] args){
        System.out.println("Customer List");
        ...
        try{
            // Since the getAllCustomers() declared exception,
            // we have to either handle it over here, or re-
            // throw it (see the throw clause explanation below)
            getAllCustomers();
        }
        catch(IOException e){
            System.out.println("Sorry, the Customer List is not available");
        }
    }
}
```

In this case, the IOException has been propagated from the getAllCustomers() to the main() method.

Clause finally

A try/catch block could be exited in different ways

- the code inside the try block successfully ended and the program continues,
- the code inside the try block ran into a return statement and the method is exited,
- an exception has been thrown and code goes to the catch block, which throws another exception
If there is a piece of code that must be executed regardless of the success or failure of the code, put it under the clause `finally`:

```java
try{
    file.read();
} catch(Exception e){
    printStackTrace();
} finally{
    ...
    file.close();
    ...
}
```

The code above will definitely close the file regardless of the success of the read operation. The `finally` clause is usually used for the cleanup/release of the system resources.

If you are not planning to handle exceptions in the current method, they will be propagated to the calling one. In this case, you can use the `finally` clause without the `catch` clause:

```java
void myMethod () throws IOException{
    try{
        ...
        file.read();
    } finally{
        ...
        file.close();
    }
}
```

Let's return to the Tax program from Lesson 5. Run the TaxFrame class and enter `abcd` in the Gross Income field. Press the button “Go” and check the console screen - here’s what you’ll see:

```
Exception occurred during event dispatching:
java.lang.NumberFormatException: abcd
    at java.lang.Integer.parseInt(Integer.java:409)
    at java.lang.Integer.parseInt(Integer.java:458)
    at TaxFrame.actionPerformed(TaxFrame.java:70)
    at java.awt.Button.processActionEvent(Button.java:308)
    at java.awt.Button.processEvent(Button.java:281)
```
This was an example of a non-handled exception. Let's handle it and display a user-friendly message. We'll apply the existing code, processing the button “Go” events in the try/catch block:

```java
try{
    int grossInc = Integer.parseInt(txtGrossIncome.getText());
    int dependents = Integer.parseInt(txtDependents.getText());
    String state = chState.getSelectedItem();
    Tax tax = new Tax(dependents, state, grossInc);
    String sTax = Double.toString(tax.calcStateTax());
    txtStateTax.setText(sTax);
}catch(NumberFormatException e){
    txtStateTax.setText("Non-Numeric Data");
}
```

We've got rid of the error message on the system output screen and displayed a simple message "Non-Numeric Data" in the tax field.

### Clause throw

If an exception has occurred in a method, you may want to catch it and re-throw it to the method’s caller. Sometimes, you might want to catch one exception but re-throw another one that has a different description of the error (see the code snippet below).

The `throw` statement is used to throw Java objects. The object that a program throws must be `Throwable` (you can throw a ball, but you can’t throw a grand piano). This technically means that you can only throw subclasses of the `Throwable` class, and all Java exceptions are its subclasses:

```java
class CustomerList{
    void getAllCustomers() throws Exception{
        ...
        try{
            file.read(); // this line may throw an exception
        } catch (IOException e) {
```
// Perform some internal processing of this error, and …
  throw new Exception {
    "Dear Friend, the file has problems..." +
    e.getMessage();
  }
}

public static void main(String[] args){
  System.out.println("Customer List");
  ...
  try{
    // Since the  getAllCustomers() declares an
    // exception, we have to either handle it,
    // or re-throw it
    getAllCustomers();
  } catch(Exception e){
    System.out.println(e.getMessage());
  }
}

User-Defined Exceptions

Programmers could also create user-defined exceptions, specific to the business. These classes have to be subclasses of one of the exception classes. Let’s say you are in business selling bikes and need to validate a customer’s order. Create a new class TooManyBikesException, and if someone tries to order more bikes than you could ship – throw it:

class TooManyBikesException extends Exception{
  TooManyBikesException (String msgText){
    super(msgText);
  }
}

class BikeOrder{
  ...
  static void validateOrder(String bikeModel, int quantity) throws TooManyBikesException{
    // perform some data validation, and if you do not like
    // the quantity or model, do the following:
    throw new TooManyBikesException("Can not ship" +
      quantity+" bikes of the model "+ bikeModel +);
  }
}
class OrderWindow extends Frame{

    void actionPerformed(ActionEvent e){
        // the user clicked on the “Validate Order” button
        try{
            bikeOrder.validateOrder("Model-123", 50);

            // the next line will be skipped in case of exception
            txtResult.setText("Order is valid");
        } catch(TooManyBikes e){
            txtResult.setText(e.getMessage());
        }
    }
}
Lesson 7

Java Streams

Introduction

Most of the programs work with data which could be provided by a database, remote computer, or a file located on your disk. Java has a concept of working with so called *streams* of data. A program reads data from a stream *serially* – byte after byte, character after character, etc. No wonder that there are different types of streams in Java: byte streams (InputStream, OutputStream), character streams, i.e. Reader and Writer. To work with files, you may consider such classes as FileInputStream, FileReader.

Classes that work with streams are located in the package `java.io`. Java 1.4 has introduced the new package `java.nio` with improved file I/O performance. There are different types of data hence different types of streams.

Here's the sequence of actions needed to work with a stream:

- Open a stream that points at a specific data source: a file, a socket, URL, etc.
- Read or write data from/to this stream.
- Close the stream.

If a program uses one of the third party programs such as database management systems, you don't need to program streams directly (SQL language is all you need), but streams will be automatically created behind the scenes.

All samples in this lesson assume that you use Java from a command window. If you use one of the popular IDE like JBuilder (Borland) or VisualAge For Java (IBM), they make complain about not finding the file. These tools may have different understanding of what's the current directory. You could either use the absolute file name, like (c:\practice\myData.txt). Java has a system property `user.dir`, and you can find out what's the current directory during the run time by calling `System.getProperty("user.dir")`. 
Byte Streams

If a program needs to read/write bytes (8-bit data), it could use one of the subclasses of InputStream or OutputStream respectively. The example below shows how to use the class FileInputStream to read a file named abc.dat. This code snippet prints each byte's value:

```java
FileInputStream myFile = null;
try {
    myFile = new FileInputStream("abc.dat");
    boolean eof = false;

    while (!eof) {
        int byteValue = myFile.read();
        System.out.print(byteValue + " ");
        if (byteValue == -1)
            eof = true;
    }
    myFile.close(); // do not do it here!!!
} catch (IOException e) {
    System.out.println("Could not read file: " + e.toString());
} finally {
    try{
        myFile.close();
    } catch (Exception e1){
        e1.printStackTrace();
    }
}
```

Please note that the stream is closed in the clause finally. Do not call the method close() inside of the try/catch block right after the file reading is done. In case of exception during the file read, the program would jump over the close(); statement and the stream would never be closed!

The next code fragment writes into a file called xyz.dat using the class FileOutputStream:

```java
// byte values are represented by integers from 0 to 255
int somedata[] = {56,230,123,43,11,37};
FileOutputStream myFile = null;
try {
    myFile = new FileOutputStream("xyz.dat");
    for (int i = 0; i < some data.length; i++){
```
Assignment. Write a file copy program by combining the code fragments above. Open 2 streams (input and output) and call `read()` and `write()` in the same loop. Create a window to allow users select file names using the class `java.awt.FileDialog`. The `FileDialog` window should pop up when the user clicks the button `Browse` and selected file name should be displayed in a text field.

Buffered Streams

So far we were reading and writing one byte at a time. Disk access is much slower than the processing performed in memory that's why it's not a good idea to access disk 1000 times for reading a file of 1000 bytes. To minimize number of disk access Java provides so called buffers which are sort of "reservoirs of data".

The class `BufferedInputStream` works as a middle man between the `FileInputStream` and the file itself. It reads a big chunk of bytes from a file in one shot into memory, and, then the `FileInputStream` will read single bytes from there. The `BufferedOutputStream` works in a similar manner with the class `FileOutputStream`.

Buffered streams are not changing the type of reading – they just make reading more efficient.

Use stream chaining (or stream piping) to connect streams – think of connecting two pipes in plumbing. Let's modify the example that reads a file:

```java
FileInputStream myFile = null;
BufferedInputStream buff =null
try {
    myFile = new FileInputStream("abc.dat");
```
BufferedInputStream buff = new
BufferedInputStream(myFile);
boolean eof = false;
while (!eof) {
    int byteValue = buff.read();
    System.out.print(byteValue + " ");
    if (byteValue == -1)
        eof = true;
}
} catch (IOException e) { … }
finally{ … buff.close(); myFile.close(); }

It's a good practice to call the method flush() when writing into a
BufferedOutputStream is done. This forces any buffered output bytes to
be written out to the underlying output stream.

While the default buffer size varies depending on the OS, it could be
controlled. For example, to set the buffer size to 5000 bytes do this:

BufferedInputStream buff = new BufferedInputStream(myFile, 5000);

Character Streams

The 2-bytes characters represent text data in Java. The classes FileReader
and FileWriter work with text files. These classes allow you to read files
either one character at a time with read(), or one line at a time with
readLine().

The classes FileReader and FileWriter classes also have their
counterparts BufferedReader and BufferedWriter.

FileReader myFile = null;
BufferedReader buff = null;
try {  
    myFile = new FileReader("abc.txt");
    buff = new BufferedReader(myFile);
    boolean eof = false;
    while (!eof) {
        String line = buff.readLine();
        if (line == null)
            eof = true;
        else
            System.out.println(line);
    }
}
There are several overloaded methods `write()` that allow you to write one character, one `String` or an array of characters at a time.

To **append** data to an existing file while writing, use the 2-arguments constructor (the second argument toggles the append mode):

```java
FileWriter fOut = new FileWriter("xyz.txt", true);
```

Below is yet another version of the tax calculation program. This time I've added a text file with states that will be used to populate a dropdown box `chStates`.

```java
import java.awt.event.*;
import java.awt.*;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
public class TaxFrameFile extends java.awt.Frame implements ActionListener {
    Label lblGrIncome;
    TextField txtGrossIncome = new TextField(15);
    Label lblDependents = new Label("Number of Dependents: ");
    TextField txtDependents = new TextField(2);
    Label lblState = new Label("State: ");
    Choice chState = new Choice();
    Label lblTax = new Label("State Tax: ");
    TextField txtStateTax = new TextField(10);
    Button bGo = new Button("Go");
    Button bReset = new Button("Reset");

    TaxFrameFile() {
        lblGrIncome = new Label("Gross Income: ");
        GridLayout gr = new GridLayout(5,2,1,1);
        setLayout(gr);

        add(lblGrIncome);
        add(txtGrossIncome);
        add(lblDependents);
        add(txtDependents);
        add(lblState);
        add(chState);
        add(lblTax);
        add(txtStateTax);
        add(bGo);
        add(bReset);
    }
}
```
// Populate states from a file
populateStates();

txtStateTax.setEditable(false);

bGo.addActionListener(this);
bReset.addActionListener(this);

// Define, instantiate and register a WindowAdapter
// to process windowClosing Event of this frame

this.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.out.println("Good bye! ");
        System.exit(0);
    }
});

public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    if (source == bGo) {
        // The Button Go processing
        try {
            int grossInc =
                Integer.parseInt(txtGrossIncome.getText());
            int dependents =
                Integer.parseInt(txtDependents.getText());
            String state = chState.getSelectedItem();
            Tax tax = new Tax(dependents, state, grossInc);
            String sTax =
                Double.toString(tax.calcStateTax());
            txtStateTax.setText(sTax);
        } catch (NumberFormatException e) {
            txtStateTax.setText("Non-Numeric Data");
        } catch (Exception e) {
            txtStateTax.setText(e.getMessage());
        }
    } else if (source == bReset) {
        // The Button Reset processing
        txtGrossIncome.setText(" ");
        txtDependents.setText(" ");
        chState.select(" ");
        txtStateTax.setText(" ");
    }
}

// This method will read the file states.txt and
// populate the dropdown chStates
private void populateStates()
    {
        FileReader myFile = null;
        BufferedReader buff = null;
        try {
            myFile = new FileReader("states.txt");
            buff = new BufferedReader(myFile);

            boolean eof = false;
            while (!eof) {
                String line = buff.readLine();
                if (line == null)
                    eof = true;
                else
                    chState.add(line);
            }
        } catch (IOException e) {
            txtStateTax.setText("Can't read states.txt");
        }
        finally {
            // Closing the streams
            try {
                buff.close();
                myFile.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String args[])
    {
        TaxFrameFile taxFrame = new TaxFrameFile();
        taxFrame.setSize(400,150);
        taxFrame.setVisible(true);
    }

Data Streams

If you are expecting to work with a stream of a known data structure, i.e. two integers, three floats and a double, use either the DataInputStream or the DataOutputStream. A method call readInt() will read the whole integer number (4 bytes) at once, and the readLong() will get you a long number (8 bytes).

The DataInputStream is nothing but yet another filter. We are building a "pipe" from the following fragments:
FileStream ➔ BufferedInputStream ➔ DataInputStream

FileStream myFile = new FileInputStream("myData.dat");
BufferedInputStream buff = new BufferedInputStream(myFile);
DataInputStream data = new DataInputStream(buff);

try {
    int num1 = data.readInt();
    int num2 = data.readInt();
    float num2 = data.readFloat();
    float num3 = data.readFloat();
    float num4 = data.readFloat();
    double num5 = data.readDouble();
} catch (EOFException eof) {...}

Class StreamTokenizer

Sometimes you need to parse a stream without knowing in advance what data types you are getting. In this case you want to get each “piece of data” (token) based on the fact that the data elements are separated by a delimiter such as a space, comma, etc.

The class java.io.StreamTokenizer reads tokens one at a time. It can recognize identifiers, numbers, quoted strings, etc. Typically an application creates an instance of this class, sets up the rules for parsing, and then repeatedly calls the method nextToken() until it returns the value TT_EOF (end of file).

Let’s write a program that will read and parse the file customers.txt distinguishing strings from numbers.

Suppose we have a file customers.txt with the following content:

John Smith  50.24
Mary Lou  234.29
Alexander Popandopula  456.11

Here is the program that parses it:

import java.io.StreamTokenizer;
import java.io.FileReader;

public class CustomerTokenizer{
    public static void main(String args[]){

StreamTokenizer stream =null;
try{
    stream = new StreamTokenizer( new FileReader("customers.txt"));
    while (true) {
        int token = stream.nextToken();
        if (token == StreamTokenizer.TT_EOF)
            break;
        if (token == StreamTokenizer.TT_WORD) {
            System.out.println("Got the string: " +
                stream.sval);
        }
        if (token == StreamTokenizer.TT_NUMBER) {
            System.out.println("Got the number: " +
                stream.nval);
        }
    }
    }catch (Exception e){
        System.out.println("Can't read Customers.txt: "+
            e.toString());
    }
    finally{
        try{
            stream.close();
        }catch(Exception e){e.printStackTrace();}
    }
}

After compiling and running the program CustomerTokenizer, the system console will look like this:

C:\Lesson7\practice>javac CustomerTokenizer.java
C:\Lesson7\practice>java CustomerTokenizer
Got the string: John
Got the string: Smith
Got the number: 50.24
Got the string: Mary
Got the string: Lou
Got the number: 234.29
Got the string: Alexander
Got the string: Popandopula
Got the number: 456.11

When a StreamTokenizer finds a word, it places the value into the sval member variable, and the numbers are placed into the variable nval.
You can specify characters that should be treated as delimiters by calling the method whitespaceChars(). The characters that represent quotes in the stream are set by calling the method quoteChar().

To make sure that certain characters are not misinterpreted, call a method ordinaryChar(), for example ordinaryChar('/');

**Class StringTokenizer**

The class `java.util.StringTokenizer` is a simpler version of a class `StreamTokenizer`, but it works only with strings.

The set of delimiters could be specified at the creation time, i.e. comma and angle brackets:

```java
StringTokenizer st = new StringTokenizer("<HTML>Yakov, 12 Main St., New York", ",<>");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

The above code fragment would print the following:

HTML
Yakov
12 Main St.
New York

The previous sample would not return the value of a delimiter – it just returned the tokens. But sometimes, in case of multiple delimiters, you may want to know what’s the current delimiter. The 3-argument constructor will provide this information:

```java
StringTokenizer st=new StringTokenizer("<HTML>IBM..price<...>86.3", ",", true);
```

If the third argument is true, delimiter characters are also considered to be tokens and will be returned, so a program may apply different logic based on the delimiter.

**Assignment.** Write a program to remove HTML tags from any .html file using the class `StringTokenizer`. 
Class File

This class has a number of useful file maintenance methods which allow rename, delete, perform existence check, etc. First you have to create an instance of this class:

```java
File myFile = new File("abc.txt");
```

The line above does not actually create a file – it just creates an instance of the class `File` class. The method `createNewFile()` should be used for the actual file creation.

Below are some useful methods of the class `File`:

- `createNewFile()` creates a new, empty file named according to the file name used during the `File` instantiation. It creates a new file only if a file with this name does not exist
- `delete()` deletes file or directory
- `renameTo()` renames a file
- `length()` returns the length of the file in bytes
- `exists()` tests whether the file with specified name exists
- `list()` returns an array of strings naming the files and directories in the specified directory
- `lastModified()` returns the time that the file was last modified
- `mkDir()` creates a directory

The code below creates a file "customers.txt" and renames it to "customers.txt.bak". If a file with such name already exists, it will be overwritten.

```java
File file = new File("customers.txt");
File backup = new File("customers.txt.bak");
if (backup.exists()){
    backup.delete();
}
file.renameTo(backup);
```

Java Serialization - Object Streams

Let's consider the following scenario: the class `A` creates an instance of the class `Employee` which has fields like First Name, Last Name, Address, Hire Date, Salary, etc. The values of these fields (object’s state) have to
be saved in a file or some other stream. Later on the Class B, that needs these data, can re-create the object Employee in memory.

We could have done it by using one of the streams like DataOutputStream, FileWriter or others. In this case both programs would need to know the format of the saved file (types and order of the fields, delimiters, etc.). The same result could be achieved in a more elegant way called Java serialization, which allows saving objects in a stream in one shot.

The Class A will serialize and the Class B will deserialize the instance of the object in question. To have this feature the classes must implement Serializable interface:

class Employee implements java.io.Serializable{
    String lName;
    String fName;
    double salary;
    ...  
}

The good news is that the interface Serializable does not define any methods hence there is nothing to implement.

**Class ObjectOutputStream**

While it may not be too difficult to convert a variable int into 4 bytes for serialization, it’s not as simple in case of classes containing variables of non-primitive data types (references to other objects). The process of converting complex object into bytes is called marshalling and the process of reconstructing on the objects from bytes is called unmarshalling and Java does this job for you.

To serialize an object into a stream perform the following actions:
- Open an output stream
- Chain it with the ObjectOutputStream
- Call the method writeObject() providing the instance of a Serializable object as an argument.
- Close the stream

class ClassA {
    public static void main(String args[]) {
        Employee emp = new Employee();
        emp.lName = "John";
        emp.fName = "Smith";
        emp.salary = 50000;
    }
}
```java
FileOutputStream fOut = new FileOutputStream("c:\practice\BestEmployee.ser");

ObjectOutputStream oOut = new ObjectOutputStream(fOut);  
    oOut.writeObject(emp);  //serializing emp

    oOut.flush();
    oOut.close();
    fOut.close();
}
```

All Java primitive data types are serializable. All member variables of the serializable class must be either Java primitives or reference variables pointing to objects that are also serializable. If you do not want to serialize some sensitive information such as salary, declare this variable using the keyword transient:

```java
transient double salary;
```

Static and transient member variables are not serialized.

### Class ObjectInputStream

To de-serialize an object perform the following:

- Open an input stream
- Chain it with the ObjectInputStream
- Call the method `readObject()` and cast the returned object to the class that is being deserialized.
- Close the stream

```java
class ClassB {
    public static void main(String args[]){
        ... 
        FileInputStream fIn = new FileInputStream("c:\practice\BestEmployee.ser");
        
        ObjectInputStream oIn = new ObjectInputStream(fIn);
        Employee bestEmp=(Employee)oIn.readObject(); //cast it
        
        oIn.close();
        fIn.close();
    }
}
```
During the process of de-serialization all transient variables will be initialized with the default value for their type, i.e. int variables will have the value of zero.

Surprisingly enough, the member variables with longer names produce larger footprint when the class is serialized – this may slow down the application, for example, if the high volume of orders is being serialized over the network.

**Interface Externalizable**

The method writeObject() puts all object’s fields into a stream. It’s easy to use, but could lead to unnecessary large resulting streams. If you are willing to do more coding to have smaller “class footprint” and more control over what is being serialized, use the Externalizable interface which is a subclass of Serializable.

This interface defines 2 methods: readExternal() and writeExternal(). The programmer has to implement these methods where he/she should call methods similar to Data stream ones: readInt(), writeInt(), etc. All fields have to be written an read in the same order.

```java
class Employee implements java.io.Externalizable {
    String lName;
    String fName;
    int id;
    double salary;
    ...
    public void writeExternal(ObjectOutput stream)
        throws java.io.IOException {
        // Serializing only the salary and id
        stream.writeDouble(salary);
        stream.writeInt(id);
    }
   ...
    public void readExternal(ObjectInputStream stream)
        throws java.io.IOException {
        salary = readDouble();  // Order or reads must be the
        id  = readInt();        // same as the order of writes
        ...
    }
    ...
}
Lesson 8

Java Network Programming

Introduction To Networking

“The network is the computer” is the corporate motto of Sun Microsystems and Java is an excellent tool to make a network programming not as complex as it sounds. Computers talk to each other using network languages called protocols, that basically define if the data should be send in pieces, how to acknowledge received data, should the connection between two computers remain open, etc.

There are various types network protocols - TCP/IP, UDP/IP, FTP, HTTP and others. Java provides classes for network programming in the package java.net.

A group of connected computers in the same building is called Local Area Network (LAN). Add to it computers from a another city and you’ll get a Wide Area Network (WAN). Connect multiple networks using TCP/IP protocol and you’ve got the Internet. If these networks belong to the same organization we call it the Intranet. For security reasons intranets are shielded from the rest of the world by special software called a firewall.

The World Wide Web (WWW) is using Unified Resource Locators (URL) to find the resources online. Here’s the example of a URL:

protocol host name port number file name.

http://www.smartdataprocessing.com:80/training.html

The host name must be unique and is automatically converted to an IP Address by your Internet Service Provider (ISP). IP address is a group of four numbers, for example 122.65.98.11.
The port is just a unique number assigned to a server program running on the machine. Any computer may have multiple servers running at the same time, and, to make sure that clients connect to the right one, not only the host name, but also the port number has to be specified. It’s somewhat similar to finding a person in an apartment building – it’s not enough to know the street number of the building we need to know the apartment number.
Multiple Java technologies exist for providing data exchange between computers in a network. This lesson shows how to read data from the Internet using class URL, Java Socket Programming, and JavaMail API. The lessons 14-21 will introduce you to other technologies such as Java Servlets, RMI, EJB and JMS.

Reading Data From the Internet

To read local file streams, a program has to specify the file's location, i.e. “c:\practice\training.html”. The same procedure is valid for reading the remote files – just open the stream over the network. Java has a class URL that will help you to connect to a remote computer on the Internet.

At first, create an instance of the URL:

```java
try{
    URL xyz = new URL("http://www.xyz.com:80/training.html");
    ...
}
catch(MalformedURLException e){
    e.printStackTrace();
}
```

The MalformedURLException could be thrown if a non-valid URL has been specified, i.e. htp instead of http, extra spaces, etc. If the MalformedURLException has been thrown, it does not indicate the remote machine problems – just check the “spelling” of the URL.

Creation of the URL object does not establish connection with the remote machine – you'll still need to open a stream and read it. Perform the following steps to read a file from the Internet:

Step 1. Create and instance of the class URL

Step 2. Create an instance of the URLConnection class and open a connection using the URL from the previous step

Step 3. Get a reference to an input stream of this object by calling the method URLConnection.getInputStream()
Step 4. Read the data from the stream. Use buffered reader to speed up the process.

Since the streams from the package `java.io` are being used for read/write operations, you'll have to handle the I/O exceptions the same way you did while reading the local files.

The server you are trying to connect to has to be up and running and, in case of using http protocol, the special software (Web Server) has to be “listening to” the specified port on the server. By default, Web servers are listening to the port number 80.

The program below reads and prints the content of the file index.html from www.yahoo.com on the system console. To test this program your computer has to be connected to the Internet. The program may not work properly if your connection goes through the firewall.

```java
import java.net.*;
import java.io.*;
public class WebSiteReader {
    public static void main(String args[]){
        String nextLine;
        URL url = null;
        URLConnection urlConn = null;
        InputStreamReader inStream = null;
        BufferedReader buff = null;
        try{
            // index.html is a default URL's file name
            url  = new URL("http://www.yahoo.com");
            urlConn = url.openConnection();
            inStream = new InputStreamReader( urlConn.getInputStream());
            buff  = new BufferedReader(inStream);
            // Read and print the lines from index.html
            while (true){
                nextLine =buff.readLine();
                if (nextLine !=null){
                    System.out.println(nextLine);
                } else{
                    break;
                }
            }
        } catch(MalformedURLException e){
            System.out.println("Please check the URL: " + e.toString() );
        } catch(IOException  e1){
```
The class WebSiteReader explicitly creates the URLConnection object. Strictly speaking we could get away just with the use of the class URL:

```
URL url = new URL("http://www.yahoo.com");
InputStream in = url.getInputStream();
Buff= new BufferedReader(new InputStreamReader(in));
```

The reason why you may consider using the URLConnection class is that it could give you some additional control over the I/O process. For example, by calling its method `setDoInput(true)` you could allow (or disallow) downloads.

### Connecting Through HTTP Proxy Servers

Most of the companies use firewalls for security reasons and their employees reach the outside Internet community through HTTP proxy servers. Check the settings of your Internet browser to find out the host name and port number of the proxy server. If you are using Microsoft Internet Explorer check the menu Internet options | Connections | LAN Setting. Netscape Navigator has proxy settings under Preferences | Advanced | Proxies.

While Java Applets know parameters of the proxy servers, independent Java program should set the following system properties:

```
System.setProperty("proxyHost","http://sdp.com");
System.setProperty("proxyPort", 8080);
```

If you do not want to hardcode these value, pass them to the program from the command line:

```
c:\practice>java  –Dhttp.proxyHost=http://sdp.com  
–Dhttp.proxyPort=8080  WebSiteReader
```
How to Download Files From the Internet

If we combine the class URL with reading files techniques, we should be able to download practically any file (images, music, binary files) from the Internet. The trick is in proper opening of the file stream. Let's write the class FileDownload which takes the URL and the file name as a command line arguments and copies it into a local file.

```java
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.net.URL;
import java.net.URLConnection;

class FileDownload{
    public static void main(String args[]){
        if (args.length!=2){
            System.out.println(
                "Proper Usage: java FileDownload URL File";
            System.exit(0);
        }

        DataInputStream in=null;
        DataOutputStream out=null;
        FileOutputStream fOut=null;

        try{
            URL remoteFile=new URL(args[0]);
            URLConnection fileStream=remoteFile.openConnection();

            // Open output and input streams
            fOut=new FileOutputStream(args[1]);
            out=new DataOutputStream(fOut);
            in=new DataInputStream(remoteFile.getInputStream());

            // Save the file
            int data;
            while((data=in.read())!=-1){
                fOut.write(data);
            }
        } catch (Exception e){
            e.printStackTrace();
        } finally{
```

The Stock Quote Program

In this section we'll write the program that can read the stock market price quotes from the Internet. There are many Internet sites providing stock market price quotes. Twenty minutes delayed quotes are free which is good enough for our purposes. Wall Street companies subscribe for the real-time market data feed. One of the popular Internet sites is Yahoo and the URL for getting stock prices is http://finance.yahoo.com. Go to this site and get the price of the stock symbol you like. Look at the URL of the returned Web page in your browser. For example, if you’ve selected a symbol IBM, the URL will look like http://finance.yahoo.com/q?s=IBM. Right click on this page and select View Source from the popup menu to see the HTML contents of this page – you’ll see lots of HTML tags and the information about the IBM’s trading will be buried somewhere deep inside. Modify just one line in our class WebSiteReader to see the content of this page printed on the system console:

```java
```

You can also store the whole page in a Java String variable instead of printing the lines. Just modify the while loop:

```java
String theWholePage;
while (txt =buff.readLine() != null ){
    theWholePage=theWholePage + txt;
}
```

If you add some smart tokenizing of theWholeString, to get rid of all HTML tags and everything but Last Price info, you can create your own little GUI Stock Quote screen. While this approach is useful to sharpen you tokenizing skills, it may not be the best solution, especially if Yahoo will change the words they use on this page. That’s why we’ll be using another Yahoo's URL that provide stock quotes in a cleaner CSV (comma separated values) format.

Here’s the URL that should be used for the IBM’s symbol:
This URL would produce a string that looks something like this (obviously the price quotes are not real):


The class StockQuote prints the price quote for the symbol that has to be provided in a command line.

```java
import java.net.*;
import java.io.*;
import java.util.StringTokenizer;
public class StockQuote {
    String csvString;
    URL url = null;
    URLConnection urlConn = null;
    InputStreamReader inStream = null;
    BufferedReader buff = null;

    StockQuote(String symbol){
        try{
            url = new URL("http://quote.yahoo.com/d/quotes.csv?s=
                           ++symbol + "&f=sl1d1t1c1ohgv&e=.csv" );
            urlConn = url.openConnection();
            inStream = new InputStreamReader(urlConn.getInputStream());
            buff = new BufferedReader(inStream);
            // get the quote as a csv string
            csvString =buff.readLine();
            // parse the csv string
            StringTokenizer tokenizer = new StringTokenizer(csvString, ",");
            String ticker = tokenizer.nextToken();
            String price  = tokenizer.nextToken();
            String tradeDate = tokenizer.nextToken();
            String tradeTime = tokenizer.nextToken();

            System.out.println("Symbol: " + ticker + " Price: " + price + " Date: " + tradeDate + " Time: " + tradeTime);
            
        }
    }
}```
catch(MalformedURLException e){
    System.out.println("Please check the spelling of " +
    "the URL: " + e.toString());
} catch(IOException e1){
    System.out.println("Can't read from the Internet: " +
    e1.toString());
}
finally{
    try{
        inStream.close();
        buff.close();
    }catch(Exception e){}
}
}

public static void main(String args[]){
    if (args.length==0){
        System.out.println("Sample Usage: java StockQuote IBM");
        System.exit(0);
    }
    StockQuote sq = new StockQuote(args[0]);
}

Start the StockQuote program as follows:

c:\practice>java StockQuote IBM

Assignment. Create a Java Frame that that will allow users to enter a stock
symbol and, after clicking on the button “Get Quote” the class StockQuote retriive the quote from the Internet and display it in a text field. Make the
program a little fancier by displaying the price in green if there was a price
increase - just check for the plus sign in the quote string, for example

    if( myString.pos("+")>0){...}

Modify the class StockQuote to allow input of multiple symbols from a
command line.

Socket Programming

If two programs need to exchange data, but they are not using HTTP
protocol, one of the technologies to consider is programming sockets. In this
section we'll use Java classes Socket and ServerSocket from the package
java.net. A socket represent a connection point in the TCP/IP or UDP/IP
network. TCP protocol maintains a stream connection for the whole period of communication, while UDP is a connectionless protocol which sends data in pieces (datagrams).

When a Java program create a ServerSocket it becomes a server that just sits in memory an “listens” for requests that may come through the specified port. The following lines create a server that is listening to port 3000:

```java
ServerSocket serverSocket = new ServerSocket(3000);
client = serverSocket.accept();
```

The client program should create a client socket pointing at the server machine and port where the server is running. For example, if the name of the server machine was “Apollo”, the client should be created this way:

```java
clientSocket = new Socket("Apollo", 3000);
```

These are the alternative ways of creating client sockets:

```java
clientSocket = new Socket("124.67.98,101", 3000);
clientSocket = new Socket("localhost", 3000);
clientSocket = new Socket("127.0.0.1", 3000);
```

While deciding which port number to use, try not to use port numbers below 1024 to avoid conflicts with other popular programs. For example, port 80 is usually used by HTTP servers, port 21 is common for FTP communication, 389 for LDAP servers, etc.

After creating a socket both client and server should obtain references to its input/output streams and use them for data exchange.

### The Stock Quote Server with Sockets

Let’s create classes StockQuoteServer and Client. A server will provide the fake price quotes for IBM and MSFT.

The server starts and listens to requests on the specified port. The method accept() is the one that waits for the client’s requests. When a client connects to the socket, the server gets references to its input/output streams and receives/sends randomly generated stock quotes. In a real life this server had to be connected to another program providing real-time market data feed.

```java
import java.io.*;
import java.net.*;
public class StockQuoteServer {
public static void main(java.lang.String[] args) {
    ServerSocket serverSocket = null;
    Socket client = null;

    BufferedReader inbound = null;
    DataOutputStream outbound = null;

    try {
        // Create a server socket
        serverSocket = new ServerSocket(3000);

        System.out.println("Waiting for a quote request");
        while (true) {
            // Wait for a request
            client = serverSocket.accept();

            // Get the streams
            inbound = new BufferedReader(new
                InputStreamReader(client.getInputStream()));
            outbound = new DataOutputStream(
                client.getOutputStream());

            String symbol = inbound.readLine();

            // Generate a random price
            String price = (new Double(Math.random() * 100)).toString();
            outbound.writeBytes("The price of "+symbol+
                " is " + price + 
                "\n");

            System.out.println("Request for " + symbol +
                " has been processed ");
            outbound.writeBytes("End\n");
        }
    }
    catch (IOException ioe) {
        System.out.println("Error in Server: " + ioe);
    }
}

The client program below connects to the server's socket, get references to I/O streams, sends the stock symbol and receives the price quote.

import java.io.*;
import java.net.*;

public class Client {
    public static void main(java.lang.String[] args) {
        if (args.length==0){
            System.out.println("Usage: java Client Symbol");
            System.exit(0);
        }
        DataOutputStream outbound = null;
        BufferedReader inbound   = null;
        Socket clientSocket       = null;

        try {
            // Open a client socket connection
            clientSocket = new Socket("localhost", 3000);
            System.out.println("Client: " + clientSocket);

            // Get the streams
            outbound = new DataOutputStream(clientSocket.getOutputStream());
            inbound = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

            // Send stock symbol to the server
            outbound.writeBytes(args[0] + "\n");
            outbound.writeBytes("End\n");

            while (true){
                String quote = inbound.readLine();
                if (quote.equals("End")){
                    break;
                }
                System.out.println(quote);
            }
        } catch (UnknownHostException uhe) {
            System.out.println("UnknownHostException: " + uhe);
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe);
        }
        finally{
            // Clean up
            try{
                outbound.close();
                inbound.close();
                clientSocket.close();
            } catch (IOException e){
            }
        }
    }
}
System.out.println("Can not close streams..." + e.getMessage());
}
}
}

How to Run the Stock Quote Server

Step 1. Open 2 command windows and get into your practice directory in each of them. Compile your Java classes.

c:\practice>javac Stock*.java
c:\practice>javac Client.java

Step 2. Start the stock quote server. This is what you should see:

c:\practice>java StockQuoteServer
Waiting for a quote request...

Step 3. Start the client program in the second command window supplying the stock symbol you are interested in. This is the look of the client’s window:

c:\practice>java Client SUNW
Client: Socket[addr=localhost/127.0.0.1,port=3000,
localport=1032]
The price of SUNW is 12.741174389310073

The server’s window should look like this:

c:\practice>java StockQuoteServer
Waiting for a quote request

Request for SUNW has been processed

Step 4. Open a couple of more command windows and start client programs there specifying different stock symbols.

From now on we’ll be using different command windows to test communication between Java programs. Since we start separate Java runtime environments in each command window, this is similar to network communication. If you can test our stock server application in a real LAN,
replace the localhost in the client program with the network name or IP address of the server’s computer and run you clients and server on different machines instead of different command windows.

Introduction to JavaMail API

Most of the people are used to send or receive e-mails using some mail client programs, such as Microsoft Outlook, Lotus Notes etc. Can you write your own program to send, browse and receive e-mails? Most likely you won’t need to do it, but you may need to know how add an e-mailing capability to one of your existing programs. For example, a serious error occurred in the running system and the program that have caught this exception needs to send a message to the tech support group.

The popular e-mail protocols are POP, SMTP, IMAP and MIME. Sun Microsystems provides implementation for these protocols, i.e. pop.jar and a generic JavaMail Application Program Interface (API) that is simple and allows you to make the same 5-6 method calls to send a message with any of the supported protocols.

SMTP stands for Simple Mail Transfer Protocol and your Internet Service Provider runs the SMTP server that delivers received mail to its subscribers using you e-mail using Post Office Protocol (POP). The third version of this protocol is called POP3. More advanced alternative to POP3 would be Internet Message Access Protocol (IMAP). The format of the message body is defined by the Multipurpose Internet Mail Extensions (MIME) protocol.

At a minimum, you have to use the following classes to send and receive an e-mail:

- javax.mail.Session
  This class knows important characteristics of your environment, such as the name of the mail server and others.

- javax.mail.Message
  This class represents your message containing recipients, subject, content, etc.

- javax.mail.internet.InternetAddress
  Recepient in your message will be represented by this class.

- javax.mail.Transport
This class will pass your message for the delivery to a protocol-specific transport.

- `javax.mail.Store`
  This class is needed only if your program receives messages.

Now we'll write a program that sends e-mail messages. Since we are not going to receive messages, the pop3 server is not needed, but you need to know the name of your SMTP server. You can find it in the settings of your e-mail client program. For example, Microsoft Outlook has this information in the menu Tools | Accounts | Mail | Properties. If your program to logon to your mail server, it won't use a logon screen, but rather provide the user id and password using the subclass of the class Authenticator. When the mail server needs to do check your credentials, it'll call the method `getPasswordAuthentication()`. The `FileNotFoundException` is thrown in case of a wrong id or a password.
If the authentication is not required – the class MyMailAuthenticator is not needed.

```java
import java.net.*;

class MyMailAuthenticator extends Authenticator {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication("yakov", "secret123".toCharArray());
    }
}

import java.util.Properties;
import java.util.Date;
import java.net.Authenticator;
import java.net.PasswordAuthentication;
import javax.mail.*;
import javax.mail.internet.*;

public class MailMan {
    private static MailMan m_instance = null;
    private static Session session = null;

    private static String to = "abc@xyz.com";
    private static String cc = "mary@xyz.com";
    private static String from = "yakov@xyz.com";
    private static String subject = "Party Invitation";

    private MailMan()
```
Authenticator.setDefault(new MyMailAuthenticator());
Properties props = new Properties();
props.put("mail.smtp.host", "myMailServer.xyz.com");

// Create a Session object using properties of your // mail provider
session = Session.getDefaultInstance(props, null);

void send(String msgText)
{
try{
    // Create and initialize the message object
    Message msg = new MimeMessage(session);
    msg.setFrom(new InternetAddress(from));
    msg.setRecipients(Message.RecipientType.TO,
        InternetAddress.parse(to, false));
    msg.setRecipients(Message.RecipientType.CC,
        InternetAddress.parse(cc, false));
    msg.setSubject(subject);
    msg.setSentDate(new Date());
    msg.setText(msgText);

    // send message
    Transport.send(msg);
    System.out.println("The Message has been sent");
}catch(Exception e){
    System.out.println("Message has not been sent: " +
        e.getMessage());
}
}

public static void main(String[] args)
{
    MailMan mm = new MailMan();
    mm.send("Test Message...");
}

How to Run the MailMan Program

This program relies on classes that have to be downloaded if you are using Java 2 Standard Edition. In case of J2EE edition no additional downloads is required.
Step 1. Download JavaMail API Implementation version 1.2 from http://java.sun.com/products/javamail/index.html. The downloaded file javamail-1_2.zip contains complete documentation of JavaMail API and the following jars that implement it: mail.jar, smtp.jar, pop3.jar, imap.jar and mailapi.jar. Extract these jars from the archive file into directory Lesson8\practice. Only the first two jars are required to send an e-mail.

Step 2. Download JavaBeans Activation Framework (JAF) from http://java.sun.com/products/javabeans/glasgow/jaf.html. Extract the activation.jar from the jaf1_0_1.zip into directory Lesson8\practice.

Step 3. Modify the class MailMan.java to specify valid from/to e-mail addresses.

Step 4. Start the script sendMail.cmd that will set the environment variables, compile the classes and run the program.

    c:\Lesson8\practice>sendMail

We’ll be often using command scripts for compilation, setting environment and starting programs like the script sendMail.bat below. Let’s define a variable HOME that you might need to change based on the location of the Lesson8 directory.

    rem --- set the environment ---
    set HOME=c:\Lesson8\practice\n
    set classpath=%HOME%mail.jar;%HOME%smtp.jar;
    %HOME%activation.jar; %classpath%

    rem --- compile the classes ---
    javac MailMan.java MyMailAuthenticator.java

    rem --- run the program ---
    java MailMan

Assignment. Write a program that reads names and addresses from a file and sends each of them an e-mail inviting to the party. The file may look like this:

    Veronika, Brodskaya, vb@abc.com
    Natalia, Babich, nb@xyz.com
Use the class `StringTokenizer` to address the recipient by name, for example,

Dearest Veronica,
You are invited...

**Resources**

1. JavaMail tutorial by jGuru:  

2. The site for downloading JavaBeans Activation Framework  

3. The site for downloading JavaMail API Version 1.2  

4. Socket Programming In Java: A Tutorial  
Lesson 9

Data Structures And Collections

Arrays Revisited

Java collection classes allow storing of handles of related data in the same object. The word handles means references to memory locations of the objects. We've discussed arrays in Lesson 3 - they let you access a group of variables by the same name. Let's recall the steps you go through to declare and populate an array.

- First, declare an instance of the array variable of the type that matches the types of objects that will be stored in there. This is an example of array declaration for storing 10 instances of class Customer:

  ```java
  Customer cust[] = new Customer[10];
  ```

  Please note that at this point we've just allocated enough space for storage of 10 handles, not the actual objects.

- Second, create instances of the objects and store their handles in the array:

  ```java
  cust[0] = new Customer("David","Fain");
  cust[0] = new Customer("Ringo","Starr");
  ...
  cust[9] = new Customer("Lucy","Mann");
  ```

  Only the memory addresses of these instances are stored in the array cust.

Now, let's give a 15% discount to all customers who spent more that $500 in our video store:

```java
int totalCust = cust.length;
for (int i=0; i<totalCust; i++){
    if (cust[i].getTotalCharges() > 500){
        cust[i].setDiscount(15);
    }
}
```
If a program tries to access an array element that is beyond the arrays length (i.e. `cust[25].setDiscount(15)`), Java will throw a runtime exception `ArrayIndexOutOfBoundsException`.

### Classes Vector and ArrayList

The package `java.util` has several classes that are quite handy when multiple instances of some objects have to be co-located in memory.

The drawback of arrays is that you have to know the number of array elements in advance. The class `Vector` does not have this restriction — you can add more elements to it as needed. Since there is no such thing as “free lunch”, you have to pay the price — vectors are a little slower than arrays and you could only store objects there — primitives are only allowed as member variables of objects. To create and populate a `Vector` object, you should instantiate it, create instances of the objects and add them to the `Vector` by calling the method `add()`:

```java
Vector customers = new Vector();
Customer cust1 = new Customer("David","Fain");
customers.add(cust1);
Customer cust2 = new Customer("Ringo","Starr");
customers.add(cust2);
```

The method `add()` does not copy the instance of the object into the vector customers, it just stores its memory address. The element numbering in a `Vector` starts with 0. Vectors are slower than arrays because JVM has to allocate memory space for object handlers every time the method `add()` is called. Since arrays know how many elements to expect, memory allocation is only done once. If you have an idea about the number of elements in a vector, for example 10, instantiate it with the following constructor:

```java
Vector customers = new Vector(10);
```

You can still add more than 10 elements, but memory allocation will be done in chunks, because a `Vector` doubles its size as needed.

The method `elementAt()` could be used to extract the elements from a `Vector`. Since a `Vector` is a generic storage for any types of objects, the method `elementAt()` returns elements as an `Object` data type and it’s the responsibility of the programmer to provide proper casting, for example:
Customer theBestCustomer=(Customer) customers.elementAt(1);

To illustrate a possible runtime error, if the casting was not properly done, let’s add an object of another type to our customers collection:

Order ord = new Order(123, 500, “IBM”);
customers.add(ord);

Java compiler will not complain because Vector can store any objects. At this point, we’ve got the elements in the vector customers – 2 customers and one order. The following code will throw IllegalCastException on the third iteration of the loop:

```java
int totalElem = customers.size(); // number of elements
for (int i=0; i< totalElem;i++) {
    Customer currentCust = (Customer) customers.elementAt(i);
    currentCust.doSomething();
}
```

The operator instanceof helps avoid this exception:

```java
int totalElem = customers.size();
for (int i=0; i<totalElem;i++) {
    Object currElement = customers.elementAt(i);
    if (currElement instanceof Customer) {
        Customer currentCust = (Customer) customers.elementAt(i);
        currentCust.doSomething();
    } else if (currElement instanceof Order) {
        Order currentOrder = (Order) customers.elementAt(i);
        currentOrder.doSomething();
    }
}
```

Programs work faster if they do not need to perform the type check with the operator instanceof, but sometimes it might become quite handy. You’ll find more samples of working with vectors in the Swing and JDBC lessons.

The class ArrayList is very similar to the class Vector, as a matter of fact, they both internally use Array as a storage.

```java
ArrayList customers = new ArrayList(5);
customers.add(new Customer(“David”, “Fain”));
customers.add(new Customer(“Ringo”, “Starr”));
```
You can convert data from an `Vector` or `ArrayList` to an `Array` by using the method `toArray()`. The following example converts the `ArrayList` `customers` into an array:

```java
int totalElem=customers.size();
Customer[] custArray =
  (Customer[]) customers.toArray(new Customer[totalElem]);
```

`ArrayList` grows at a slower rate (50% increase as opposed to doubling in a `Vector`). Another important difference is that `Vector` is internally synchronized, while `ArrayList` is not (read the section “My Brokerage Firm With Threads” in the Lesson on multithreading).

### Classes `Hashtable` and `HashMap`

The classes `Hashtable` and `HashMap` offer yet another way of storing and accessing the element of a collection – by a key. You can assign a key to an instance of the object and use it as a reference. These classes deal with key/value pairs. Let's say we need to store instances of classes `Customer`, `Order`, and `Portfolio` in the same collection. In the following example, we'll create these instances first, and then put them in the collection under some names (keys):

```java
Customer cust = new Customer("David","Fain");
Order ord = new Order(123, 500, "IBM");
Portfolio port = new Portfolio(123);

Hashtable data = new hashtable();
data.put("Customer", cust);
data.put("Order", ord);
data.put("Portfolio", port);
```

The values in double quotes represent keys by which the objects could be retrieved. In this example, the keys are represented by a Java class `String`, but you can use any objects for this (no primitives are allowed).

If you can have an idea of how many elements you are planning to have in the `Hashtable`, use another constructor:

```java
Hashtable data = new Hashtable(10); // 10-element capacity
```
The method `get()` provides access to these objects by the key (do not forget about the proper casting):

```java
Order myOrder = (Order) data.get("Order");
```

The method `size()` returns the number of elements in the `Hashtable`

```java
int totalElem = data.size();
```

Methods `containsKey()` and `containsValue()` help you find out if the collection contains a specific key or value.

You can find the sample of `Hashtable` usage in the code of the class `MyBrokerageFirm` in the Lesson on multithreading.

The class `HashMap` is similar to `Hashtable`, but it allows nulls as key or values and is not synchronized (see the Lesson 10).

### Interface `Enumeration`

If a collection object implements the interface `Enumeration`, you can work with its elements in another fashion – sequentially, without even knowing the total number of elements. You just need to obtain the enumeration of all elements and use two methods: `hasMoreElements()` and `nextElement()`.

For example, to process the elements of the `Vector customers` do the following:

```java
Vector customers = new Vector();
...
Enumeration enum = customers.elements(); // returns Enumeration
while(enum.hasMoreElements()){
    Customer currentCust = (Customer)enum.nextElement();
    currentCust.doSomething();
}
```

You can also obtain and process the enumeration of `Hashtable`'s keys or elements. For example,

```java
Hashtable data = new Hashtable();
...
Enumeration enumKeys = data.keys();
while(enum.hasMoreElements()){
```
It has always been a good idea to minimize the number of hard-coded values. Windows-based applications often store some configurable parameters in the .ini files. These parameters are usually stored as key/value pairs. I am pretty sure that you could find plenty of files with the name extension .ini on your Windows PC.

These days, Java applications store their properties either in the .properties files or in the XML files. This section explains how the Java class Properties could be used to store key/value pairs and how easily it could read Java .properties files.

For example, the program MailMan from Lesson 8 could have read the e-mail server names and from/to values from a .properties file instead of using hard-coded values. For example, you could create a text file mailman.properties with the following contents:

```
SmtpServer=mail.xyz.com
to=abc@xyz.com
cc=mary@xyz.com
from=yakov@xyz.com
```

The class Properties is a subclass of the Hashtable and has the following restrictions - both the key and the value have to be from type String.

To load this file into the Properties object, just define an input stream on this file and call the method load(). After the file has been loaded into the Properties object, each individual property could be obtained using the method getProperty():

```
Properties prop=new Properties();
FileInputStream in =null;
try{
    new FileInputStream ("mailman.properties");
    prop.load(in);
}catch(Exception e){...}
finally{... in.close();...}
```
String from = prop.getProperty("from")
String mailServer=prop.getProperty("SmtpServer ");
...

Java does not have global variables, but as a workaround you can make these properties available to any object of your application by turning them into system properties:

System.setProperties(prop);

Now you can get these values from any other class of your application, for example:

String mailServer = System.getProperty("SmtpServer ");

**Class BitSet**

Suppose you need to write a program that sends a signal with information about some device. For example, some vending machines have smart chips that could automatically dial a phone number and send a signal with information about its coin collector, one of 10 possible failures, etc. A set of flags (bits set to 1 or 0) is the most economical way to do it. For example, a Java primitive long is a 64-bit (8 bytes) number. The BitSet does not have a 64-bit limitation and can grow as needed. Depending on which bit is set (has the value of 1) it could mean the following:

Bit 0 - the coin box is empty
Bit 1 - the coin box is half full
Bit 2 - the coin box is full
Bit 3 - the coin box is removed
Bit 4 - The Coca Cola row is empty
...

If a vending machine will just send one BitSet, it could contain multiple parameters describing its status. The program that receives this signal could print a nice report and the owner of this remote machine could decide if he needs to send a technician there.

Java class BitSet is nothing more than a vector of bits. The code below prepares a signal saying that the coin box is full and there is no Coca Cola bottles left.
import java.util.BitSet;
class VendingMachineSender {
    public static void main(String args[]) {
        BitSet report = new BitSet();
        report.set(2);   // box is full
        report.set(4);   // no Coca Cola
    }
}

Let's say when the phone call comes in, the callback phoneRings() is called and the signal could be decoded like this:

import java.util.BitSet;
class VendingMachineListener {
    public void phoneRings(BitSet signal) {
        int size = signal.size();
        for (int i=0; i<size; i++) {
            if (signal.get(i)) {
                switch (i) {
                case 0: System.out.println("Box is empty"); break;
                case 1: System.out.println("Box is half full"); break;
                case 2: System.out.println("Box is full"); break;
                ...
            }
        }
    }
}

The class BitSet also has methods for various bit manipulations such as logical OR, AND, XOR, and others.

Class LinkedList

If you need to work with a sequential list of objects and often insert the object in the beginning and the end of the list, the class LinkedList could fit the bill. These features of the LinkedList allow you to create queues - first-in-first-out (FIFO), and stacks - first-in-last-out (LIFO).
Elements of the list are smart enough to know if there is an element before and after the current one (hence the word **linked**). You can navigate through the list using the class `ListIterator` and process the list using such methods as `next()` and `previous()`.

**Resources**

1. The Java tutorial from Sun Microsystems. Collection trail:  

2. The Collection Framework  

3. Getting Started With Java Collections by Dan Becker  

3. Collections FAQ from jGuru.com  
   [http://www.jguru.com/faq/Collections](http://www.jguru.com/faq/Collections)
Lesson 10
Multithreading In Java

Introduction

Let’s discuss and write a program that will display market news and stock portfolio data of a customer in the same window. Market news are coming from a remote computer and stock portfolio data are retrieved from the database located on the local machine. Assume that it takes 5 seconds to get the market news and 3 seconds to get the portfolio data. If these two tasks are performed sequentially (one after another), you’d need 8 seconds to complete the job.

But market news do not directly depend on your portfolio data and these two tasks can run in parallel. In a real-world applications these tasks run on different computers, and hence use different processors. In case of parallel processing, the total time should be close to 5 seconds (the time needed for the longer task).

A Java program can start multiple threads (just two in our case) that are running in parallel.

Even if you have only one processor in your computer, it still could be a good idea to parallel some tasks. Think of a web browser that allows you to perform download of a file and page browsing at the same time. The browser is just one program that works in a multithreaded mode. If these two jobs would have run sequentially, the browser’s screen would have been frozen till the download is complete. In case of one processor each thread gets a slice of the processor’s time. Since it happens pretty fast, a user can’t notice small delays and has a feeling that she browses the web page smoothly.

People also can work in a multi-threaded mode, for example drinking coffee while talking on the cell phone and driving a car.

Some other programming languages (i.e. C++) also allow multithreading, but thread programming is much easier in Java.

A thread is a sort of a lightweight process within another process. Think of people running on a moving escalator steps – they will reach the destination faster than if they’d be just standing on the steps.
Class Thread

To become a thread Java class has to either be inherited from a class Thread or it has to implement the Runnable interface. In both cases the processing must be initiated from the method run(). If a class is inherited from the class Thread it has to override the method run(). The first version of our market-portfolio example has 3 classes – two of them are threads which are subclasses of the class Thread (MarketNews and Portfolio) and the third one (TestThreads) is just a testing program that creates and starts them.

```java
public class MarketNews extends Thread {
    public MarketNews (String str) {
        super(str); // str is an arbitrary thread name
    }
    public void run() {
        System.out.println( "The stock market is " +
                            "improving every day! ");
    }
}

public class Portfolio extends Thread {
    public Portfolio (String str) {
        super(str);
    }
    public void run() {
        System.out.println( "You have 500 shares of IBM ");
    }
}

public class TestThreads {
    public static void main(String args[]) {
        MarketNews mn = new MarketNews("Market News");
        mn.start();
        Portfolio p = new Portfolio("Portfolio data");
        p.start();
        System.out.println( "TestThreads is finished");
    }
}
```

Even though the method main() calls the thread's method start(), it invokes the code located in the thread's method run(). After calling the mn.start() the program does not wait and immediately executes the lines below (creates and starts the thread Portfolio).
Interface Runnable

The second way of creating threads is by implementing a Runnable interface. In this case your class also has to have business logic in the method run(). The second version of our market-portfolio example has also 3 classes, but MarketData2 and Portfolio2 are not inherited from the class thread. Creation of a thread in this case is a two-step process: an instance of a class that implements Runnable is created first and then it is being used during instatiation of a class Thread.

```java
public class MarketNews2 implements Runnable {
    public void run() {
        System.out.println( "The stock market is " +
                            "improving every day!");
    }
}

public class Portfolio2 implements Runnable {
    public void run() {
        System.out.println( "You have 500 shares of IBM ");
    }
}

public class TestThreads2 {
    public static void main(String args[]) {
        MarketNews2 mnClass = new MarketNews2();
        Thread mn = new Thread(mnClass, "Market News");
        mn.start();

        Portfolio2 portClass = new Portfolio2();
        Thread p = new Thread(portClass, "Portfolio Data");
        p.start();
        System.out.println( "TestThreads2 is finished");
    }
}
```

The Runnable interface provides more flexible way of using threads, because it allows a class to be inherited from any other class, while having all features of a thread. For example, an applet must be a subclass of a java.applet.Applet, that's why it should implement Runnable to have multi-threading features.
Thread States

Any thread goes through various states during its life time. Here they are:

- **New.** A thread is in this state when its instance was created but the method `start()` has not been called yet.

- **Runnable.** A thread enters this state after the method `start()` has been called. It could yield the processor's time to another thread with higher priority for some time while running, but it's still remains in a runnable state.

- **Not Runnable.** The reasons that could place a thread into this state are:
  
a) The method `sleep()` has been called;

b) The method `wait()` has been called;

c) The thread is processing some input/output (it's blocking on I/O).

- **Dead.** The threads are very much like people – they could either die naturally, when its `run()` method is finished, or violently, when some outside process caused its death. There are such thread killers as deprecated method `stop()`. It was deprecated because it could not guaranteed that the thread would be killed. Another example of stopping a thread is when I/O error occurred while reading or a stream was suddenly closed.

A class `Thread` has a method `isAlive()` that can help you to find out the status of a thread. If it returns true, the thread is either in a Runnable or in a Not Runnable state. If it returns false, the thread is either New or Dead.

Sleeping Threads

One of the ways to yield the processor to another thread is by using the method `sleep()`. It takes one parameter specifying how long the thread has to sleep – it's time in milliseconds. Here is an example:

```java
class MarketNews3 extends Thread {   
  public MarketNews3 (String str) {    
    super(str);    
} ```
```java
public void run() {
    try{
        for (int i=0; i<10;i++){
            sleep (1000); // sleep for 1 second
            System.out.println( "The market is improving " + i);
        }
    }catch(InterruptedException e ){
        System.out.println(Thread.currentThread().getName()
                        + e.toString());
    }
}

public class Portfolio3 extends Thread {
    public Portfolio3 (String str) {
        super(str);
    }
    public void run() {
        try{
            for (int i=0; i<10;i++){
                sleep (700); // Sleep for 700 milliseconds
                System.out.println( "You have " + (500 + i) +
                        " shares of IBM");
            }
        }catch(InterruptedException e ){
            System.out.println(Thread.currentThread().getName()
                        + e.toString());
        }
    }
}
```

After adding the sleeping part to our thread, the program TestThreads3 will generate mixed console output from both threads, that proves that they are taking turns even with the single processor machine.

Let's do one more test: add the line `mn.interrupt()` to the class TestThreads right after starting the MarketNews. This will trigger the `InterruptedException` and MarketNews will “wake up”.

If you had a multi-processor computer, each of these threads would be utilizing its own processor. Real-time trading systems often run on Unix boxes with 4, 8 and even more processors. By increasing the number of running threads you can boost the performance of your system. The right ratio between the number of threads and processors should be defined during performance tuning phase.
Thread Priorities

Single processor computers utilize a special scheduling algorithm that allocates processor time slices to the running threads based on their priorities. If a Thread1 is using the processor and a higher priority Thread2 wakes up, a Thread1 is put aside and a Thread2 gets the processor. It is said that the Thread2 preempts the Thread1.

The class Thread has a method setPriority() that allows you to control its priority. Priorities are final integer variables defined in the class Thread and have the following names: MIN_PRIORITY, NORMAL_PRIORITY, and MAX_PRIORITY. Here's an example of their usage:

```java
Thread myThread = new Thread(“Portfolio”);
myThread.setPriority(Thread.NORMAL_PRIORITY + 1);
```

If two threads with the same priority need the processor, it'll be given to one of them using an algorithm specific to computer's OS.

Thread Synchronization. Race Condition

During design stage of a multithreaded application you should consider possibility of so called race condition which happens when multiple threads need to modify some resource (a class variable) at the same time. The classical example is when a husband and wife are trying to withdraw cash from different Automatic Teller Machines at the same time. If a thread class is responsible for the validation and update of the balance in their account record, the chances are that the husband's thread gets OK to withdraw $100 (the balance on the account was $120) and right after the husband's thread has validated the transaction but before the actual withdrawal, wife's thread comes in trying to validate $50 withdrawal. She also gets OK, because $120 is still there! They will successfully withdraw a total of $150 leaving the bank with a negative balance in their account.

To prevent race conditions Java has a special keyword synchronized that allows to place a lock (a monitor) on important object or a piece of code to make sure that only one thread will have an access to this piece of code. The following example would lock the whole method withdrawCash().

```java
class ATMProcessor extends Thread{
    ...
```
synchronized withdrawCash(int accountID, int amount) {
    boolean allowTransaction = validateWithdrawal(accountID, amount);
    if (allowTransaction) {
        updateBalance(accountID, amount, "Withraw");
    } else {
        System.out.println("Not enough money on the account");
    }
}

The locks should be placed for a short time to avoid slowing down of the program, that’s why synchronizing code blocks is preferable to synchronizing the whole methods:

withdrawCash(int accountID, int amount) {
    // Some thread-safe code goes here, i.e. reading from
    // a file or a database
    ...
    synchronized (this) {
        if (allowTransaction) {
            updateBalance(accountID, amount, "Withraw");
        } else {
            System.out.println("Not enough money on the account");
        }
    }
}

When a synchronized block is executed, the object in parenthesis is locked and cannot be used by any other thread until the lock is released.

Wait and Notify

The class Object also has some relevant to threads methods: wait(), notify() and notifyAll(). Since every Java class is inherited from the class Object, these methods could be called on any object.

Let’s revisit our class TestThreads3 that spawns the threads MarketNews and Portfolio. It has the following line at the end of the main() method:

System.out.println("TestThreads is finished");

The program prints on the console something like this:
The stock market is improving every day! 1
You have 500 shares of IBM

The main method of TestThreads is finished

The stock market is improving every day! 2
You have 501 shares of IBM

The stock market is improving every day! 3
You have 502 shares of IBM

... 

The method main() did not wait for the threads' completion! What if a class needs to wait for some results produced by the threads? The method wait() will help you with this:

```java
public static void main(String args[]){
    ...
    mn.start();
    ...
    p.start();
    synchronized (this) {
        try{
            wait(10000);
        } catch (InterruptedException e){  ...}
    }
    System.out.println("The main method of TestThreads is finished");
}
```

The method call wait(10000) means “wait up to 10 seconds”. The last print statement will be executed either after 10 seconds, or if this thread will receive a notification that some important event – whichever comes first. A thread could notify another one(s) by calling the method notify() or notifyAll().

The sleep(10000) will put a thread into a Not Runnable state for exactly 10 seconds, while the wait(10000) may bring it back to a Runnable state earlier.

If the method wait() is called without any arguments, the calling program will wait for indefinite time until it receives notification. The call wait() makes a current thread to give up its lock to another one and puts it to sleep while notification comes from another class. Here’s one of the use cases:

A ClassA spawns a thread ClassB and start waiting. The ClassB retrieves some data, and when it’s done, sends notification back to the ClassA. The ClassA resumes its processing after notification has been received. Below is a sample code to illustrate this scenario:
class ClassA {
    String marketNews = null;

    void someMethod() {
        // The ClassB needs a reference to the locked object
        // to be able to notify it
        ClassB myB = new ClassB(this);
        myB.start();
        synchronized(this) {
            wait();
        }
        // Some further processing of the MarketData goes here...
    }

    public void setData (String news) {
        marketNews = news;
    }
}

class ClassB extends Thread {
    ClassA parent = null;

    ClassB(ClassA caller) {
        parent = caller; // store the reference to the caller
    }

    run() {
        // Get some data, and, when done, notify the parent
        parent.setData("Economy is recovering...");
        ...
        synchronized (parent) {
            parent.notify(); // notification of the caller
        }
    }
}

The method `notifyAll()` will notifies and wakes up all waiting threads. If more than one thread is waiting for the lock to be released, Java runtime environment will pick one of them, and it's not guaranteed which one is going to be picked.
Deprecated Methods

The following methods of the class Thread were deprecated: stop(), suspend() and resume(). These methods are still available but are not recommended for use and may not be supported in the future releases.

The method stop() unlocks the monitors and could leave the object in an inconsistent state. You better try to find some other solution for killing unwanted threads. For example, if a file reading thread runs way too long, just close the IO stream – this will generate an exception and the thread will die.

Methods suspend() and resume() could lead to deadlocks since it could be difficult to control the status of the locks when the execution of a thread is being resumed.

My Brokerage Firm With Threads

The web site of your bank or brokerage firm contains a lot of pieces of information – Market News, Stock Quotes, Portfolio, Graphs, Financial Forecast, etc. All these pieces are usually provided by different processes that may run on different computers, but the screen comes up as a whole piece in one shot. This could be achieved by using the following design pattern: a main program spawns threads, collects the data returned by them and displays the result after all threads are finished.

In this section we are going to put all pieces together and create a program by the following specification:

1. Create a Frame window MyBrokerageFirm with two AWT TextField components – txtStockSymbol, txtStockPrice, a TextArea txtPortfolio, and a Button “Go”.
2. Create two subclasses of Thread: PortfolioFile and Quote.
3. When the button is pressed, spawn the threads PortfolioFile and Quote. The thread PortfolioFile has to read a text file Portfolio.txt and display its content in the field txtPortfolio. The thread Quote has to connect to the Internet, get the stock price quote for the entered symbol and display the price in the field txtStockPrice.
4. The screen must be updated simultaneously when portfolio data and price quote are available.
The code of the frame `MyBrokersFirm` is shown below. Enter the stock symbol and click on the button Go to spawn the threads `PortfolioFile` and `Quote`. The screen is populated by the method `setData()`. This method will be called by the both threads, it checks the size of the `Hashtable` and displays the result only when both threads have returned the values.

```java
import java.awt.event.*;
import java.awt.*;
import java.util.Hashtable;

public class MyBrokersFirm extends java.awt.Frame implements ActionListener {
    Label lblSymbol = new Label("Enter Symbol");
    TextField txtStockSymbol = new TextField(5);
    Label lblPrice = new Label("LastPrice: ");
    TextField txtStockPrice = new TextField(10);
    Label lblPortfolio = new Label("Your Portfolio: ");
    TextArea txtPortfolio = new TextArea(20,40);
    Button bGo = new Button("Go");

    private Hashtable data = new Hashtable();

    MyBrokersFirm() {
        GridLayout gr = new GridLayout(4,2,1,1);
        setLayout(gr);

        add(lblSymbol);
        add(txtStockSymbol);
        add(lblPrice);
        add(txtStockPrice);
        add(lblPortfolio);
        add(txtPortfolio);
        add(bGo);

        bGo.addActionListener(this);
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.out.println("Good bye!");
                System.exit(0);
            }
        });
    }

    public void actionPerformed(ActionEvent evt) {
        try{
            // spawn 2 threads
```
PortfolioFile p = new PortfolioFile(  "Thread Portfolio",this);
p.start();
Quote q = new Quote("Thread Quote",  
txtStockSymbol.getText(),this);
q.start();
}
catch (Exception e){
  e.printStackTrace();
}
}
// Collect the data returned by the threads in the  
// Hashtable data. Only after both threads are finished  
// display the results.
public void setData(String threadName,  
    String threadData){
  data.put(threadName, threadData);
  if (data.size()==2){
    // display the results
    txtPortfolio.append((String)data.get("Portfolio"));
    txtPortfolio.setCaretPosition(0); //cursor on top
    txtStockPrice.setText((String)data.get("Quote"));
  }
}
public static void main(String args[]){
  MyBrokerageFirm firm = new MyBrokerageFirm();
  firm.setSize(400,200);
  firm.setVisible(true);
}

The class Quote is a slightly modified version of the class StockQuote from the Lesson 8.

import java.net.*;
import java.io.*;
import java.util.StringTokenizer;

public class Quote extends Thread{
  String csvString;
  URL url = null;
  URLConnection urlConn = null;
  InputStreamReader inStream = null;
  BufferedReader buff = null;
  String symbol;
  MyBrokerageFirm parent;
Quote(String threadName, String symbol, MyBrokerageFirm parent){
    super(threadName);
    this.symbol=symbol;
    this.parent=parent;
}

public void run(){
    String price;
    try{
        url = new URL("http://quote.yahoo.com/d/quotes.csv?s=
            + symbol + ";f=sl1d1c1ghv&v=.csv ");
        urlConn = url.openConnection();
        inStream = new InputStreamReader(urlConn.getInputStream());
        buff = new BufferedReader(inStream);

        // get the quote as a csv string
        csvString =buff.readLine();

        // parse the csv string
        StringTokenizer tokenizer = new
            StringTokenizer(csvString, ",");
        String ticker = tokenizer.nextToken();
        price = tokenizer.nextToken();
        String tradeDate = tokenizer.nextToken();
        String tradeTime = tokenizer.nextToken();

        System.out.println("Symbol: " + ticker +
            " Price: " + price + " Date: " + tradeDate + "
            Time: " + tradeTime);
    } catch(MalformedURLException e){
        price = "Bad URL";
        System.out.println("Check the spelling of the URL:
            " + e.toString());
    } catch(IOException e1){
        price = "Internet Problem";
        System.out.println("Can't read from the Internet:
            " + e1.toString());
    }
    finally{
        try{
            inStream.close();
            buff.close();
        }catch(Exception e){}
    }
    parent.setData("Quote", price);
}
The thread PortfolioFile reads the file Portfolio.txt. I put this thread to sleep for 5 seconds to prove that even though the Quote thread has been finished earlier, the screen is populated simultaneously.

```java
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class PortfolioFile extends Thread {
    String portfolio="";
    MyBrokerageFirm parent;
    public PortfolioFile (String str,
        MyBrokerageFirm parent) {
        super(str);
        this.parent=parent;
    }
    public void run() {
        FileReader myFile = null;
        BufferedReader buff = null;
        try { 
            myFile = new FileReader("Portfolio.txt");
            buff = new BufferedReader(myFile);
            boolean eof = false;
            while (!eof) { 
                String line = buff.readLine();
                if (line == null) 
                    eof = true;
                else 
                    portfolio=portfolio + line + "\n";
            }
        }catch (Exception e){
            portfolio="Can't read Portfolio.txt";
        }
    finally{
        // Closing the streams
        try{
            buff.close();
            myFile.close();
        }catch(IOException e){
            e.printStackTrace();
        }
    }
    // Let's imitate a slow process and pause
    // this thread for 5 seconds
    try{
        sleep(5000);
    }
}
```
catch(InterruptedException e){}

// return the result
parent.setData("Portfolio", portfolio);
}

Compile the classes and run this program. Your computer has to be connected to the Internet to receive stock quotes.

c:\Lesson10\practctice> javac *.java
c:\Lesson10\practctice> java MyBrokerageFirm

Resources

1. Threads: Doing Two or More Tasks At Once
   http://java.sun.com/docs/books/tutorial/essential/threads

2. Tech Tips, Why Use Threads
Lesson 11
Working With Databases
Using JDBC

Introduction

Business applications usually store data in the databases. So called relational Database Management Systems (DBMS) are the most popular ones. They store the data tables and understand the SQL language, which is out of the scope of this book. The major commercial relational DBMS are Oracle, DB2, Sybase, and SQL Server. All samples from this and further lessons will use the Oracle database server. An evaluation copy of which could be downloaded from www.oracle.com.

Java provides two ways of working with relational data: JDBC and SQLJ. At the time of this writing, JDBC drivers of version 2.0 are used for commercial applications, that's why we'll discuss their features. Java 2 Standard Edition includes the package java.sql. Refer to the driver's documentation to see which version of the JDBC specification is supported.

JDBC Driver Types

The JDBC driver plays the role of the middleman between a Java program and DBMS. Drivers are freely available from the database vendors' Web sites, from Sun Microsystems, and from the third-party vendors of the application servers. The vendors may also provide their drivers in the javax.sql package. There are four general types of JDBC drivers and their brief characteristics are listed below.

A type 1 driver is a JDBC-ODBC bridge that allows Java programs to work with the database using so called ODBC drivers. If you have, say a ODBC driver for Oracle installed on the user's computer, no additional Java classes are required. The drawbacks of ODBC drivers are that they are slower than the others, must be installed and configured (via the Control Panel) on each user's machine, and can only work in the Microsoft Windows machines.

A type 2 driver is Java classes that work in conjunction with the non-Java native drivers provided by the database vendors and installed on the client’s
machines. These drivers work much faster but also require installation and configuration on the machine where the Java programs run.

A type 3 driver is a Net-Protocol driver, which is provided by some application servers and consists of two parts – client’s portion performs DBMS-independent SQL call, which is then translated to a specific DBMS protocol by the server’s portion of the driver.

A type 4 driver is a pure Java driver, which comes as a .jar or a .zip file full of Java classes that perform direct calls to the database server. It does not need any configuration on the client’s machine – just the name of the main driver’s class is needed. That’s why it’s known as a “thin” driver. The applets could be packaged with this type of driver (remember the HTML tag <applet> and its attribute archive), which provides automatic downloads to the user’s machine memory.

To simplify installation, we’ll be using JDBC drivers of type 4 in this lesson, but a majority of the production systems use the type 2 drivers.

## Installing Oracle Database Server

First, download and install an evaluation copy of the Oracle database server version 8i or 9i. During installation, you’ll be prompted to enter the database name - write it down because you’ll need to specify it in the Java program. To confirm that Oracle is properly installed, start the utility SQLPlus from the Oracle | Application Development menu. Logon to the Oracle database server using the id scott and the password tiger (leave the Host String box empty because your server is installed locally). If you get TNS:protocol adapter error, make sure that OracleService and OracleTNSListener are running (find the Services icon under Control Panel menu). In case of a successful installation, you should see the following SQLPlus prompt:

```
SQL>
```

Run a simple SQL Select statement to see a list of employees:

```
SQL> Select * from EMP
```

Check the directory jdbc\lib under your Oracle directory – zip files with JDBC drivers are there, just add them to the CLASSPATH variable on your machine, for example:
CLASSPATH=c:\Oracle8i\jdbc\lib\classes12.zip;...

That’s all you need to make JDBC drivers of type 4 available to your Java programs.

Sample JDBC Program

In this section, we’ll go over several simple steps that have to be done in any Java program that work with a relational database using JDBC. Let’s first create the laundry list, and then implement it in a sample program that displays the list of employees from the Oracle table EMP.

Step 1. Load the JDBC driver using the method Class.forName(). You have to find out the name of the class to load from the database vendor’s documentation.

Step 2. Obtain the database connection by calling DriverManager.getConnection(). Since obtaining the connection to the database is a slow process, we usually use database connection pools and JDBC DataSources, which are explained in Lesson 19.

Step 3. Create a Statement object by calling Connection.createStatement(). As an alternative, you could create a PreparedStatement or a CallableStatement explained later in this lesson.

Step 4. For SQL Select perform Statement.executeQuery(). For SQL Insert, Update or Delete the call method Statement.executeUpdate(). For SQL queries, which produce more than one result, set the use method execute().

Step 5. Write a loop to process the database result set, if any:

```java
while (ResultSet.next()) {...}
```

Step 6. Free system resources by closing the ResultSet, Statement, and Connection objects.

All of the above steps are implemented in the class EmployeeList, which displays Employees from the table EMP using JDBC drivers of type 4.
import java.sql.*;

class EmployeeList {

    public static void main(String argv[]) {
        Connection conn=null;
        Statement stmt=null;
        ResultSet rs=null;

        try {
            // Load the JDBC driver class
            // (has to be in the CLASSPATH)
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // Connect to the database server by specifying
            // the driver type 4 - jdbc:oracle:thin,
            // id, password, hostname (the name of your computer),
            // port (1521 is default for Oracle) and database name
            conn = DriverManager.getConnection("jdbc:oracle:thin:scott/tiger@yakovs:1521:ORA0999");

            // Build an SQL String
            String sqlQuery = ""

            // Create a Statement object
            stmt = conn.createStatement();

            // Execute SQL and get obtain the ResultSet object
            rs = stmt.executeQuery(sqlQuery);

            // Process the result set - print Employees
            while (rs.next()){
                int empNo = rs.getInt("EMPNO");
                String eName = rs.getString("ENAME");
                String job = rs.getString("JOB");
                String hireDate = rs.getString("HIREDATE");
                System.out.println("" + empNo + ", " + eName
                        + ", " + job + ", " + hireDate);
            }
        }
    
        } catch( SQLException se ) {
            System.out.println("SQLException: "+ se.getMessage()
                        + " code: " + se.getErrorCode());
        } catch( Exception e ) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        } finally{
            // clean up system resources
            try{
                rs.close();
            }
stmt.close();
conn.close();
} catch(Exception e){
e.printStackTrace();
}
}
}

The output of the program EmployeeList should look like this:

C:\Lesson11\Practice>javac EmployeeList.java
C:\Lesson11\Practice>java EmployeeList

7369, SMITH, CLERK, 1980-12-17 00:00:00.0
7499, ALLEN, SALESMAN, 1981-02-20 00:00:00.0
7521, WARD, SALESMAN, 1981-02-22 00:00:00.0
7566, JONES, MANAGER, 1981-04-02 00:00:00.0
7654, MARTIN, SALESMAN, 1981-09-28 00:00:00.0

Processing Result Sets

After execution of the line \texttt{rs = stmt.executeQuery(sqlQuery)}, the cursor \texttt{rs} points at the very first row of the result set in memory. Each row contains as many columns as were specified in the SQL \texttt{Select} statement. Each of the values are extracted depending on the data type by calling such methods as \texttt{rs.getString()}, \texttt{rs.getInt()}, etc. If you know the names of columns in the result set, specify them as a method arguments:

\begin{verbatim}
int empNo = rs.getInt("EMPNO");
String eName = rs.getString("ENAME");
\end{verbatim}

JDBC drivers are smart enough to convert the data from the database specific types to the corresponding Java types, i.e. \texttt{varchar} and \texttt{String}. You could have gotten the same value by specifying the relative position of the column from the result set:

\begin{verbatim}
int empNo = rs.getInt(1);
String eName = rs.getString(2);
\end{verbatim}

Columns are numbered from the left starting with 1. In some cases, this is the only choice you have, for example, the following SQL query does not have a column name:

\begin{verbatim}
stmt.executeQuery("Select count(*) from EMP");
\end{verbatim}
The class `EmployeeList` just prints the retrieved data in a loop. The result set could also be placed in a Java collection object for further processing. The `ResultSet` object holds the database connection and is not serializable. That’s why the common practice is to create a class representing a row from the result set and populate a `Vector` or other Java collection object with its instances:

```java
class Employee{
    private int empNo;
    private String eName;
    private String job;
    private String hireDate;

    public void setEmpNo(int val){empNo=val;}
    public void setEName(String val){eName=val;}
    public void setJob(String val){job=val;}
    public void setHireDate(String val){hireDate=val;}

    public int getEmpNo(){return empNo;}
    public String getEName(){return eName;}
    public String getJob(){return job;}
    public String getHireDate(){return hireDate;}
}

class EmployeeList {
    ... 
    Vector employees = new Vector(); // employee collection
    while (rs.next()){
        Employee currentEmp = new Employee();
        currentEmp.setEmpNo(rs.getInt("EMPNO"));
        currentEmp.setEName(rs.getString("ENAME"));
        currentEmp.setJob(rs.getString("JOB"));
        currentEmp.setHireDate(rs.getString("HIREDATE"));

        employees.add(currEmp);
    }
    ... 
}
```

Class `ResultSetMetaData`

JDBC allows you to process result sets when the number of returned values is unknown. Imagine that you need to write a program that will accept any SQL `Select` statement and display the retrieved data. The class
ResultSetMetaData allows you to dynamically find out how many columns there are in the result set, what are their types, and names.

```java
String sqlQuery = "select * from EMP";
ResultSet rs = stmt.executeQuery(query);

ResultSetMetaData rsMeta = rs.getMetaData();
int colCount = rsMeta.getColumnCount();

for (int i = 1; i <= colCount; i++) {
    System.out.println(
        " Column name: " + rsMeta.getColumnName(i) +
        " Column type: " + rsMeta.getColumnTypeName(i));
}
```

Consider the class `ShowAnyData` that prints a result set based on the SQL Select statement passed from the command line, for example

c:\lesson11\practice\>java ShowAnyData "Select * from EMP"

```java
import java.sql.*;

class ShowAnyData {

    public static void main(String args[]) {
        Connection conn=null;
        Statement stmt=null;
        ResultSet rs=null;

        if (args.length==0){
            System.out.println("Usage: java ShowAnyData SQLSelectStatement");
            System.out.println("For example: java ShowAnyData \"Select * from EMP\"");
            System.exit(1);
        }

        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            conn = DriverManager.getConnection("jdbc:oracle:thin:scott\tiger@yakovs:1521:ORA0999");

            stmt = conn.createStatement();

            rs = stmt.executeQuery(args[0]);
        }
```
// Find out the column number, their names, 
// and display the data
ResultSetMetaData rsMeta = rs.getMetaData();
int colCount = rsMeta.getColumnCount();

for (int i = 1; i <= colCount; i++) {
    System.out.print(rsMeta.getColumnName(i) + " ");
}
System.out.println();

while (rs.next()){
    for (int i = 1; i <= colCount; i++) {
        System.out.print(rs.getString(i) + " ");
    }
    System.out.println();
}

} catch( SQLException se ) {
    System.out.println ("SQLException: " + se.getMessage ()
    + " code: " + se.getErrorCode ());
} catch( Exception e ) {
    System.out.println(e.getMessage());
    e.printStackTrace();
} finally{
    try{
        rs.close();
        stmt.close();
        conn.close();
    } catch(Exception e){
        e.printStackTrace();
    }
}
}  

Scrollable Result Sets

So far we've been navigating the result set using the method next(), which allowed us to move only in one direction - forward. Another option is to create a scrollable result set, so the cursor could be moved back and forth. There is a two-arguments version of the method createStatement(). The first argument specifies the type of scrolling (TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE), and the second one makes the result set updateable or read only (CONCUR_READ_ONLY or CONCUR_UPDATABLE), for example:
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * from EMP ");

The TYPE_FORWARDONLY only allows the forward cursor's movement. The difference between the TYPE_SCROLL_INSENSITIVE and the TYPE_SCROLL_SENSITIVE is if the scrolling will reflect the changes, which might have been done to the result set. The next example sets the cursor at the end of the result set and moves the cursor backword:

rs.afterLast();
while (rs.previous()){
    int empNo = rs.getInt("EMPNO");
    String eName = rs.getString("ENAME");
    String job = rs.getString("JOB");
    String hireDate = rs.getString("HIREDATE");
    System.out.println("" + empNo + ", " + eName
                 + ", " + job + ", " + hireDate);
}

You can also move the cursor to a specific row by using the following self-explanatory methods:

rs.absolute(25);  // moves the cursor to the 25th row
rs.relative(-4);  // moves the cursor to the 21st row
rs.first();
rs.last();
rs.beforeFirst();

If the result set is updatable (CONCUR_UPDATABLE), you can modify the underlying database table while scrolling. For example, the following statements will update the job title of the employee based on the current cursor's position:

rs.updateString("JOB","Manager");
rs.updateRow();

**JDBC-ODBC Bridge**

If you'd like to try JDBC-ODBC bridge (driver of type 1), create a new Data Source Name (DSN) by selecting ODBC Data Sources (find it under Control Panel) | Add | Oracle ODBC Driver. After pressing the
button Finish, enter only the data source name, i.e. MyOracleData and press the button OK.
To create a DSN pointing to a remote Oracle database, the Service Name field has to contain the name of the computer, port, and the oracle service name.

Connection part of the class EmployeeList should look as follows:

```java
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String dsn = "jdbc:odbc:MyOracleData";
Connection conn = DriverManager.getConnection(dsn, "scott", "tiger");
```

## Class PreparedStatement

This is a subclass of the Statement, which pre-compiles the SQL statement before execution and could take parameters. Let’s say we need to execute the same query “SELECT * from EMP WHERE empno=...” multiple times providing an empno values from an array empNumbers[]. If we use the class Statement, the sqlQuery will be compiled in each iteration of the loop:

```java
for (int i=0;i<empNumbers.length; i++){
    sqlQuery="SELECT * from EMP WHERE empno=" + employees[i];
    stmt.executeQuery(sqlQuery);
}
```

The class PreparedStatement gives a more effective solution:

```java
PreparedStatement stmt=conn.prepareStatement(" SELECT * from EMP WHERE empno=?");
for (int i=0;i<empNumbers.length; i++){
    // pass the parameter - replace the question mark
    stmt.setInt(1,employees[i];)
    stmt.executeQuery(sqlQuery);
}
```

In this case, the SQL statement is only compiled once and parameters are provided by the appropriate setXXX() method depending on the data type. The first argument provides a parameter’s number. For example,

```java
PreparedStatement stmt=conn.prepareStatement("SELECT * from EMP WHERE empno=? and ename=?");
for (int i=0;i<empNumbers.length; i++){
    // pass the parameter - replace the question mark
    stmt.setInt(1,employees[i];)
    stmt.executeQuery(sqlQuery);
}
```
stmt.setInt(1, empNumbers[i];)
stmt.setString(2, empNames[i];)
stmt.executeQuery(sqlQuery);
}

A special method setNull() should be used if the value NULL should be used in a query.

Class CallableStatement

This class extends the PreparedStatement and used for executing database stored procedures from a Java program. Let’s say there is a stored procedure changeEmpTitle that takes two parameters: empno and title. Here’s the code to execute it:

```
CallableStatement stmt = conn.prepareCall("call changeEmpTitle(?,?)");
stmt.setInt(1, 7566);
stmt.setString(2, "Salesman");
stmt.executeUpdate();
```

If a stored procedure returns some values using output parameters, each of the OUT data types has to be registered before the statement is executed:

```
CallableStatement stmt = conn.prepareCall("call getEmpTitle(?,?)");
stmt.setInt(1, 7566);
stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
stmt.executeQuery();
String title=stmt.getString(2);
```

Batch Updates

Sometimes several database modifications have to be processed as a batch, and if one of the updates fail, the whole transaction has to be rolled back. The database operations have to be explicitly committed in case of success, or rolled back in case of failure:

```
try{
    conn.setAutoCommit(false);
    Statement stmt = con.createStatement();
    stmt.addBatch("insert into Orders " +
```
"values(123, 'Buy', 'IBM', 200)";
stmt.addBatch("insert into OrderDetail " + "values('JSmith', 'Broker131', '05/20/02')");
stmt.executeBatch();

conn.commit(); // Transaction succeeded

} catch (Exception e) {
    conn.rollback(); // Transaction failed
    e.printStackTrace();
}

My Brokerage Firm With the Database

In this assignment, you'll need to modify the thread PortfolioFile from Lesson 10. Instead of reading portfolio data from a flat file, it should connect to a database and select the data from the table Portfolio, which could be created using the following SQL statement:

create table Portfolio(
    id     NUMBER  NOT NULL,
    symbol    VARCHAR2(10) NOT NULL,
    quantity    VARCHAR2(10) NOT NULL,
    price       NUMBER       NOT NULL,
    PRIMARY KEY (id)
);

Run the above statement in Oracle SQL*Plus utility. If you are using another DBMS, just modify the data types and use respective SQL utility. The following statements will insert the sample data into the table Portfolio:

insert into Portfolio  values (1,'IBM',500,105.50);
insert into Portfolio  values (2,'AMZN',1000,15.25);
insert into Portfolio  values (3,'SUNW',2000,32.50);

Your first step is a simple copy/paste operation to replace the code that reads the file in PortfolioFile.run() with the code that selects the data from the table Portfolio. When this mission is accomplished, think of making the code more efficient by reusing the Connection object (peek into the section Database Connection Pools in the Lesson 19).
Resources

1. JDBC driver from Sun Microsystems:
   http://java.sun.com/products/jdbc

3. SQL Tutorial:
   http://www.sqlcourse.com

4. JDBC Tutorial:
Lesson 12

Introduction to Swing

Swing is a library of Java classes, which are located in the package javax.swing, and could be used for the creation of GUI components instead of AWT classes, because of the following reasons:

- Swing contains a lot more of the GUI components than AWT - JTable, JTree, JTabbedPane, etc.

- It contains many lightweight components, which simply means that it's smart enough to draw components such as JButton without the help of the operational system.

- It allows an easy change of the windows appearance using so called pluggable look and feel.

- Many of the Swing components were designed using the Model-View-Controller (MVC) pattern, which separates presentation classes from the data storage/processors. For example, a JTable that looks like a spreadsheet could store its data in a different class – descendent of the class AbstractTableModel.

Swing component names start with the letter J. For example, AWT class Button corresponds to the Swing's JButton, class java.awt.List has a counterpart JList, etc. It's not recommended though to mix AWT and Swing components on the same screen.

The goal of this lesson is to explain the principles of working with Swing objects focusing on the JTable component. Besides being a very useful component, it illustrates the use of the MVC pattern.

Swing Components And Containers

The screen design is nothing more than the process arrangement of GUI components (JButton, JLabel, JTable, etc.) within some
containers such as JFrame, JApplet, JDialog, and others. You could also use a JPanel component which allows you to group several components and add them to a window as one piece. Yet another important Swing container is a content pane – the panels and components should not be added to a JFrame directly, but rather to its content pane, for example compare the following code fragments:

AWT:  myFrame.add(myButton);

Swing: myFrame.getContentPane().add(myButton);

Let’s look at the sample logon screen class that consists of two labels, two text fields, and two buttons. First, we add these components to a panel, and then the panel to the content pane of the frame.

```java
global java.awt.*;
global javax.swing.*;
global class LogonScreen extends JFrame {
    JLabel lblID  = new JLabel("Enter ID:");
    JLabel lblPassword =
        new JLabel("Enter password:");
    JTextField txtID = new JTextField( );
    JPasswordField txtPassword =
        new JPasswordField( );
    JButton btnOK = new JButton("OK");
    JButton btnCancel = new JButton("Cancel");
    JPanel myPanel = new JPanel();
    public LogonScreen ( ) {
        super("Logon Screen");
        setSize(300, 300);
        myPanel.setLayout(new GridLayout(2,3 ));
        myPanel.add(lblID);
        myPanel.add(txtID);
        myPanel.add(lblPassword);
        myPanel.add(txtPassword);
        myPanel.add(btnOK);
        myPanel.add(btnCancel);
```
// Adding the panel to the content pane
Container contentPane = this.getContentPane();
contentPane.add(myPanel);
}

public static void main(String[] args) {
    JFrame myFrame = new LogonScreen();
    myFrame.setVisible(true);
}
}

The above example could have been re-written without the use of the panel – in that case, all components should have been added to the content pane directly.

Using Threads with Swing

When a user works with the Swing window, various events causing screen updates may happen. For example, some class initiated database retrieval operations and the result set has to be displayed in a JTable. All drawings and event handling in Swing are done in the event-dispatching thread. This means that we should not just directly assign the data to a Swing component, but rather ask the event-dispatching thread to do it. Most of the Swing components are not thread safe. Only some of the methods automatically use the event-dispatching thread, for example setText() in a text field, actionPerformed() and paint().

There are two methods in the class SwingUtilities that have to be used to ensure that Swing components are updated from the event-dispatching thread: invokeAndWait() and invokeLater(). Both methods expect an argument: a Runnable object that performs business processing. The method invokeLater() returns immediately without waiting for the completion of the launched process, as opposed to invokeAndWait() that will wait.

The following example shows how to call a method readMyDatabase() that reads order-related data from a database and puts the result into the JTextArea called txtMyOrderData. This method is called from a thread myRunnable that is started by the method invokeLater().

void getOrderData(int orderID) {
    Runnable myRunnable = new Runnable() {
        public void run() {

try {
    readMyDatabase(orderID);
} catch (Exception e) { e.printStackTrace();}

SwingUtilities.invokeLater(myRunnable);

void readMyDatabase(int orderID){
    // some database operations go here to populate a String result
    ...
    txtMyData.setText(result);
}

Using JTable – the Big Picture

The JTable is an excellent component for displaying any tabular data in a spreadsheet-like manner. The data is represented as rows and columns, that’s why the JTable component is often used to present data from relational databases which also store data as rows and columns. This component was built using a MVC pattern, that’s why the visible portion and the data storage are located in different classes. The JTable component is a visible portion (the View part of MVC) and another class that implements theTableModel interface (the Model and Controller) stores the data. You can find more samples of the MVC usage in the Lesson on JSP.

In fact, JSDK comes with an excellent set of Swing samples with the source code demonstrating the use of various Swing components. Open a command window, go to the demo directory in your JSDK installation and start the demo, for example:

c:\jdk1.4\demo\jfc\SwingSet2>java -jar SwingSet2.jar
Below is a fragment of one of the JSDK screens demonstrating the use of the JTable.

Swing provides the class `AbstractTableModel`, that implements the `TableModel` interface, and also has methods to notify a `JTable` when the data is changing.

A programmer usually creates a subclass of the `AbstractTableModel` and this class has to contain the data in some collection object, for example in a 2-dimensional array or a `Vector`. When the `JTable` needs to be populated, it gets the data from a subclass of the `AbstractTableModel` using such callback methods as `getColumnCount()`, `getValueAt()`, and others. When an instance of a `JTable` is created the program assigns the corresponding table model class.

The `TableModelListener` interface should also be mentioned here. This interface defines just one method `tableChanged()`. This method should contain the code performing data modifications, for example code to save the data in the database.

```java
public class MyFrame extends JFrame implements TableModelListener{
    MyTableModel myTableModel;
```
JTable myTable;

MyFrame (){
    myTableModel = new myTableModel();
    myTable = new JTable(myTableModel);

    // Register an event listener
    myTableModel.addTableModelListener(this);
}

public void tableChanged(TableModelEvent e) {
    // Code to process data changes goes here
}

public static void main(String args[]){
    JFrame myFrame = new JFrame( "My Test Window" );

    // Add the JTable to frame and enable scrolling
    myFrame.getContentPane().add(new JScrollPane(myTable), BorderLayout.CENTER);
    this.pack();
    frame.setVisible( true );
}
}

In very simple cases you can create a JTable without the creation of the table model class (see its no-argument constructor), but Java will internally use its DefaultTableModel class anyway.

### Using Table Models

The class that implements the TableModel interface will feed the data to the JTable and must have the three callback methods `getColumnCount()`, `getRowCount()`, and `getValueAt()`:

- `int getColumnCount()`

  This method is called by a JTable once and has to return the number of columns that your data contains. For example, if you are planning to display orders that consist of the `Order ID`, `Stock Symbol`, `Quantity`, and `Price`, just put one line in this method:

    return 4;
int getRowCount()

This method will also be called once. Since the data is placed into a Vector or array before they appear on the screen, the code of this method could look like this:

return myData.size();  //myData is a Vector in this sample

Object getValueAt(int row, int col)

This method will be called once for each cell (row and column intersection). You have to provide the code returning the value for the requested row and column. Let’s say you store instances of class Order in the Vector myData. The class Order has methods getXXX() to return its attributes. In the code fragment below, I’m using Java wrapper classes because the method has to return objects and not primitives. Below is a sample code that you could use this method:

switch (col) {
    case 0:
        return ((Order) myData.elementAt(row)).getOrderID();
    case 1:
        return ((Order) myData.elementAt(row)).getSymbol();
    case 2:
        int qty =((Order) myData.elementAt(row)).getQuantity();
        return new Integer (qty);
    case 3:
        double price = ((Order) myData.elementAt(row)).getPrice();
        return new Double(price);
    default:
        return "";
}

You could find an example of code that populates a Vector with instances of user objects retrieved from a database in the lesson explaining JDBC.

The following optional callback methods could be overridden in the table model class:

String getColumnName(int col)

This method will be called once for each column and its return value will be used as a column’s header. By default, the column headers will just be numbered as it’s done in popular spreadsheet programs, for example:

...
String[] orderColNames = { "Order ID", "Symbol", "Quantity", "Price"};
...
String getColumnName(int col) {
    return orderColNames[col];
}

• boolean isCellEditable(int row, int col)

If you want to make some of the columns or cells not editable, just return false from this callback for the data in question. Here’s how to make the first column of the JTable non-editable:

boolean isCellEditable(int row, int col) {
    if (col == 0)
        return false;
    else
        return true;
}

• Class getColumnClass(int col)

This method is called once for each column. If not provided, Java will display all your data as left-justified strings:

Class getColumnClass(int col) {
    return getValueAt(0, col).getClass();
}

• void setValueAt(Object value, int row, int col)

Override this method if your JTable is editable. It’s called automatically when the user changes the value in a table cell. This is the right place to sync up the screen and model data:

void setValueAt(Object value, int row, int col) {
    switch (col) {
        case 0:
            ((Order) myData.elementAt(row)).setOrderID((Integer) value);
            break;
        case 1:
            ((Order) myData.elementAt(row)).setSymbol((String) value);
            break;
        case 2:
            ((Order) myData.elementAt(row)).setQuantity((Integer) value);
break;
case 3:
    ((Order)
myData.elementAt(row)).setPrice((Double) value);
    break;
  }
  }

• Vector getModel()

  This method returns the model data. This is not a callback, but it’s a good idea to add it to your table model class. If any other class will need the data, say for a database update, this method could return it, for example:

```java
Vector getModel()
{
    return myData;
}
```

How to Save JTable Data in the Database

The method `setValueAt()` has to modify the content of the `Vector` with data. If the database has to be updated, you should decide if it has to be done immediately when a cell is updated, or once when all changes are done. For example, to apply each cell’s change to the database, call the method `fireTableCellUpdated()`. To apply all changes at once – call the method `fireTableDataChanged()`. These method calls will invoke the method `tableChanged()` in the class implementing the `TableModelListener` interface. If the GUI class implements it, you should register the listener with the model and add the method `tableChanged()` to it:

```java
public class MyFrame extends JFrame implements TableModelListener{
    ...
    MyFrame(){
        myTableModel.addTableModelListener(this);
    }
    public void tableChanged(TableModelEvent e) {

        // Get the first modified row
        int row = e.getFirstRow();

        // Get the modified column
        int column = e.getColumn();
```
Order modifiedOrder =
(Order) myTableModel.getModel().elementAt(row);

// Add some JDBC calls here to save the modified order in the database.
...}
}

The class TableModelEvent that is given to the method tableChange() will contain the useful information about modifications in your model. For example, the method getFirstRow() returns the number of the first modified row, getColumn() gives you the modified column, etc.

### Working with TableCellRenderer Interface

Each TableColumn has an associated interface TableCellRenderer that defines a method getTableCellRendererComponent(). This method prepares the content for the cells of a JTable and returns a Component class that will be used to actually render a cell. A default renderer (data feeder) is always used unless you create a custom one. Custom renderers gives you full control of how the cell is displayed.

The class DefaultTableCellRenderer extends JLabel and is the Swing’s implementation of the TableCellRenderer interface.

Let’s look at the example that formats the text in first column to be center justified. We’ll extend the DefaultTableCellRenderer, than gets a hold of a JLabel returned by the method getTableCellRendererComponent() and sets the label properties as desired.

```java
// Get the reference to the first column
TableColumn column = table.getColumnModel().getColumn(0);

// Create a new cell renderer as an anonymous inner
// class and assign it to the column

column.setCellRenderer(
    new DefaultTableCellRenderer(){
        public Component getTableCellRendererComponent(
            JTable table, Object value, boolean isSelected,
            boolean hasFocus, int row, int col) {

                JLabel label =
                super.getTableCellRendererComponent(
```
table, value, isSelected, row, col);

label.setHorizontalAlignment(JLabel.CENTER);

return label;
}
} // end of getTableCellRendererComponent
} // end of new ...

};

Resources

1. Java Swing Tutorial:
   http://java.sun.com/docs/books/tutorial/uiswing/

2. Fundamentals of JFC/Swing. Part 1:

3. Fundamentals of JFC/Swing. Part 2:
Lesson 13

Miscellaneous Topics

Casting

Casting allows you to assign objects of one type to the reference variable of a different type. It has been introduced in Lesson 9 when an object returned by the `Vector`'s method `elementAt()` was casted from `Object` to `Customer`:

```java
Customer cust = (Customer) customers.elementAt(1);
```

Casting could be done with the objects having something in common, for example, ancestor-descendent relationships. The method `elementAt()` returns an `Object`, but since all classes in Java are its descendents, the casting is allowed here. Casting of descendent classes to the ancestors is done automatically, for example:

```java
Object cust = new Customer();
```

You can find another example of the automatic casting in the class `Tester` from the section “Abstract Classes” below.

If you will try to cast apples to oranges, the `IllegalCastException` will be thrown, for example:

```java
Orange org=new Apple();  // throws an exception
```

You can also cast an object to the interface implemented by this class. Let's say a class `Customer` has the following definition:

```java
class Customer implements Annoyable, Giveable, Receivable{
...
}
```

All of the statements below are valid:

```java
Annoyable ann = new Customer();
Giveable giv = new Customer();
Receivable rcv = new Customer();
```
The difference between the variables `ann`, `giv`, and `rcv` is that each of them will only see the methods that were defined in the corresponding interface.

**Abstract Classes**

A class is called abstract if it has at least one abstract (not implemented) method. The keyword `abstract` has to be placed in the definition of the method(s) and the class itself. For example, the following class has one concrete and one abstract method:

```java
abstract class Person {

    public void greetMe(){
        System.out.println("What's up?");
    }

    abstract boolean raiseSalary(int percent);
}
```

The abstract classes can not be instantiated. The class `Employee` is a subclass of `Person` and it implements the method `raiseSalary()`:

```java
class Employee extends Person{

    boolean raiseSalary(int percent){

        if (percent<10){
            // The code to increase salary goes here
            ...
            return true;
        } else{
            System.out.println("This year our limit is 10%, sorry...");
            return false;
        }

    }
}
```

Abstract classes allow you to create superclasses with implementing some of the functionality, while leaving one or more methods to be implemented in the subclasses. This concept could be better explained through example.
Everyone who works in our company is a Person, but some persons are employees and others are consultants. The only difference between them is how they are promoted – employees receive a new salary and a one week Florida vacation, and consultants get their hourly rate increased and also go to the beach. The class Person can have dozens of concrete methods that are the same for every person, but the raiseSalary() should stay abstract. Please pay attention to the fact that even though this method is abstract, it could be called in the abstract class assuming that by the time the concrete class will be instantiated, the method will be implemented. This method will be implemented differently in the subclasses Employee and Consultant:

abstract class Person {
    ...
    public void promote(int percent){
        sendToFlorida();
        raiseSalary(percent);  // calling the abstract method
    }

    abstract boolean raiseSalary(int percent);
}

class Employee extends Person{
    boolean raiseSalary(int percent){
        // The code increasing the salary goes here
    }
}

class Consultant extends Person{
    boolean raiseSalary(int percent){
        // The code increasing the hourly rate goes here
    }
}

The class Tester below could find out the person's work status from the database and instantiate the appropriate concrete class. Please note that the variable worker has the data type of a superclass, but could be used to hold the reference to instances of Employee and Consultant. This feature is called run-time binding and could also serve as an illustration for the automatic upcasting.

class Tester {

    Person worker=null;
    ...
    if (status.equal("EMP"){
        worker = new Employee();
    }else {
The designer of the class Person may not know specifics of the raising salary process, which does not stop him from calling the method raiseSalary(). Programmers who will write the subclasses are forced to write the implementation of this method according to its signature declared in the abstract class. If they declare the raiseSalary() with a different argument list, this will be considered method overloading and the subclass will remain abstract.

A similar functionality could be implemented by using interfaces (see the Payment interface in Lesson 5) and this might be your only choice if you cannot select a superclass for the class Employee. While both abstract classes and interfaces can ensure that a concrete class will definitely have all required methods, the abstract classes can also provide methods that are already implemented, but the interfaces can not.

Polymorphism

One of the main features of any Object-Oriented languages is polymorphism, which is easier to understand through example. We'll look at the classes Person, Employee, and Consultant from a different angle. Look at the class Tester below - suppose we read the records from the database table, and based on some flag, populate a Vector with instances of classes Employee or Consultant.

```java
import java.util.Vector;
import java.util.Enumeration;
public class Tester {
    public static void main(String[] args) {
        Vector workers = new Vector();

        workers.add(new Employee());
        workers.add(new Consultant());
        workers.add(new Employee());
        workers.add(new Employee());

        Enumeration enum = workers.elements();
        while (enum.hasMoreElements()){
            // Raise salary for every worker
            ((Person)enum.nextElement()).raiseSalary();
        }
    }
}
```
The output of the Tester looks like this:

Raising salary for Employee...
Raising hourly rate for Consultant...
Raising salary for Employee...
Raising salary for Employee...

Both classes Employee and Consultant are inherited from the same base class Person, which declares a method raiseSalary(). This allows us to cast every element of the collection workers to the class Person regardless of what the actual data type of the current worker is. This is an example of polymorphism - instead of having different methods for increasing the worker's compensation, we give a polymorphic behavior to the method raiseSalary(), which behaves differently depending on the type of object from the collection. Even though it looks like we are calling the same method raiseSalary(), different methods are being called.

abstract public class Person {
    abstract public void raiseSalary();
}

public class Employee extends Person{
    public void raiseSalary() {
        System.out.println("Raising salary for Employee...");
    }
}

public class Consultant extends Person{
    public void raiseSalary() {
        System.out.println("Raising hourly rate for Consultant...");
    }
}

The while loop in the class Tester will remain the same even if we'll add some other type of worker inherited from the class Person. The actual type of the object that will be casted to the Person is only identified during the run-time.

Java reflection is another way of identifying the class during the run-time and we'll discuss it in the next section of this lesson.
Reflection

Reflection is a feature that allows you to find out the internals of a Java class (methods, constructors, fields) during the run-time. After "questioning" the class about its methods you can invoke these methods passing the required arguments and receiving returned values, if any. A special class called `Class` can load the class in memory and after that you can explore the content of the class by using classes from the package `java.lang.reflect`. The following example loads our class `Employee`, prints its method signatures, finds its superclass and its methods:

```java
import java.lang.reflect.*;
public class ReflectionSample {
    public static void main(String args[]) {
        try {
            Class c = Class.forName("Employee");
            Method methods[] = c.getDeclaredMethods();
            System.out.println("The Employee methods:");

            for (int i = 0; i < methods.length; i++){
                System.out.println("*** Method Signature:");
                methods[i].toString();
            }

            Class superClass = c.getSuperclass();
            System.out.println("The name of the superclass is ");
            superClass.getName();

            Method superMethods[] =
                superClass.getDeclaredMethods();
            System.out.println("The superclass has:");

            for (int i = 0; i < superMethods.length; i++){
                System.out.println("*** Method Signature:");
                superMethods[i].toString();
                System.out.println("Return type: ");
                superMethods[i].getReturnType().getName();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

This is the output of the program `ReflectionSample`:

The Employee methods:
*** Method Signature:public void Employee.raiseSalary()
The name of the superclass is Person
The superclass has:
  *** Method Signature: public abstract void
  Person.raiseSalary()
       Return type: void

Some other methods of the class Class are: getInterfaces(), getConstructors() and getFields().

The following code snippet shows how to get the names, types and the values of the public member variables of the loaded class:

```java
Class c = Class.forName("Employee");

Field[] fields = c.getFields();
for (int i = 0; i < fields.length; i++) {
    String name = fields[i].getName();
    String type = fields[i].getType().getName();

    System.out.println("Creating an instance of Employee");
    Object obj = c.newInstance();
    Object value = fields[i].get(obj);
    System.out.println("Filed Name: ", Type: ", Value: ",
                          + type + " Value: " + value.toString());
}
```

Obviously, the most interesting part of reflection is the ability to not only find out what the method signatures are, but also call these methods. The method invoke() will let you do this, for example:

```java
Class c= Class.forName("Employee");
Method raiseSalary = c.getMethod( "raiseSalary", null);
raiseSalary.invoke(c.newInstance(),null);
```

The first argument of the method invoke() represents the instance of the object Employee, and the null means that this method does not have arguments. The arguments are supplied as an array of objects. You can find out what the method arguments are by calling a method Method.getParameterTypes(), or create and populate them on your own as in the example below. Consider a method with the following signature:

```java
void changeAddress(String newAddress).
```

This method could be invoked as follows:

```java
Class c= Class.forName("Employee");
```
Java reflection allows you to build "properties-driven" applications. For example, you can create a menu based on the .properties file containing the names of menu items along with the method names that have to be invoked. Your class MainMenu could have a total of 50 methods, but each group of users will have a different subset of available methods based on the respective .properties file.

Reflection also helps in building component-based applications. Suppose you have 3 classes: BusinessObject1, BusinessObject2, and BusinessObject3. Different modules of your application may need to use all or some of these objects based on some business conditions. The switch statement is one of the solutions, but if the new condition will come up in the future, the switch statement has to be expanded. A better solution could be creation of .properties files having the names of the business objects in the proper sequence. Reflection classes will instantiate the objects and invoke the methods, and if a new combination of them will be needed in the future, just provide a new .properties or the XML file.

Cloning

An instance of a Java object could be created using one of the following:

- operator new
- method newInstance()
- object cloning

We've been using new a lot, newInstance() in the previous section, and it's time to clone an object. Suppose you created an object TradingOrder having 20 instance variables. Now you want to create another instance of this object, which will have most of the fields the same as the first one (commissionRate, brokerID, brokerPhone, etc). Only a few of the fields will be different - symbol, quantity, and orderType. If you use the operator new to create this instance, all 20 fields have to be initialized.
The better way is to clone the first object, which will create exactly the same TradingOrder, and then only modify the fields symbol, quantity, and orderType.

A Java class must implement the Cloneable interface to be cloneable. The process of cloning is performed by the method clone(), which is declared in the class Object and has to be overridden in the user's class, for example:

```java
class TradingOrder implements Cloneable {
    private int symbol;
    ...
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e.toString());
        }
    }
}
```

The exception CloneNotSupportedException should never happen since we did not make mistakes in the method clone() implementation, but if it does, the code throws the InternalError, which means "unexpected error". Please note that the method clone() has been declared as protected in the class Object and you may need to change its access level to public or package, to allow non-descendent classes to use it:

```java
class Tester {
    public static void main(String[] args){
        TradingOrder order1 = new TradingOrder();
        order1.setSymbol("IBM");
        order1.setComissions(7);
        ...
        TradingOrder order2 = (TradingOrder)order1.clone();
        order2.setSymbol("SUNW");
    }
}
```

Since the method clone() returns an Object, it has to be casted to the proper data type.

The above sample creates a shallow copy of the TradingOrder, which could be fine if this object's fields do not refer to other objects. The shallow copy would not create new instances of the referred objects, but rather create copies of the object's fields pointing to the same ones. The sample below shows you how to create a deep copy of the slightly modified version of the TradingOrder class. The class below has a reference to an object
OrderDetail. I assume that the class does not have references to other objects, otherwise the deep copy should have been done on those objects also.

class TradingOrder implements Cloneable {
    private int symbol;
    OrderDetail od = new OrderDetail();
    ... 
    public Object clone() {
        try {
            TradingOrder tOrd = (TradingOrder)super.clone();
            tOrd.od = new OrderDetail();
            // populate the fields of the tOrd.od here
            ... 
            return tOrd;
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e.toString());
        }
    }
}

The object OrderDetail in the example above could also be re-created using Java serialization.

Resources

1. The Reflection API:
   http://java.sun.com/docs/books/tutorial/reflect/

2. Using Java Reflection:
   http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection/

3. Writing Abstract Classes and Methods:
   http://java.sun.com/docs/books/tutorial/java/javaOO/abstract.html

4. Technical Tips. Cloning Objects
Lesson 14

Java Servlets

Java 2 Enterprise Edition - J2EE

The J2EE specification defines components for implementing services as multi-tier distributed applications that are easy to access, manage and scale (increase the data throughput). J2EE defines multi-tier model: a client tier, a middle tier and a backend tier. The middle tier could consist of Web and EnterpriseJava Beans containers.

Multiple software vendors are working on J2EE compliant application servers. A large portion of the Java application server's market is shared by the WebLogic (BEA Systems) and WebSphere (IBM). Some other players on this market are: Oracle Application Server (Oracle), JRun (Macromedia), IPlanet (Sun Microsystems), PowerTier (Persistence) ...

J2EE is a component based architecture that contains the following major components:

- JDBC (access to relational Database management Systems)
- JNDI (Java Naming and Directory Interface)
- JTA (Java Transaction API)
- JMS (Java Messaging Service)
- EJB (Enterprise Java Beans)
- RMI-IIOP protocol
- Java IDL to communicate with CORBA objects
- Java Mail
- JAF (Java Activation Framework) – for Java Beans support
- Java Servlets
- JSP (Java Server Pages)
- Java Applets
- Java Applications
- Connectors
Introduction To Servlets

Starting from this lesson we are moving to the exiting world of the server side programming, when most of the processing is done not on the client computer, but on the remote one. When you buy something on the Internet the whole process of purchasing most likely is done on the server side. The Web applications may utilize Java applets on the client’s machine, but they have security restrictions and depend on the version of the JVM that user’s browser supports. Beside that if the browser has to download large Java program, it increases the wait time of your users. That’s the better choice is to keep only light-weight HTML pages on the client’s machine (thin client) and Java programs running on the server side. This computer is located in your company and could be as powerful as you can afford, has the proper version of the JVM, could be upgraded if needed, optimized, etc. Java Servlets is one of the ways to support the Web applications on the server side.

Let’s design the on-line store SDPBooks.com. Here’s the setup and the players:

1. The client’s machine just need a Web browser (the old and rusty one is fine). The web pages will use HTML to get the user’s input and send it for processing to SDPBooks.com

2. The SDPBooks.com machine has to run some Web Server software (usually it runs on port 80) that will “listen to” the users’ requests. If it’s a simple request of a static HTML page, the Web server will process the request alone, without any additional software.

3. The SDPBooks.com will also run the Servlet Engine software - a special JVM where Java servlets are running. If it was a user request to find books based on some criteria, we’d need to access the database and create the HTML pages dynamically. In this case the Web server passes the request over for processing to the appropriate Java Servlet (see the action attribute in the HTML <form> tag in the next section of this lesson).

4. The servlet creates on the fly the HTML page with requested information, and passes it back to the Web server which, in turn, sends it to the user.
5. The user's browser displays the received page without knowing if it was a static HTML page, or the fresh one right from the oven.

**Thin Client – The Front Tier**

Let's create a simple HTML screen with a text field and Submit button to find a book by some criteria.

```
<html>
<head>
<title>Find a book</title>
</head>
<body>
Enter a word from the book title:
<form action="http://www.SDPBooks.com/servlet/FindBooks" method=Get>
<input type=Text name=booktitle>
<input type=Submit value="Search">
</form>
</body>
</html>
```

**Assignment:**

1. Create a file `BookSearch.html` containing the text above.

2. Open this file in a web browser using the menu File | Open, and enter something in text field and press the button Search.

3. Read the URL field in your browser and the error message, and try to explain what you see.

**Servlet - The Middle Tier**

Java Servlets run in a servlet engine (i.e. Tomcat) or in an application server (WebLogic, WebSphere, JRun, etc.).

Servlet engines provide a multi-threaded environment for deployed servlets. Each client's request automatically spawns a new thread without need of
thread programming. This gives a huge advantage to servlets over the CGI programming using, say, the Perl language.

Below is the class hierarchy of Java servlets assuming that the name of your class is `FindBooks`:

```
GenericServlet
    service(), init(),
    destroy() …

HttpServlet
    doGet(), doPost() …

FindBooks
    doGet()
```

**Browser-Servlet Data Flow**

Since we have a situation when 2 computers talk to each other, it's very important to understand what, when and where happens if a Web page invoke a servlet. The table below lists these steps on the example of buying a book online.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1.</td>
<td>User enters the book information and presses the button “submit” on the sdpbooks.com.</td>
</tr>
<tr>
<td>Step 2.</td>
<td>The Web browser connects to the <code>FindBooks</code> servlet’s URL specified in the HTML <code>&lt;form action...&gt;</code> attribute and sends the entered data using the method <code>Get</code>, <code>Post</code> or other.</td>
</tr>
<tr>
<td>Step 3.</td>
<td>A servlet engine checks if the specified servlet is already running.</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>Step 4.</td>
<td>If the servlet FindBooks is not running, the engine starts it and calls its method init().</td>
</tr>
<tr>
<td>Step 5.</td>
<td>The engine calls the method service() of the servlet's superclass with arguments HttpServletRequest and HttpServletResponse.</td>
</tr>
<tr>
<td>Step 6.</td>
<td>The method service() calls the FindBook's method doGet() or doPost() depending on the method attribute of the tag &lt;form&gt; with arguments HttpServletRequest and HttpServletResponse.</td>
</tr>
<tr>
<td>Step 7.</td>
<td>YOU write the code in the FindBook.doGet() to &quot;extract&quot; the data entered by the user from the HttpServletRequest object (see the method getParameter() below).</td>
</tr>
<tr>
<td>Step 8.</td>
<td>YOU write the code to process the request (i.e. connect to the database and find requested book).</td>
</tr>
<tr>
<td>Step 9.</td>
<td>YOU write the code to get a reference to the PrintWriter object (see the method getWriter() below) that knows how to send text data to the user. For non-textual data use the class OutputStream instead of the PrintWriter.</td>
</tr>
<tr>
<td>Step 10.</td>
<td>YOU set the type of data that are being sent to the user (see the method setContentType()).</td>
</tr>
<tr>
<td>Step 11.</td>
<td>YOU write the code to prepare and send to the user Strings with HTML tags and relevant data using method println().</td>
</tr>
<tr>
<td>Step 12.</td>
<td>The Web browser happily displays received HTML page to the user.</td>
</tr>
<tr>
<td>Step 13.</td>
<td>User's hand slowly pulls the credit card out of the wallet.</td>
</tr>
</tbody>
</table>

The servlet engine controls when the servlet is loaded, and when its init(), service(), and destroy() methods are called.

The method destroy() is called when a server administrator decides to unload the servlet, or server is gracefully shutting down, or when the server needs to free some resources.

**Your First Servlet**

The servlet below retrieves the name of the book entered by the user and responds with the hard-coded price of $65.

```java
import java.io.*;
```
import javax.servlet.*;
import javax.servlet.http.*;

public class FindBooks extends javax.servlet.http.HttpServlet {

    // Be careful with instance variables – the only instance
    // of the servlet is used by multiple clients.
    String title;

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        String title;
        PrintWriter out = res.getWriter();
        title = req.getParameter("booktitle");
        res.setContentType("text/html");
        out.println("<HTML><BODY>");
        out.println("<H2>the book "+title+" costs only $65")
        out.println("<P>Please enter your credit card number")
        out.println("</BODY></HTML>");
    }
}

Assignment.

1. Enter and compile the above text and save it as FindBooks.java. Compile the class and deploy according to the rules of your Servlet Engine (see some deployment directions at the end of this lesson).

2. Create a Login HTML page and a servlet that will do the following:
   - Connect to a database and create a table users containing two columns: userID and password.
   - Validate a user against the table Users: perform an existence check of the user with the entered id and password.
For valid users, display the HTML page with the information about user's orders from the database table Orders.

If the user name/password is not valid, display the “Try again...” login screen.

### HTTP Get and Post Requests

If the HTML tag `<form>` has the attribute `method=Get`, the servlet has to override the method `doGet()`. In this case the Web browser will append entered values to the end of the URL after the question mark. For example, if the user have entered the work “Apollo” as a book title, the URL will look as follows:

```
http://www.sdpbooks.com/servlet?booktitle=Apollo
```

The method `Get` has the following disadvantages:

- The length of the URL string is limited.
- It could be used only for text data exchange.
- The data is not protected (i.e. entered password will be visible as a part of the URL string).

If the HTML tag `<form>` has the attribute `method=Post`, the servlet has to override the method `doPost()`.

If you’d like to write a servlet that handles both Get and Post requests, write your code, in the method `doGet()` and do the following in the `doPost()`:

```java
public void doPost(HttpServletRequest req,
                     HttpServletResponse res)
                       throws ServletException, IOException {
    doGet(req, res);
}
```

### Session Tracking With Servlets
HTTP is a **stateless** protocol. If a user makes selects some items on the web page and then goes to another one, this second page does not know what has been selected on the first one, unless a session tracking has been utilized.

A **session** is a logical task which user is trying to complete by visiting a web site. For example, the process of buying a book may involve several steps – book selection, input of billing and shipping information, etc. This is an example of a session. Sometimes these kinds of processes are called **shopping cart** applications.

Session information could be stored either on the client or on the server side. The following techniques could be used to keep track of user’s session on the client side:

- Cookies
- URL re-writing
- HTML hidden fields

The server side alternative for storing session data is so called Session tracking API that utilizes the class `javax.servlet.http.HttpSession`.

### Cookies

A **Cookie** is a small portion of data that a Web Server could send for storing on the client's disk. Strictly speaking, the Web Server send the cookie to the client's Web browser and the latter saves the cookie as a file on the user's disk. This file contains some relevant (to this URL) information describing a user. Obviously, your online bank's site and the book store will have different cookies describing you as a customer. Whenever you connect to a URL, your Web browser finds all cookies for this site and sends them to this URL using the object `javax.servlet.http.HttpServletRequest`.

**Pros:**

- Cookies are persistent (stored on a disk).

**Cons:**

- The user can disable the cookies in his/her Web browser.
- Cookies have size restrictions.
The code snippet below shows how a servlet could create and send a cookie:

```java
Cookie myCookie = new Cookie("book","Java Is My Life");
// Set the lifetime of the cookie for 24 hours
myCookie.setMaxAge(60*60*24);
response.addCookie(myCookie);
```

This is how a servlet could retrieve client's Cookies coming with HTTP request:

```java
Cookie[] cookies = request.getCookies();
for (i=0; i < cookies.length; i++){
    Cookie currentCookie = cookies[i];
    String name = currentCookie.getName();
    String value = currentCookie.getValue();
}
```

### URL Rewriting

In case of URL rewriting the session information is attached to the end of the URL string to help the Servlet Engine to identify the client in subsequent requests. It adds the session info the same way as it adds parameters.

**Pros:**

- URL rewriting works even if a user disables the cookies.
- The Session data survive Web server crash because they are stored on the client's machine.

**Cons:**

- The length of the URL string is limited.
- The Session information is visible (unprotected).

The switch between the use of cookies and URL rewriting is handled by the servlet engine and does not requires any special action in the servlet's code.

When a servlet appends the session info to the URL, the method `response.urlEncode()` should be used. This will replace "illegal" characters with their codes. For example, Yakov Pain will look like Yakov%20Pain (a space has been replaced with %20).
The following code

```java
String myURL = "http://www.sdpbooks.com/servlet/FindBooks?id=Yakov Fain";
response.urlEncode(myURL);
```

will send to the browser this string:

http://www.sdpbooks.com/servlet/FindBooks?id=Yakov%20Fain

If servlet parameters contain a question mark or an ampersand, encoding helps Web browsers to tell apart delimiters from the parameters' values.

### Hidden Fields

HTML form's hidden fields is yet another place for storing a session data. When a servlet prepares an HTML page it could include one or more hidden fields. The `hidden` type is one of the allowed HTML `<form>` types for the input fields. Below is the code fragment from a servlet that creates an HTML page and stores selected books in the HTML code.

```java
out.println("<form action=" +
           http://localhost:7001/BookStoreServlet>");
out.println("Select another book title: <input type=text name=item">");
out.println("<input type=submit value=Go>");
String selectedBook="My life in Java";
out.println("<input type=hidden name=item value=" +
           selectedBook + ">");
```

The user will just see an empty text field with a button, but by selecting the “View Source” option of the browser the hidden field with the value “My life in Java” will be displayed as the part of the HTML code.

During subsequent requests, the servlet can retrieve all the values from the hidden fields:

```java
String[] selectedItems=request.getParameterValues("item");
```

Now all previously selected books (our shopping cart) are stored in the array `selectedItems`. 
Hidden fields give you more flexibility than URL rewriting, because you do not have size restriction and your selections are not visible unless an educated user will look at the HTML page’s code. It still not a good idea to send the content of your shopping cart back and forth over the network, that’s why the Session Tracking API has been introduced (see the next section).

Session Tracking API - HttpSession

Instead of sending the context of the shopping cart across the network to the client, you could keep the shopping cart inside of the javax.servlet.http.HttpSession object on the server, in the servlet engine's memory. The engine creates one HttpSession per client and the servlet could store there any serializable objects.

The line below creates or finds previously created session object:

```java
HttpSession mySession = request.getSession(true);
```

The call `getSession(true)` means “find mine session object or create a new one if not found”. Usually a shopping process consists of a number of subsequent servlet calls (list an inventory, add an item to the shopping cart, enter shipping information, etc.). The `getSession(true)` should be used in the very first servlet. At this moment the application server generates a
unique session ID and sends it to the user's Web browser using either a cookie or a URL rewriting.

The `getSession(false)` means “find my session object” assuming that it has been created in the previous steps of the session. If this call returns null, it means that the session object for the session has been destroyed and you might want to display a message saying that the session has expired start the process from scratch.

Let's assume that the following class represents a book:

```java
class Book {
    String title;
    double price;
}
```

This is what you can do in the method `doGet()` of the servlet:

```java
...  
// Find/create the session object
HttpSession session = request.getSession(true);

// We'll use a Vector object here to store selected books.
// Try to get the shopping cart that might have been created during previous calls to this servlet.
Vector myShoppingCart=session.getAttribute("shoppingCart");

if (myShoppingCart == null){
    // This is the first call - create a Vector only once
    myShoppingCart = new Vector();
}

// create an instance of a book object
Book selectedBook = new Book();

selectedBook.title=request.getParameter("booktitle");
selectedBook.price=Double.parseDouble(request.getParameter("price"));

// Add the book to our shopping cart (Vector)
myShoppingCart.addElement(selectedBook);

// Put the shopping cart back into the session object
session.setAttribute("shoppingCart", myShoppingCart);

...
When the order has been placed the program should close the session by making the following call:

```java
session.invalidate();
```

If the session has not been invalidated explicitly, the application server will do it automatically after a specific period of time which is a configurable parameter.

### Testing Servlets in VisualAge for Java

Professional and Enterprise edition of VisualAge for Java come with the Websphere Test Environment in its repository (it does not work under Windows 98). This test environment allows you to test servlets without having Websphere Application Server installed.

To bring this test environment to the VisualAge's Workbench select the following menu items: File | Quick Start | Features | Add Feature. This will open the window with projects that you have in your repository, but not on your Workbench. Select the project IBM WebSphere Test Environment (WTE).

Follow the steps below to test your servlets:

- Select the following menu items: Workspace | Tools | WebSphere Test Environment
- Select the Servlet Engine in the Control Center window.
- Press the button Edit Class Path... and check off the projects that contains your servlet and all other required classes (i.e. JDBC drivers, non-servlet classes, etc.).
- Press the button Start Servlet Engine and wait for the message “Servlet Engine is started” on the console. The servlet engine has been started on port 8080.
- Start your Web browser, and if the name of your servlet is `FindBooks`, enter the following URL:

```java
http://localhost:8080/servlet/FindBooks
```
The servlet engine will find and start the servlet `FindBooks`, and the Web browser will display its output.

The above URL is valid if your servlet’s been created in the default package (the class does not have the `package` statement). If your servlet was created, say in the package `com.sdp`, the correct URL would look as follow:


If the URL is not correct, the Web browser will display an HTTP error 404 – resource not found. If the servlet was found, but some other errors occurs, use VisualAge’s debugger the same was as you would do it with regular Java classes.

**Web Applications**

J2EE specification introduces Web applications and a standardized way of their deployment under any application server that is J2EE compliant. It suggests that all components of a web application should be packaged in the **Web Application Archive** – the file with extension `.war` that could be created with the Java utility `jar`.

The WAR has to store deployed files in a specific directory structure:

- Top level contains the resources that you’d put in a regular document root directory (HTML files, Java Server Pages and client side resources).
- Directory **WEB-INF** could contain the following elements:
  - Deployment descriptor `web.xml` and JSP tag library descriptors. The XML file `web.xml` maps names of the deployed objects to full names of corresponding Java classes and could contain some other application properties such as session configuration, security and others.
  - Subdirectory **classes** that contains compiled servlets, beans and utility classes.
  - Subdirectory **lib** for any additional jar files
  - Subdirectory **tlds** for JSP tag libraries (see the Lesson 15)
Subdirectory META-INF which contains the file manifest.mf with information about the files in this archive.

The following command adds all files from the current directory into the file myBrokFirm.war:

c:\practice> jar cvf myBrokFirm.war *

Create the directory structure on your disk that matches the above description before creating the war. You may also use a utility called Packager that comes with J2EE or any other deployment tool provided by the application server's vendor.

Deployment of your web application could be as simple as simple copying of the war file into the proper directory (see the section “Deploying Web Applications in the WebLogic Application Server”).

**XML in 60 Seconds**

While HTML has a limited number of tags, XML is extensible markup language that allows create your own tags. XML files are stored in plain text files with extension .xml, for example the file Customer.xml could have the following content:

```xml
<? xml version="1.0" ?>
<Customer>
  <LastName>Smith</LastName>
  <FirstName>John</FirstName>
</Customer>
```

Every XML document must be well formed which means that every tag has to have a matching closing tag.

The tags <LastName> and <FirstName> are nested – they belong to the tag <Customer>.

XML documents are called valid if only allowed tags are included in the document. In this case a document type definition file (.dtd) should be created which lists the valid tags and relationships between them. For example a tag <Customer> is allowed to have multiple nested tags <PhoneNumber>, but only one <LastName>.

Should the .dtd file exist, our example above would look like this:
We are not going to discuss the format of .dtd files in this book.

XML is a popular tool for storing any kinds of properties. Application servers store the properties of their own and user objects in the xml files (servlets' properties, EJB deployment descriptors, tag libraries, etc). XML files are replacing old-fashioned .ini and .properties files.

XML files should be read by so called xml parsers – the programs that could read xml files and provide a simple access to the tags' values. Application Servers have their own parsers which read the properties such as deployed servlets, session timeouts, connection pool parameters, and others during the server start up.

In general, XML is an excellent way to standardize the data exchange between different applications.

Please keep in mind that HTML tags allow you to define presentation of your page, as opposed to XML tags that deal only with the data formatting.

Deploying Servlets in WebLogic

The concept of a Web application simplifies the deployment of Servlets and JSP. The WebLogic's installation process creates the applications directory, for example:

c:\bea\wlserver6.1\config\mydomain\applications

After successful compilation of all classes create a .war archive and copy it into the applications directory. The WebLogic server during the startup will automatically deploy all applications located in this directory. For example, if you've created and deployed the servlet HelloWorld.war, you can see it after visiting the system console at: http://localhost/console/. Look at the left panel of your console screen and find the following items:

mydomain
  Deployments
  Webapplications
Double click on the name HelloWorld and look at its properties on the right panel.

To run this servlet direct you Web browser to the following URL:

http://localhost:7001/HelloWorld

My Brokerage Firm With Servlets

Let's re-design the My Brokerage Firm application from the Lesson 10 to use a thin HTML client and the servlets MarketNews and Portfolio. The market news and portfolio data should be retrieved in parallel using Java threads. In the Lesson 16 we'll add an applet StockQuotes that will be communicating with a servlet. Below are the guidelines for the project development.

- To make the HTML generation easier start with creating the file that contains an HTML Table. First row of the table (<tr>) has two cells (<td>) – Market News data and StockQuote applet tag (<Applet>). The second row of the table contains one cell that has a nested HTML table to display the Portfolio data. When your Java code will be ready, cut and paste the HTML from this file into out.println() statements of the respective servlets.

- Re-use the thread classes Portfolio and MarketNews from the Lesson 10. Constructors of these classes should have an argument of type Dispatcher (see below).

- Create a Java class called Dispatcher that will spawn the MarketNews and Portfolio threads from its constructor. After that the class Dispatcher has to wait for notification (which has to be performed only after both threads have returned their outputs.

- The class Dispatcher has to have a method with the following signature:
  setData(String threadName, String data )

  This method has to be called by each thread when its data is ready.
• Create a servlet MyBank.java that will instantiate the Dispatcher and will pass the HttpServletRequest reference to it.

• Add the code to the Dispatcher.setData() that will check if all threads returned data. If the returned data is stored in a Hashtable, check the size of it after adding the thread’s output to it. If the size of the Hashtable is equal to 2, it means that both threads are done.

• Add code to Dispatcher.setData() that will create a class Showman that formats and send the whole HTML page to the user. While creating the Showman make sure that it has a reference to the user’s HttpServletResponse object.

• After the Showman had finished displaying the page, send a notification to the Dispatcher.

This version of My Brokerage Firm is available for the download, but it will be re-designed in the next lessons to utilize Java Server Pages and a Java Beans.

Resources

1. Servlets Tutorial from Sun Microsystems:
   http://java.sun.com/docs/books/tutorial/servlets/

2. Introduction to servlets and IBM WebSphere Application Server:

3. Programming WebLogic Server HTTP Servlets:
   http://e-docs.bea.com/wls/docs61/\\/\\/\\/\\/servlet/index.html
Lesson 15.

Java Server Pages

What’s Wrong With Servlets?

Let’s say, you’ve created and deployed a servlet, which displays a HelloWorld HTML page by calling the method println():

```java
out.println("<HTML><BODY>Hello World </BODY></HTML>");
```

... What if you need to change the layout of this page, i.e. add several empty lines on top? It’s not a big problem – just ask Alex, who is a Java consultant and he’ll be happy to modify the call to `out.println()`, re-compile, and redeploy the servlet. But do we want to keep expensive Alex on staff just to make these minor changes? Of course not – we’d rather ask a junior programmer Matilda to do this job, but the problem is that she knows nothing but HTML. This is where JSP become very handy. Ask Matilda to create an HTML page as shown below, and save it in a file named `HelloWorld.jsp`:

```html
<HTML>
<BODY>
  Hello World
</BODY>
</HTML>
```

This file has to be placed into a special directory known as the `document root` on the machine `xyz.com` where the JSP engine runs. The document root is usually a directory that could be specified in the properties of your application server (see section Testing JSP in VisualAge for Java). Users can access this JSP from the WEB browser by typing something like this:

```
http://www.xyz.com/HelloWorld.jsp
```

Upon the first request to this page, the JSP engine will automatically generate, compile, and deploy a servlet based on the content of the file `HelloWorld.jsp`. Since the servlet `HelloWorld` will be already running, all subsequent jsp calls will be processed a lot faster, because the servlet `HelloWorld` will already be running.
You may say that we could achieve the same effect by creating HelloWorld.html without all these complications. This is true, as long as your page is a static one and does not need to perform any calculation, file or database access, etc. HTML is not a programming but a markup language. It could not perform even simple calculations such as 2 + 2, but JSP can easily do it (see MyCalculator.jsp below).

**Embedding Java Code Into HTML**

JSP use tags that allow you to embed Java code into an HTML page. When the servlet is automatically generated behind the scenes, this Java code will also be included and executed as part of this servlet. JSP tags are included in angle brackets, i.e. `<%=2+2%>`. During the servlet generation process performed by the JSP engine, these tags will be replaced with the regular Java code. For example, the tag `<%=2+2%>` will be replaced with a Java statement like this one:

```
out.println(2+2);
```

Here’s the code for the calculator MyCalculator.jsp:

```html
<HTML>
<BODY>
    HTML created by Matilda goes here...
    You may not know that 2 + 2 is `<%= 2 + 2%>`
    The code between percentage signs was created by Java programmer Alex...
    More HTML created by Matilda goes here...
</BODY>
</HTML>
```

This program (when deployed), will display the following page in your browser:

```
HTML created by Matilda goes here...
You may not know that 2 + 2 is 4
The code between percentage signs was created by Java programmer Alex...
More HTML created by Matilda goes here...

Please note that the expression `<%2 + 2%>` has been substituted with 4.
```
A JSP tag `<%= ... %>` from the example above could contain any Java expression that will be evaluated and its result will be displayed in place of the expression by the JSP engine.

If you need to change the appearance of the page (colors, fonts, data allocation) without changing the expression `(2+2)`, Matilda could do it easily! After the changes are applied, the JSP will be automatically re-generated into a new servlet and re-deployed. Usually you do not even have to restart the server. The only exception are pre-loaded JSP that are set to be initialized on the server startup. Any business logic changes contained inside of the JSP tags will be programmed by Alex.

A Java Server Page is nothing more than a servlet that is automatically generated from a file containing valid HTML and JSP tags.

### Major JSP Tags

- **Directives:**
  Directives do not generate screen output, but inform the JSP engine about the rules that have to be applied to the JSP.

  `<%@ page import="java.io.*" %>`
  `<%@ include file="bankRates.txt" %>`
  `<%@ taglib uri="my_taglib.tld" prefix="test" %>`

  Page directives start with `<%@ page` and are only in effect within the current page. They are used with such attributes as `import`, `extends`, `session`, `errorPage`, `contentType`, and some others.

  Include directives allow inclusion of any text from a file or code from another JSP, at the time when the page is compiled into a servlet, for example:

  `<%@ jsp:include page="calcBankRates.jsp" %>`

- **Expressions:**
  `<%= salary*1.2 %>`

  Expressions start with `<%=` and could contain any Java expression that will be evaluated and the result will be displayed in the HTML page.
• **Declarations:**

```jsp
<%! String lastName; %>
```

The variable above is available in this page only. This is a method declaration:

```jsp
<%! private void myMethod(){
     ...
}
%>
```

The code contained in the declaration block is placed into the servlet body outside of any existing methods.

• **Scriplets:**

```jsp
<% lastName = "Smith"; %>
```

Scriplets could contain any valid Java code that will be included in the resulting servlet’s method `_jspService` during code generation.

• **Comments:**

```jsp
<%-- Some comments --%>
```

These comments will appear on the JSP only. To pass the comments to the Web page, regular HTML comments (`<!--...-->`) should be used instead.

• **Actions:**

```jsp
<jsp:include page="header.jsp" />
<jsp:forward page="someOther.jsp" />
<jsp:plugin type=applet code=Quotes.class jreversion=1.2 >
   <jsp:param> name="Symbol" value="SUNW" /></jsp:plugin>
```

The directive `include` adds the content of the included page during the compile time, while the action `jsp:include` does it during the runtime.

The action `forward` allows you to redirect the program flow to a different page while maintaining the same request and response objects. The other way of redirecting the flow is by using `response.sendRedirect(someURL)`, but in this case, the new request and response objects are created, which leads to an additional connection between the client and the server.

The plugin ensures that if your JSP includes an applet, the Java plugin will be downloaded to avoid JVM compatibility issues. If you have not specified...
The version of the Java runtime, the default one is 1.1. The action `<jsp:plugin type=applet...` has similar attributes to the HTML’s tag `<APPLET>`.

The `<jsp:param>` is used to pass parameters to an applet and they are nested within the action `<jsp:plugin>`:

```xml
<jsp:plugin type=applet code=Login.class...>
  <jsp:params>
    <jsp:param name="userID" value="SCOTT" />
    <jsp:param name="password" value="TIGER" />
  </jsp:params>
</jsp:plugin>
```

**Implicit JSP Objects**

Below is the list of pre-defined variables that you can use in JSP pages. These variables are initialized by the JSP engine and are ready for use.

- **request**
  It has the same functionality as `HttpServletRequest`.

- **response**
  It has the same functionality as `HttpServletResponse`.

- **out**
  It represents the class `JspWriter` and has the same functionality as the variable received by this code:

  ```java
  HttpServletResponse.getWriter()
  ```

- **session**
  This variable represents an instance of the `HttpServletRequest` object.

- **exception**
  This variable represents an instance of the `Throwable` object and contains the error information. This variable is only available from the JSP error page that has the directive `isErrorPage=true` (see the Use of Error Page section below).

- **page**
  This variable represents the instance of the JSP’s servlet. Usually is
not used.

- pageContext
  This variable represents the JSP context and is used with Tag Libraries.

- application
  Provides access to the Web context. See the class ServletContext.

- config
  Provides initialization information by the JSP engine. See the class ServletConfig.

Error Pages

Let’s say we have a file calcTax.jsp containing code that may throw Java exceptions. Instead of scaring users with stack trace output screens, we’d rather prepare a friendly taxErrors.jsp explaining the problem in plain English.

The calcTax.jsp could have some HTML form where users enter their gross income and number of dependents. The code may throw an exception either in the validation stage or during the process of calculation.

<HTML>
  Some code to calculate tax and other HTML stuff goes here ...

  <%@ page errorPage=taxErrors.jsp %>
</HTML>

The sample taxErrors.jsp shows how to use the JSP variable exception, which displays the error message in a user friendly manner, and also contains the exception description for the technical support team:

<HTML>
<BODY>
Dear friend!
<P>
We are sorry to inform you that there was a little problem during your tax calculations.
</P>
Please try to enter only numeric values in the fields GrossIncome and Dependents.

If you think that you did everything right, please contact our award winning technical support team at (732) 598-4027 and provide them with the following information:

```html
<%=exception.toString()%
</BODY>
</HTML>
```

## Java Beans

Java Bean is a class that implements `Serializable` interface, has a no-arguments constructor, private or protected fields and setter/getter methods. Java Beans are used mainly for data storing and exchange. They help to avoid mixing Java code and HTML (see also the Tag Libraries at the end of this lesson). Please note the setter/getter naming conventions used in the code below.

class Student implements Serializable{
    private String lastName;
    private String firstName;
    private boolean undergraduate;
    
    Student(){
        ... 
    }
    
    public String getLastName(){
        return lastName;
    }
    
    public String getFirstName(){
        return firstName;
    }
    
    public void setLastName(String value){
        lastName = value;
    }
    
    public void setFirstName (String value){
        firstName = value;
    }
    
    public void setUndergraduate(boolean value){
        undergraduate = value;
    }
}
public boolean isUndergraduate (){  
  return undergraduate;  
}
}

Please note that Enterprise Java Beans (EJB) is a completely different animal that will be covered later.

**Using Java Beans in JSP**

To use a Bean with JSP, first you need to specify its name, location, and after that you could set or get its properties. These are the samples of bean usage:

```jsp
to useBean id="Student" class="com.sdp.Student" />

<jsp:getProperty name="Student" property="LastName" />

<jsp:setProperty name="Student" property="LastName" value="Smith"/>
```

If you need to use a bean that has been serialized into a file MyStudents.ser, here's the sample:

```jsp
<jsp:useBean id="Student" beanName="MyStudents.ser" type="com.sdp.Student" />
```

If you need to populate bean's properties based on the data from HTML Form, these are the samples (let's say the HTML form has text fields called LName and FName):

```jsp
<jsp:setProperty name="Student" property="LastName" value=""<%= request.getParameter("LName") %>">

<jsp:setProperty name="Student" property="FirstName" value=""<%= request.getParameter("FName") %>">
```

If property names are the same in the HTML form and in the bean, the values entered on the HTML screen could be passed to JSP even simpler using the asterisk:
How Long Does a Bean Live?

If a JSP variable is declared inside a scriptlet, it has a local scope, and to
have an instance scope, the variable has to be declared using the
declaration tag. The bean’s scope could be defined using the scope attribute
of the tag <jsp:useBean>. The short explanation of the various scopes are
given below.

- **page** – the bean is only available within the current page and will be
destroyed as soon as the user exits the page, for example:

  `<jsp:useBean id="Student" class="com.sdp.Student" scope="page" />`

  This is a default JSP scope.

- **request** – the bean’s life span is the same as the request’s object
  one. Even if the control will be redirected to a different JSP using the
tag <jsp:forward>, the bean will be available on that page, because
  it’ll be using the same request object:

  `<jsp:useBean id="Student" class="com.sdp.Student" scope="request" />`

- **session** – the bean is available for all pages until the session
  object expires (see the section “Session Tracking With Servlets” in
  Lesson 14).

  `<jsp:useBean id="Student" class="com.sdp.Student" scope="session" />`

- **application** – the bean is available for all users and all pages – this
  is sort of a global bean.

  `<jsp:useBean id="Student" class="com.sdp.Student" scope="application" />`

Loading JSP From Servlets
Let's say you have a servlet that needs to load a different JSP (or a servlet) based on the user's selection on the HTML screen. If you do not need to reconnect to the user's machine to get new copies of the request and response object, you'd need to create an instance of the `RequestDispatcher` class and call its method `forward()` providing the `HttpServletRequest` and `HttpServletResponse` as arguments, for example:

```java
public class MyServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,
                        HttpServletResponse res){
        ServletContext context = getServletContext();
        RequestDispatcher requestDisp = null;
        String make = req.getParameter("carMake");
        if (make.equals("Toyota") {  
            requestDisp = context.getRequestDispatcher("Toyota.jsp");  
            requestDisp.forward(req,res);  
        }
        else if (make.equals("Nissan") {  
            requestDisp = context.getRequestDispatcher("Nissan.jsp");  
            requestDisp.forward(req,res);  
        }
    }
}
```

The method `forward()` in the sample above passes the request and response objects from the servlet to `Toyota.jsp`, which is sent to the user. In some cases, the current servlet performs all interactions with the user, and just needs to load the code of another servlet or JSP. For this purpose, the method `include()` should be used instead of `forward()`:

```java
requestDisp.include(req,res);
```

Since this redirection happens on the server, the initial URL would still be displayed on the browser's address bar. To provide the new URL (e.g. to allow the user to properly add the resulting page to his “Favorites”), `response.sendRedirect("/a_new_URL")` should be used.

**Testing JSP in VisualAge/WebSphere**
The IBM's VisualAge for Java v.4 comes with the WebSphere test environment supporting JSP version 1.1. It makes the JSP testing process a breeze (see the section “Testing Servlets With VisualAge/WebSphere” from the Lesson 14).

The default document root directory for HTML and JSP files is specified in the following file:

\<VAJ installation dir>\ide\project_resources\IBM WebSphere Test Environment\ SERunner.properties.

The name of the property is docRoot and it's default value is

\docRoot=D:\IBMVJava\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web

If the web directory does not exist, create one and copy your HelloWorld.jsp there. After starting the servlet engine, open your Web browser and enter the following URL:

http://localhost:8080/HelloWorld.jsp

If this JSP was never requested before, it'll be re-compiled into a Java Servlet, deployed, and executed. The first time it'll be slow, but all subsequent calls will be processed much faster, because JSP will already be running.

If the WebSphere's machine is a part of the network, try to access the same JSP from a different computer - start the Web browser and enter the URL replacing the localhost from the example above with the name or IP address of the computer where the HelloWorld.jsp is deployed, for example:

http://110.64.105.98:8080/HelloWorld.jsp

The source code for JSP could be found in a VAJ project called IBM Page Compiled Generated Code. The generated code does not look pretty, but it could be used for debugging with VAJ Debugger.

Another handy tool is the JSP Execution Monitor that allows you to monitor the execution of JSP, map the JSP code with Java code, and set the debugger breakpoints. Enable the use of the JSP Execution Monitor by checking off this option in its Properties screen. By default, it runs on the port 8082, but it could be changed, if needed. Its window will pop up when you try to access a JSP showing the corresponding JSP and Java Servlet's lines of code that are being executed.
Stock Portfolio Project with JSP

Modify the My Brokerage Firm project from the previous lesson. Replace the ShowMan’s functionality with Java Bean/JSP combination by performing the following steps:

- Create a Java Bean `FinancialInfoBean.java` that will store the thread results in a `Hashtable` and will have getter and setter methods. It could look like this:

  ```java
class FinancialInfoBean{
    private Hashtable data=null;
    FinancialInfoBean(){
      public void setData(Hashtable threadOutputs){data=threadOutputs;}
      public Hashtable getData(){ return data; }
    }
  }
```

- Create an instance of the `FinancialInfoBean` in the `Dispatcher` and call its method `setData()` when all thread are finished.

- Create the `MyBank.jsp` containing the HTML Table. The first row of the table (`<tr>`) has two cells (`<td>`) – Market News data and the `StockQuote <Applet>` tag. The second row contains a single cell that has a nested HTML Table to display the Portfolio data.

- The JSP `MyBank.jsp` has to contain the tag

  ```jsp
  <jsp:useBean id="data" class="FinancialInfoBean" />
  ```

- For each data element, insert JSP expressions after appropriate `<td>` tags, for example:

  ```jsp
  <td>data.getData().get("Portfolio")</td>
  ```

Load your JSP from the class `Dispatcher` using the following call:

```jsp
RequestDispatcher.forward().```
Deploying JSP in WebLogic

To deploy a simple JSP, you do not even need to create a war file - just copy HelloWorld.jsp into the default document root directory:

c:\bea\wlserver6.1\config\mydomain\applications\DefaultWebApp_myserver

Start the application server and direct your Web browser to http://localhost:7001/HelloWorld.jsp

It’ll take a couple of seconds to run the JSP for the very first time because the application server needs to generate and compile the JSP into a servlet, but all future requests will be processed a lot faster.

Tag Libraries

Even though JSP allow mixing of HTML and Java code, it’s better to hide Java, so the HTML code and JSP tags are separated (remember MVC paradigm?). Java beans is one of the ways to hide Java code. The other alternative is creation of JSP custom tags grouped in the tag libraries of reusable components. Each custom tag looks similar to regular JSP tags, but it’s always supported by a Java class(es) written by a programmer to provide required functionality.

The following three action have to be performed for implementing a custom tag:

- Create a tag library descriptor - an XML file with extension .tld.
- Create a Java class that provides business logic supporting the tag.
- Register the tag library with the Web Application (see the Lesson 14).

A sample of a tag library descriptor taglib.tld is listed below. The tag called DowJohnes should display a Dow Johnes index value.

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>sts</shortname>
  <uri>http://www.xyz.com:7001/taglib</uri>
  <info>Wall Street tag library</info>
```
The empty body content means that this is a simple JSP tag with no content and could be used like this: `<sts:DowJones/>`.

The class supporting a JSP tag has to implement the interface `javax.servlet.jsp.tagext.Tag`. The JSP engine will call its methods to set the JSP context - `setPageContext()`, start the execution of the tag's code - `doStartTag()`, etc. The other option is to inherit the tag handler class from the `javax.servlet.jsp.tagext.TagSupport`. This class gives you a default implementation of the `Tag` interface and initialized references to the `pageContext` and `parent`.

```java
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class DowJohnesHandler extends TagSupport{
  public int doStartTag() throws JspException{
    String dowQuote;
    // Obtain the DowJohnes quote similarly to the sample
    // StockQuote from the Lesson 8 changing the URL to
    // http://finance.yahoo.com/q?d=t&s="^DJI"
    // and write it to the client
    dowQuote=...;
    try{
      pageContext.getOut().write(dowQuote);
    } catch(IOException) { throw new JspException("...");}
    return SKIP_BODY; // ignore the tag body
  }
}
```

The registration of our tag library makes it visible to your application server and is performed by inserting the following fragment into the file `web.xml` (see the Lesson 14):

```xml
<taglib>
  <taglib-uri>
    http://www.xyz.com/taglib
  </taglib-uri>
</taglib>
```
When all of the above is done, create a simple file test.jsp that is using the tag `<DowJohnes>`:

```html
<html>
<head>

```<%@ taglib uri="http://www.xyz.com/taglib" prefix="sts" %>
</head>
<body>

Today's Dow Jones index: <sts:DowJohnes/>
</body>

</html>
```

If a tag requires some parameters, they should be specified in the `.tld` file using the tag `<attribute>`, for example:

```html
<tag>
  ...
  <attribute>
    <name>tradeDate</name>
    <required>false</required>
  </attribute>
</tag>
```

The setter method has to be provided in the tag handler class for each parameter. The setter methods have to be named using the same naming convention as in Java Beans:

```java
public void setTradeDate(String tradeDate)
{
 ...
}
```

### Resources

1. JSP Tutorial

2. JSP Fundamentals by jGuru

3. Tag Libraries Tutorial
Lesson 16

Two Web Applications

Applet – Servlet Communication

Web applications based on HTML clients that work with servlets is a good choice if the screens are simple. But if you need a little more sophisticated GUI, Java clients with AWT or Swing components should be considered. This time we’ll teach Java applets how to talk to servlets. Below are some of the applet/servlet design considerations:

- Applets are downloaded to the client’s machine, hence we still want to keep them light-weighted.

- To eliminate the need to maintain client’s code, applets should not connect directly to server databases. It’s better to move from the client-server (applet-dbms) to multi-tier architecture (applet-servlet-dbms). With the latter design, applets do not need to download the JDBC drivers to the client’s machine – the servlet does the database interaction.

- Applets do not access user’s disks and should be used primarily for the data entry, validation, display, and simple calculations.

- Applets may not work properly if the user’s Web browser uses an outdated JVM.

- While applets can connect to a remote computer using socket or RMI programming (see Lesson 17), the HTTP protocol and such Java classes as URL and URLConnection simplifies the network programming. With an HTTP protocol, applets can receive/send the text as well as binary objects using Java Serialization.

An applet could collect the user’s input from the GUI components, package the data in some kind of an object, and send it to the servlet. An applet could do it with the help of Text or Object streams.

To receive data from a servlet, the applet has to play the role of a Web browser. It has to understand the data types it receives and display the data in proper GUI components. If an applet expects the text data, it needs to get
a reference to the servlet’s input stream and read the received text, for example:

```java
URL servletURL = new
    URL("http://www.sdbooks.com/servlet/FindBooks");
URLConnection con = servletURL.openConnection();
InputStream in = con.getInputStream();
DataInputStream servletData = new DataInputStream(in);

String bookTitle = servletData.readLine();
String bookPrice = servletData.readLine();

txtTitle.setText(bookTitle);   // display the data in a AWT TextField
txtPrice.setText(bookPrice);
```

The code has the same drawback as any DataInputStream – you have to know the structure and the order of received data, and also you have to convert it into appropriate data types (in our example the book price should be converted to a number to calculate a discount).

That’s why we should consider the use of Java Serialization, which allows easy conversion of a Java object into a stream of bytes.

Let’s look at the applet-servlet conversation using Java serialization (see Lesson 7). Our applet is going to prepare a trading order and send it over to a servlet, which will save this order in a database. This scenario could be implemented by the following steps:

**Step 1.** Create a class TradingOrder that implements Serializable interface.
**Step 2.** Create an applet using Java AWT or Swing components to collect info about the trading order. Include the button Send to submit the order to a servlet.
**Step 3.** Create a database table for storing the trading order and a servlet that can work with the database.
**Step 4.** In the applet
   a) under the actionPerformed() of the button Send:
   b) create an instance of the TradingOrder, initializing it with the values entered on the screen.
   c) Connect to a servlet using the class URLConnection.
   d) Obtain a reference to the OutputStream object of the servlet.
   e) Create an ObjectOutputStream chaining it with the
servlet's OutputStream and give it the order by calling the method writeObject().

f) Close the streams

Step 5. In the servlet:
a) Obtain and reference the applet's InputStream using request.getInputStream().
b) Create the ObjectInputStream, chaining it with the applet's InputStream and call the method readObject() to de-serialize the TradingOrder.
c) Connect to the database and save the received order in a table(s).
d) Close the streams.

A similar scenario could be implemented to send data from a servlet back to the applet. For example, a servlet connects to the database, selects all customer's orders, puts the result set into a Vector, and serializes it into the applet's stream to be displayed in a JTable.

Assignment. Implement the above mentioned steps using the code samples below.

class TradingOrder implements java.io.Serializable{
  private String symbol;
  private int quantity;
  private String type;
  private float price;

  TradingOrder(String symbol, int quantity, String type, float price){
    this.symbol=symbol;
    this.quantity=quantity;
    this.type=type;
    this.price=price;
  }
  public String getSymbol(){return symbol;}
  public int getQuantity(){return quantity;}
  ...
}
Here is the applet’s part:

class Client extends java.applet.Applet implements ActionListener {

    ObjectOutputStream out = null;

    // The GUI components should be created here
    ...

    public void actionPerformed(ActionEvent event) {

        if (event.getSource() == buttonSend) {
            TradingOrder tOrd = new TradingOrder(
                txtSymbol.getText(),
                Integer.parseInt(txtQuantity.getText()),
                listType.getSelectedItem(),
                Float.parseFloat(txtPrice.getText()));

            try {
                URL orderServlet = new URL("http://localhost:8080/servlet/OrderServlet");
                URLConnection con = orderServlet.openConnection();

                // We’ll be only sending data (otherwise call setDoInput(true))
                con.setDoOutput(true);

                // We are sending binary data, that’s why the content type
                // should be the application/octet-stream.
                con.setRequestProperty("Content-Type", "application/octet-stream");

                out = new ObjectOutputStream(con.getOutputStream());

                // Send the TradingOrder to the servlet
                out.writeObject(tOrd);
            } catch ({...

        } finally {
            out.flush();
            out.close();
        }
    }
}
This is the servlet’s part:

```java
class OrderServlet extends HttpServlet {

    // Since we are passing the binary data from the applet, we can’t use doGet()

    public void doPost(HttpServletRequest request, HttpServletResponse response) {
        ObjectInputStream appletStream = null;
        TradingOrder receivedOrder = null;

        try {
            // get the applet’s input stream
            appletStream = new ObjectInputStream(request.getInputStream());

            // de-serialize the order from applet
            receivedOrder = (TradingOrder) appletStream.readObject();

            // connect to the database and save the received order

            // If the servlet needs to send one or more rows to the applet, it could be done in a similar fashion,
            // but this time we’d need to call the request.get Ou tputStream(), create an instance of the ObjectOutputStream and send a serializable object over there.

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            ... 
            appletStream.close();
        }
    }
}
```
The Online Store

Every purchasing process on the Internet consists of the steps that are pretty much the same, regardless of what you are buying. Users perform the following steps:

- Browse the inventory of the online store.
- Select an item and add it to your shopping cart. At this point, you may either proceed to checkout or keep browsing the inventory.
- Enter the payment and shipping information.
- Press that scary looking button “You are about to place THE ORDER! Are you 100% sure?”.
- Save the order in the database and display (or e-mail) the receipt.

The payment information is usually entered on the page located on secure servers that provide data encrypting in the so called secure socket layer (SSL). The URL of this page uses the HTTPS protocol. Only those users who successfully logged on to the system will be allowed to access these servers.

You might sleep a little better knowing that all these online stores do not have direct access to your credit card accounts, but rather delegate the actual validation/charge process to a centralized Sybercash payment system.

Let’s discuss the Java components that could be used for developing of an online store. First of all, we want to utilize the Model-View-Controller design pattern to separate the business logic, presentation part, and the data.

The View portion (the screens) is implemented using JSP.

The Model portion could be done using Java classes and beans communicating with JSP. It should also include some data storage such as database management system or flat files.

The Controller is a navigation object that re-directs control to appropriate classes based on user’s choices or some other events that may happen in the system. A servlet is a good candidate on the role of controller in the online store.

Online stores usually consist of several screens and have the top and side panels with menus. The top panel may show the global menu and a company logo. This global menu is usually displayed on every page. The side panel shows the navigational menu and may be different for each screen. These panels should be created as separate JSPs, and other screens (also JSP) will include the panels as needed using the directive <jsp:include>.

The shopping process consists of multiple steps, so we should take care of session tracking to remember every item that’s been added to the shopping
cart, payment, and shipping information. Java bean will be created for the shopping cart and the HttpSession object will store this bean – the ShoppingCartBean.

The items placed into the shopping cart will be represented by the class Item:

class Item {
    long productCode;
    String description;
    double unitPrice;
    int quantity;

    // Some setters/getters methods go here
}

Since the shopping cart may contain more than one item, the ShoppingCartBean has to be able to store a collection of Items, i.e. Vector:

import java.util.Vector;
class ShoppingCartBean implements Serializable {
    private Vector selectedItems = new Vector();

    ShoppingCartBean() { }

    public Vector getItems() {
        return selectedItems;
    }

    public void addItem (Item selectedItem) {
        selectedItems.add(selectedItem);
    }
}

The following JSPs should be created:

ProductCatalog.jsp
Billing.jsp
Shipping.jsp
Receipt.jsp

Each of these JSPs generate a screen containing an HTML Form element with appropriate information. The same Java servlet, say MainServlet, processes all requests and loads the corresponding JSP. For example, if the button Proceed To Checkout is pressed on the HTML form, the
MainServlet should load the Billing.jsp. If the user pressed the button Continue on the Billing screen, the MainServlet would load the Shipping.jsp. If the user pressed the button Return to shopping, it should load the ProductCatalog.jsp again.

The servlet should be light-weighted and just perform the navigational functions. It should have an if statement, which loads appropriate JSP with the help of the class RequestDispatcher. For example:

class MainServlet extends HttpServlet{

    public void doGet(HttpServletRequest request,
    HttpServletResponse response) {

        HttpSession session = request.getSession(true);
        RequestDispatcher disp = null;

        // Find existing or create a new Shopping Cart
        ShoppingCartBean shoppingCart = (ShoppingCartBean) session.getAttribute(“ShoppingCart”);

        if(shoppingCart == null) {
            session.setAttribute(“ShoppingCart”,
                                new ShoppingCartBean ());
        }

        String nextScreen = request.getParameter(“ScreenName”);

        if (”Billing”.equals(nextScreen)) {
            disp = getServletConfig().getServletContext().
                    getRequestDispatcher(”Billing.jsp”);
        } else if (”Shipping”.equals(nextScreen)) {
            disp = getServletConfig().getServletContext().
                    getRequestDispatcher(”Shipping.jsp”);
        }

        ...

        if (disp != null) {
            disp.forward(request, response);
        }
    }
}

When the application comes up, one more class should be instantiated and populated – the ProductCatalog. This class connects to the database, and retrieves and stores all available products. If the number of products is too
big or you’d like to show the actual data from the database, just re-load the
ProductCatalog class every time the user opens the
ProductCatalog.jsp.

Complete code of a simple online store application is included in the book’s
source code.
Lesson 17

RMI And JNDI

Remote Method Invocation

Remote Method Invocation (RMI) allows data exchange between different Java programs running under different JVMs. These JVMs could be located on the same or separate computers, but the most important feature is that one Java class can invoke methods belonging to an object that lives in another (remote) JVM. This enables applications to call methods remotely.

In most of the books RMI is explained in the chapters on network programming, but I decided to move it closer to the EJB lessons because they share many similar ideas. RMI applications are simpler than the EJB one(s) and they do not need any application servers - regular JVMs are sufficient.

Any RMI application consists of an RMI server, client, and the registry (the naming service). These three components could run on three different networked computers. The server creates some objects, registers them with the naming service, and waits for remote clients to invoke some methods on these objects. A client application gets a reference to a remote server object(s) from the registry and then invokes methods on these remote objects. RMI provides the mechanism for remote Java programs to communicate when more than one JVM is involved. The main concept of RMI is that even though the methods are being called in the client’s JVM, they are executed on the server’s one!

RMI classes and the registry tool are come with the JSDK which means that application servers are not required.

Developing Applications With RMI

Writing distributed RMI applications involves the following steps:

- Declaring the remote interface.
• Implementing the remote interface.

• Writing the client that connects to remote server and calls remote methods.

• Generating of the stubs (client proxies) and skeletons (server entities).

• Starting the registry and registering the RMI server with it.

• Starting the server and the client applications.

Let’s look at each step by developing the RMI version of the Stock Quotes Server that will provide a client with the price quotes for a specified stock. It could be used for by various clients such as an online users, other stock trading systems, etc.

Defining Remote Interfaces

A remote interface defines method(s) that can be invoked remotely by a client. The client will “have a feeling” that they call local methods, but these calls will be redirected to a remote server via RMI protocol. These are the rules for creating remote interfaces:

• The remote interface must declare public methods to allow clients remote method invocation.

• The remote interface must extend the java.rmi.Remote interface.

• Each method must declare a java.rmi.RemoteException.

• Method arguments and return data types have to be serializable.

Below is a code of the StockServer interface that will be used on the client side. This interface defines two business methods - getQuote() and getNasdaqSymbols():

```java
import java.rmi.*;
import java.util.Vector;

public interface StockServer extends java.rmi.Remote {
```
public String getQuote(String symbol) throws java.rmi.RemoteException;

public Vector getNasdaqSymbols() throws java.rmi.RemoteException;

}

Implementing Remote Interfaces

While the remote interface just declares the methods, the actual class that provides implementation for these methods will run on the server side. This class has to extend the java.rmi.server.UnicastRemoteObject.

Here is the class StockServerImpl that will process the client's requests.

import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;

public class StockServerImpl extends UnicastRemoteObject implements StockServer {
    private String price=null;
    private Vector nasdaqSymbols = new Vector();

    public StockServerImpl() throws RemoteException {
        super();

        // Define some hard-coded NASDAQ symbols
        nasdaqSymbols.addElement("PRSW");
        nasdaqSymbols.addElement("MSFT");
        nasdaqSymbols.addElement("YHOO");
        nasdaqSymbols.addElement("AMZN");
        nasdaqSymbols.addElement("SUNW");
    }

    public String getQuote(String symbol) throws RemoteException {
        if(nasdaqSymbols.indexOf(symbol.toUpperCase()) != -1) {
            // Generate a random price for valid symbols
            price = (new Double(Math.random()*100)).toString();
        }

        return price;
    }
}
public Vector getNasdaqSymbols() throws RemoteException {
    return nasdaqSymbols;
}

Registering Remote Objects

To make a remote object available to clients we will bind it to a registry which nothing else but a naming service that knows where exactly in the network the server is running. This will allow clients to look up the object on the host machine by name.

import java.rmi.*;
import java.rmi.registry.LocateRegistry;

public class StartServer {
    public static void main(String args[]) {
        try {
            StockServerImpl ssi = new StockServerImpl();
            Naming.rebind("rmi://localhost:1099/QuoteService", ssi);
            System.out.println("<QuoteService> server is ready.");
        } catch (Exception ex) {ex.printStackTrace();}
    }
}

There are two methods in the class java.rmi.Naming that can bind an object in the registry. The method bind() binds an object to a name. It throws the AlreadyBoundException if the binding already exists. The method rebind() replaces any pre-existing binding with the new one.

To bind the objects to the registry it must running. One way to start the registry is by typing the following in the command window:

c:\practice>rmiregistry

Instead of starting the registry manually, you could have also started it from within the StartServer program itself:
Writing RMI Clients

The client performs a lookup in the registry on the host machine and obtains a reference to the remote object. Please notice the casting from of the lookup() return to the `StockServer` type. Even though the class `StockServerImpl` has been bound to the name `QuoteService`, since this class implements the `StockServer` interface, we can cast the returned object to it. The variable `myServer` “will see” only the methods defined in this interface, while the class `StockServerImpl` many have other public methods.

```java
import java.rmi.*;
import java.util.Vector;

public class Client {
    public static void main (String args[]) {
        if (args.length == 0) {
            System.out.println("\nUsage: java "+
                -Djava.security.policy=security.policy Client "+
                symbols.toString());
            System.exit(0);
        }
        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            StockServer myServer = (StockServer)
                Naming.lookup("rmi://localhost:1099/QuoteService");
            String price = myServer.getQuote(args[0]);
            if (price != null){
                System.out.println("The price of " + args[0] +
                    " is:" + price);
            }
            else{
                System.out.println("Invalid Nasdaq symbol. " +
                    "Please use one of these:" +
                    myServer.getNasdaqSymbols().toString());
            }
        }
    }
}
```
The class `java.rmi.RMISecurityManager` extends the class `java.lang.SecurityManager` and provides security context under which RMI application executes. If no security manager has been set, the stub classes (see below) can only be loaded based on the local `CLASSPATH`. This protects applications from downloading unsecured code via remote method invocation.

**Finding Remote Objects**

Clients find remote services by using a naming or directory service. A naming service runs on a known host and port number.

As I mentioned earlier, an object can start its own registry that performs the naming services for the RMI clients. The behavior of the registry is defined by the interface `java.rmi.registry.Registry`.

The RMI registry runs by default on the port 1099, unless the other port number is specified. When the client wants to invoke methods on the remote object, it obtains a reference to it by looking up the name. The lookup returns to the client a remote reference called `stub`.

The method `lookup()` takes the object name URL as an argument in the following format:

```
   rmi://<host_name>[:<name_service_port>]/<service_name>
```

- The `host_name` is a name of the computer on the local area network (LAN) or a DNS name on the Internet.
- The `name_service_port` has to be specified only if the naming service is running on a port other than the default one.
- The `service_name` is the name of the remote object that should have been bound to the registry.
Stubs and Skeletons

The stub is a client-side object that represents the remote object. The stub has the same interface, or list of methods, as the remote object, but when the client calls a stub method, the stub forwards the request via the RMI infrastructure to the remote object, which actually executes it. The stubs simplify the client's code by hiding the network communications and serialization of method parameters (they have to be serializable).

On the server side, the skeleton object processes the network call, deserializes received object and executes the requested method.

RMI compiler rmic generates stubs and skeletons from existing class. Start it from the command window (go to your work directory first) as follows:

c:\practice>rmic StockServerImpl

The above command will create two additional classes having suffixes _stub and _skel respectively.
Starting the Server and the Client

To run the RMI server you should open one more command window and start the `StartServer` class from your working directory:

`c:\practice>java StartServer`

To run the client you need to open the third command window and run the `Client` class specifying our security policy file:

`c:\practice>java -Djava.security.policy=security.policy Client SUNW`

Policy files contain permissions granted to users of this application. You could find detail explanations of how to write security policy files at the following Internet site:

`http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html`

Setting Up the Stock Quote Server Application

Below are the steps that need to be done to successfully install and run our RMI stock server:

Step 1. Create the interface `StockServer` that extends `java.rmi.Remote` interface.

Step 2. Create the server class `StockServerImpl` that extends `UnicastRemoteObject` and implements the interface from the Step 1.

Step 3. Create the class `StartServer` to start the server from the Step 2 and bind it to the naming service (rmiregistry).

Step 4. Create the class `Client` that will access the remote server from the Step 2.

Step 5. Save all above classes in one folder, say `c:\practice` and compile them:

`c:\practice>javac *.java`
Step 6. Create stub and skeleton classes from the `StockServerImpl` using the `rmic` compiler:

```
c:\practice>rmic StockServerImpl
```

Step 7. Open three command windows and get into directory where all your classes are:

```
c:\practice>cd practice
```

Step 8. Start the RMI registry from the first command window. The naming service will listen to a default port 1099:

```
c:\practice>rmiregistry
```

Step 9. Register your `StockServer` with your naming service from the second command window:

```
c:\rmistock>java StartServer
```

Step 10. Run the `Client` from the third command window. Pass the stock symbol as a command line argument and the client will connect to our “remote” server and receive the price quote:

```
c:\practice>java -Djava.security.policy = security.policy Client SUNW
```

Make sure that the file `security.policy` exists in the same directory where the `Client.class` is located.
Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) is designed to simplify finding objects in a distributed applications. It plays the role similar to the telephone company directory assistance service. Various software vendors could create “directory assistance” software, but JNDI is not one of them – it just provides a standard API to access this kind of software.

You may already had a chance to use different APIs to access directory services such as file system on your PC, LDAP (Lightweight Directory Access Protocol) or Sun’s NIS (Network Information Service).

Naming and Directory Services

A Naming Service allows you to add, change, or delete the names of objects that exist in some naming tree. It provides a unique name for every entry that is registered (bound) with the service. Every naming service will have one or more contexts (similar to a concept of directories/sub-directories in a file system). The naming tree originates from the so called initial context (similar to a root directory on the disk).

A Directory Service allows you to search the naming tree by object attributes rather then the object names. One of the Internet examples is its Domain Naming Service (DNS), which takes a Domain Name and returns IP address.

There has to be a process that initially binds the objects to a naming tree to make the directory service available. This could be an independent program that binds, say employee names to a directory server of some firm. Similar job is usually done by application servers that binds such objects as EJB, Servlets, etc. to their naming services during the startup process.

Let’s consider an example that shows how to obtain and use the WebLogic server’s naming context. JNDI classes are located in the javax.naming package inside of the jndi.jar that is installed with J2EE or is provided by the application server vendor.
import java.util.*;
import java.io.*;
import javax.naming.*;

A client program JndiExample creates an instance of the InitialContext class and passes properties of the naming server using a Hashtable.

• java.naming.provider.url property value should specify location of the server that supports JNDI. In case of WebLogic server the default URL is t3://localhost:7001.

• java.naming.factory.initial is used to specify how the initial context will be created.

Application server vendors will provide their own factory classes. The class WLInitialContextFactory is specific to WebLogic application server.

The values of provider and the factory have to be set to the PROVIDER_URL and INITIAL_CONTEXT_FACTORY variables of the class Context.

public class JndiExample {
    public static void main(String[] args) {
        Context ctx = null;
        try {
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                     "weblogic.jndi.WLInitialContextFactory");
            env.put(Context.PROVIDER_URL, "t3://localhost:7001");

            ctx = new InitialContext(env);

            System.out.println("WebLogic initial context created");
        } catch(Exception e){...}
    }
}

If you need to access the Context not form a remote client, but from an object located inside the application server (for example a from a bean), write the following code:

ctx = new InitialContext();
The next example performs the object binding and lookup in the same program. In the real life these actions are done in different programs.

```java
String bindObject = new String("1965, 203 Main Street, New York, NY 10001");
String bindName = "JohnSmith";

try {
    ctx.bind(bindName, bindObject);
    System.out.println("Bound " + bindObject + " to "
                        + bindName);
} catch (NameAlreadyBoundException e) {
    System.out.println("Rebinding " + bindObject+
                        " to " + bindName);
    ctx.rebind(bindName, bindObject);
}

try{
    String foundObject = (String)ctx.lookup(bindName);

    if (foundObject.equals(bindObject)) {
        System.out.println("Found " + foundObject);
    } else {
        System.out.println("Can't find the object" + bindName);
    }
}catch (NamingException e) {
    System.out.println(e.toString());
}

finally {
    if (ctx != null) {
        try {
            ctx.close();
        } catch (NamingException e) {
            System.out.println("Can't close the context:" + e);
        }
    }
}
```
Setting Up the JNDI Example

The source code of the JNDI example could be found in the file JndiExample.java. Perform the following steps to test it:

Step 1. Download and install the evaluation copy of Weblogic 6.1 application server (see appendix B).

Step 2. Open one command window and get to the Weblogic's application folder, i.e. c:\bea\wlsserver6.1\config\mydomain.

Step 3. Run the following command files to set WebLogic environment variables and start the server:
   c:\bea\wlsserver6.1\config\mydomain>setEnv
   c:\bea\wlsserver6.1\config\mydomain>startWebLogic

Step 4. Open the second command window in the same directory and run the following command:
   c:\bea\wlsserver6.1\config\mydomain>setEnv

Step 5. Switch to your work directory compile and run the JndiExample:
   c:\practice>javac JndiExample.java
   c:\practice>java JndiExample

Note. Open the WebLogic's console by pointing your Web browser at http://localhost:7001/console/, select Servers | myserver on the left panel and click on the Monitoring tab on the right. Follow the View JNDI Tree Link to see bound objects.

Assignment.
Create two programs: one should bind some objects to JNDI naming tree, and the other one should be able to find these objects by name.
Lightweight Directory Access Protocol

LDAP servers are highly optimized for reading. This makes them a good choice for such directory services as Employee List or Phone directory. JNDI for LDAP servers plays the same role as JDBC plays for DBMS.

The popular LDAP servers are Netscape Directory Server and Microsoft Active Directory. The LDAP tree has a root entry which consists of one or more distinguished names. On the top of the hierarchy you place the object with the prefix o - organization. One level below goes ou - organizational unit, cn stands for common name, etc.

As opposed to other naming services, the search string starts from the very lowest hierarchical entry and the root entry has to be specified the last, for example.

cn=jsmith, ou=accounting, o=smartdataprocessing.com

Below is a code fragment that prepares JNDI properties, connects to LDAP server, and find the object called CustomerHome.

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=Directory Manager");
env.put(Context.SECURITY_CREDENTIALS, "myPassword");

DirContext ctx = new InitialDirContext(env);
CustomerHome custHome = (CustomerHome) ctx.lookup("cn=CustomerHome, ou=BigProject, o=sdp.com");

For training purposes you can install and run all examples from this book on a single computer, but a real world distributed application can be designed and operate as follows:

• Computer #1 runs the LDAP server.

• Computer #2 runs an application server that had registered (published) some objects with the LDAP server on Computer #1.
• Computer #3 has a Client program that finds object references on the Computer #1 and invokes their methods on the Computer #2.

• Computer #4 has a database management system which is being used by the application server running on the Computer #2.

• Computer #5 publishes Market Data and Computer #2 subscribes to this service.

• ...and on, and on and on...

Resources

1. JNDI Tutorial:
   http://java.sun.com/products/jndi/tutorial

2. RMI documentation:
   http://java.sun.com/products/jdk/rmi/

3. RMI frequently asked questions:
   http://java.sun.com/j2se/1.3/docs/guide/rmi/faq.html

4. Tips for LDAP users:
Lesson 18

Enterprise Java Beans

Enterprise Java Beans (EJB) is yet another way of creating multi-tier applications that could be run in a distributed environment. So what? Isn't the same statement correct for the JSP/Servlet applications? It's true, but first of all JSP and servlets are good for the Web based application only, while EJB architecture does not have this limitation and has a lot of additional benefits:

- EJB application have better scalability.
- The same EJB components could be deployed in application servers from different vendors (in a real life it may not be as easy as it sounds).
- Properly designed applications could improve performance a lot by using cached objects, clusters and failover features of application servers.
- The same components could be reused by various types of clients (applets, servlets, JSP, standalone Java applications, other EJB running on the same or different application server, C++ programs through Corba interface and others.
- Some vendors provide tools to create database application without writing any SQL at all (i.e. PowerTier application server). This also means that switch to a different DBMS vendor (say from Oracle to Sybase) comes down to a simple modification of connection parameters in a properties file.
- EJB containers take care of object pooling minimizing time spent on Java garbage collection.

If EJB are so good, why not every application is developed this way? Here's why:

- This architecture is more difficult to learn than the servlets and JSP technologies. Because of this some people are trying to find any reason for not using EJB.
• The need to hire more senior developers increases the development cost, which could be an important consideration for project managers.

• The EJB servers are more expensive than JSP/servlet engines.

Application Server And EJB Container

Java application servers contain various components such as Servlet/JSP container (engine), EJB container, Web server, and others. They provide clustering, fail-over and load balancing services for EJB containers. If an application server vendor states that their product supports J2EE version 1.3, you can figure out what functionality to expect from it.

An EJB container provides a living space for beans, arranges bean pooling, multithreading, bean creation, removal and some other services. To find out what services are available in a particular application server, you need to know what version of EJB specification is supported.

Beans Types

Starting from EJB specification 2.0 there are 3 types of beans:

• **Session beans** - a place to program business logic of the application. These do not survive server crashes because they exist only in memory. Session beans could be either stateless or stateful.
  
  **Stateless** beans do not remember the current state of any particular client – they are just classes containing methods with business logic.

  **Stateful** beans store in their instance variables data of a particular client.

• **Entity beans** represent persistent data that are typically stored in a database. One instance of an entity bean corresponds to a row from a database table, or a result set returned by an SQL statement or a stored procedure. They can survive server crashes because the data are stored not in memory, but on the disk (database, file, etc).

• **Message-Driven beans** support Java Messaging Service (JMS) and are used for retrieval of messages from queues. These beans are available only in application servers that support EJB specification version 2.0 (for example WebLogic 6.1).
Stateless Session Beans

**Question:** How many classes should I write to print “Hello World” using EJB?

**Answer:** Four. You have to create one client class and 3 EJB classes:

- **Home** interface, **Remote** interface and the **Bean** itself. To make you laugh even harder let me tell you this: the application server will use these 3 classes to generate several additional classes that will provide implementation for your home and remote interfaces and support for the RMI/IIOP protocol relieving you from writing stubs, skeletons and other networking stuff. The good news is that EJB architecture is not being used for the hello-world type applications.

This is how a typical EJB application works:

- An EJB client finds the bean’s home interface on the server.
- The Home interface obtains a reference to the bean’s remote interface.
- A client calls business methods defined in the remote interface.

This scenario is very similar to what’s happening between the RMI client and server. Let’s compare these two technologies:

<table>
<thead>
<tr>
<th>RMI</th>
<th>EJB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A program starts the RMI server and registers it with running RMI registry</td>
<td>1. An application server starts and registers Home interfaces of the beans to a naming server.</td>
</tr>
<tr>
<td>2. A client connects to the naming server (RMI registry), finds the RMI server using the Naming.lookup() and gets a reference to its remote interface.</td>
<td>2. A client connects to the naming server, finds the Home interface of the bean using JNDI method lookup().</td>
</tr>
<tr>
<td>3. A client calls business methods defined in the remote interface of the RMI server</td>
<td>4. A client calls business methods defined in the bean’s remote interface.</td>
</tr>
</tbody>
</table>
You can find the EJB client’s code sample in this lesson’s section “Writing EJB Clients”.

Home Interface

Home interface class contains methods for creating, removing and finding the bean instance.

Let’s start creating a bean that can process a stock trading order. This bean will contain only the order processing business logic that is the same for every client. Since it won’t represent any particular order we’ll create a **stateless** session bean. Please pay attention to the way we name the bean’s classes: TradeOrderHome, TradeOrder and TradeOrderBean. In all of the examples below we’ll be using package names assuming that the project named Stock Trading System (sts) is being developed in the company that has a website sdp.com. To better organize our classes we’ll create the following disk directory structure:

```plaintext
com
  sdp
    sts
      beans
        session
        entity
        mdb
      util
    client
```

Here’s the code of the “bean’s Home” that must be a subclass of the EJBJHome interface.
The TradeOrderHome is a Java interface that’s why it contains only method signatures – the implementation for this interface will be generated by the tools provided by the application server's vendor.

At a minimum, every Home interface must include the signature of the method create().

Since the clients of the bean will be running outside of the EJB container they might get a CreateException or RemoteException, if for any reason the remote interface could not be created or some other error occurred.

Remote Interface

Bean developer has to add signatures of the public business methods to the Remote interface of the bean which has to be a subclass of the EJBObject. Client programs call business methods on the Remote interface as if these the bean is located in the client's JVM. But the actual bean lives and works within the EJB container on the server’s side and should never be directly accessed by a client – a Remote interface is the middleman.

The remote interface TradeOrder contains the signatures of 3 business methods: validateAccount(), createOrder() and getOpenOrders(). The application server provides tools for generation of additional classes which take care of networking, serialization and marshalling processes to support these business methods.
import javax.ejb.*;
import java.rmi.*;
import java.util.*;

public interface TradeOrder extends EJBObject {
    public double validateAccount (String id, double amount)
            throws RemoteException;

    public Savings createOrder (String symbol, Integer quantity, Double price)
            throws RemoteException;

    public Vector getOpenOrders() throws RemoteException;
}

Please Welcome - Mr.Bean

The third singer in the trio is the bean itself that contains implementation of the business methods defined in the Remote interface, and callback methods providing the life support of the bean - ejbCreate(), ejbRemove(), etc. When a client calls the method create() on the Home interface, the method ejbCreate() is being called on the bean. When a client calls the method validateAccount() on the Remote interface, the method validateAccount() is being called on the bean.

Since we create a session bean, the class must implement the interface SessionBean, which contains callback methods that will be used by the EJB container.

Please find the comment lines “Your code goes here” in the example below – these are the places where developers should write code that implements business logic of the application.

package com.sdp.sts.beans.session;

import javax.ejb.*;
import javax.naming.*;
import javax.transaction.UserTransaction;
import java.rmi.*;
import java.util.*;

public class TradeOrderBean implements SessionBean {

    // Instance variables declarations, if any, goes here.
public boolean validateAccount (String id, double amount) 
    throws RemoteException {
    // Your code goes here
}

public Savings createOrder (String symbol, Integer quantity, Double price) throws RemoteException {
    // Your code goes here, for example:
    if (price < 5) {
        return "The chances are slim that " + symbol + " will be traded below $5.00";
    } else {
        return "The order to buy " + quantity + " shares of " + symbol + " has been placed.";
    }
}

public Vector getOpenOrders() throws RemoteException {
    // Your code goes here
}

// You may also define some private methods for
// the internal bean’s use:

private double calcMarginLimit() { ...}

// Callback methods supporting the life cycle of
// this
// bean. We’ll leave them empty for now.

public void ejbCreate() { }
public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbRemove() { }
public void setSessionContext (SessionContext ctx) 
    throws RemoteException {
    this.ctx = ctx;
}


Writing EJB Clients

An EJB could be used from different types of clients:

- Any Java application (an applet or independent Java program)
Another enterprise bean

A Java Servlet

A Java Server Page

A bridge from a non-Java application. For example, IBM's WebSphere 4.0 comes with the ActiveX Bridge Service that allows Visual Basic clients access EJBs.

To work with a bean any client needs to perform the following steps:

Step 1. Obtain a reference to the Home interface of a bean using JNDI lookup.

Step 2. Call the method `create()` or `find()` on the Home interface to get a reference to a Remote interface.


A sample client below calls the method `createOrder()` of the `TradeOrder` session bean passing the hardcoded values for the stock symbol, quantity and price.

```java
import java.rmi.*;
import javax.ejb.*;
import javax.naming.*;
import java.util.*;

public class Client{
    public static void main(String args[]){
        InitialContext ctx;
        try{
            // JNDI initialization
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "weblogic.jndi.WLInitialContextFactory");
            env.put(Context.PROVIDER_URL, "t3://localhost:7001");
            ctx = new InitialContext(env);
            System.out.println("WebLogic initial context has been created");

            // finding the home
            TradeOrderHome toHome = (TradeOrderHome) ctx.lookup("TradeOrderHome");
            // your business logic here

        }
    }
}
```
// creating remote interface
TradeOrder tradeOrder = toHome.create();

// calling one of the business methods of the bean
System.out.println("Placing order...");
String result = tradeOrder.createOrder("IBM", 100, 86.5);
System.out.println("The bean responded:" + result);

} catch (NamingException e) {
    System.err.println("Could not find TradeOrderHome:");
    e.toString());
} catch (CreateException e1) {
    System.err.println("Could not create TradeOrder:");
    e1.toString());
} catch (RemoteException e2) {
    System.err.println("Error during ...");
    e2.toString());
}

EJB Deployment Descriptors

One of the main selling points of the EJB is that the application could be fine-tuned by playing with properties as opposed to performing code modifications. The Bean's properties are specified in so called deployment descriptors which are xml files. To deploy a bean the file ejb-jar.xml must be provided for any EJB container. Additional xml files could be used with vendor-specific properties, for example weblogic-ejb-jar.xml. At this point we'll create a minimal deployment descriptor file.

Here's an example of the mandatory ejb-jar.xml:

```xml
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN" 'http://java.sun.com/dtd.ejb-jar_2_0.dtd'>
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>TradeBean</ejb-name>
      <home>TradeOrderHome</home>
      <remote>TradeOrder</remote>
      <ejb-class>TradeOrderBean</ejb-class>
      <session-type>Stateless</session-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```
This is WebLogic specific `weblogic-ejb-jar.xml`:

```xml
<?xml version="1.0"?>

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>TradeBean</ejb-name>
    <jndi-name>TradeStateless</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

### Testing TradeOrder Session Bean in WebLogic

Step 1. Open two command windows and get into the Weblogic's domain folder in each of them, i.e. `c:\bea\wlserver6.1\config\mydomain`.

Step 2. Run the command file `setEnv` in both windows to set the server's environment variables:

```
c:\bea\wlserver6.1\config\mydomain>setEnv
```

Step 3. Enter the code of the Client and three bean's classes - `TradeOrderHome`, `TradeOrder` and `TradeOrderBean` and save them in the directory `practice`.

Step 4. In the first command window run provided command files to build and copy the jars into the deployment directory:

```
c:\practice >build
```
```
c:\practice >deploy
```
Step 5. Start the WebLogic server in the second command window. At this point it’ll find and deploy the bean from the jar:

`c:\bea\wlserver6.1\config\mydomain>startWebLogic`

Step 6. In the first command window run the command file to start the client application:

`c:\practice>run`

Open the WebLogic’s console and see if the TradeOrder bean has been published to the JNDI tree.

**Stock Trading System Specification**

Specifications for a system development could be given to you by using such tools as a pencil and a napkin in your cafeteria, Unified Modeling Language (UML) diagram, a conversation between you and your manager, etc. Let’s put a short description of the functionality and database design on a napkin:

Proposed functionality of the Stock Trading System (STS):

- The users (customers) should be able to logon to a system.
- A customer should be able to have several Accounts.
- A customer should be able to have several stock portfolios.
- A customer should be able to place a buy/sell order.
- A customer enters the stock symbol, quantity, order type (Market or Limit) and the action: Buy, Sell or Cancel.
- Only the Limit orders that were not sent for execution could be cancelled.
- All transactions have to be saved in a database.
- The Order and Execution modules should communicate using a messaging service.
- All manipulations with Customer/Order records have to be stored in the database tables Transactions and History.
- Customer's account could have an Active or a Frozen status.
A customer should be able to see the content of his/her portfolio and the status of the placed orders.

The front end design has a low priority, since another group will take care of that. For now we just need a simple Java client to be able to test all features of the system.

The phase two of the system may include e-mail confirmations and reporting.

If you decide to create a fancy client GUI, look at the existing online trading systems, such as Datek (www.datek.com), E*Trade (www.etrade.com) or others. Datek has a Streamer program that provides real time stock quotes. See if you can create something similar using Applet-Servlet communication or RMI.

Stock Trading System Database

Below are the scripts to create Oracle database tables for the Stock Trading System. If you decide to use different database, find a replacement for the Oracle sequence feature that allows automatic generation of unique values, for example identity column in Sybase. These numbers will be used as primary keys in the database tables. The following two scripts are provided to create and populate the tables.

tablesCreate.sql:

drop table Users cascade constraints;
CREATE TABLE Users (  
    userID     NUMBER  NOT NULL,  
    username    VARCHAR2(10) NOT NULL,  
    password    VARCHAR2(10) NOT NULL,  
    role        VARCHAR2(10) NOT NULL CHECK (role IN  
                   ('Trader', 'TRDManager', 'Customer', 'Admin')),  
    PRIMARY KEY (userID)
);
DROP SEQUENCE user_seq;
CREATE SEQUENCE user_seq;

drop table Customer cascade constraints;
CREATE TABLE Customer (  
    customerID      NUMBER NOT NULL,  
    lastname   VARCHAR2(20) NOT NULL,  
    firstname  VARCHAR2(20) NOT NULL,  
    PRIMARY KEY (customerID)
);
street   VARCHAR2(30) NOT NULL,
city     VARCHAR2(30) NOT NULL,
state    VARCHAR2(2) NOT NULL,
zip      NUMBER(5) NOT NULL,
country  VARCHAR2(10) NOT NULL,
phone    CHAR(12)  NOT NULL,
fax      CHAR(12)  NOT NULL,
email    VARCHAR2(40) NOT NULL,
 PRIMARY KEY (customerID)
);
DROP SEQUENCE customer_seq;
CREATE SEQUENCE customer_seq;

drop table Account cascade constraints;
CREATE TABLE Account ( 
accountID      NUMBER NOT NULL,
customerID     NUMBER NOT NULL,
type           VARCHAR2(4) NOT NULL CHECK (type IN ('REGL',
'MARG','MRKT'))),
status          VARCHAR2(6) NOT NULL CHECK (status IN
('ACTIVE', 'FROZEN')),
balance         NUMBER NOT NULL,
 PRIMARY KEY (accountID),
FOREIGN KEY (customerID) references Customer
);
DROP SEQUENCE account_seq;
CREATE SEQUENCE account_seq;

drop table Orders cascade constraints;
CREATE TABLE Orders ( 
orderID         NUMBER NOT NULL,
accountID       NUMBER NOT NULL,
symbol          VARCHAR2(5) NOT NULL,
quantity        NUMBER NOT NULL,
orderType       VARCHAR2(1) NOT NULL CHECK (orderType IN
('B', 'S')),
priceType       VARCHAR(5) NOT NULL CHECK (priceType IN
('MRKT', 'LIMIT')),
price           NUMBER NOT NULL,
creationDate    DATE  NOT NULL,
statusOrder     VARCHAR2(4) NOT NULL CHECK (statusOrder IN
('OPEN', 'PART', 'FILL')),
trackingNumber  VARCHAR(10)  NOT NULL,
 PRIMARY KEY (orderID),
FOREIGN KEY (accountID) references Account
);
DROP SEQUENCE order_seq;
CREATE SEQUENCE order_seq;

drop table Transactions cascade constraints;
CREATE TABLE Transactions (  
    transactionID NUMBER NOT NULL,
    orderID NUMBER NOT NULL,
    type VARCHAR2(3) NOT NULL CHECK (type IN ('NEW', 'MOD','CAN','EXE')),
    quantity NUMBER NOT NULL,
    price NUMBER NOT NULL,
    brokerID NUMBER NOT NULL,
    execTime DATE,
    PRIMARY KEY (transactionID),
    FOREIGN KEY (orderID) references Orders
);
DROP SEQUENCE transaction_seq;
CREATE SEQUENCE transaction_seq;

TablesData.sql:

insert into Users  values   (1,'rob','rob','Admin');
insert into Users values    (2,'john','john','Customer');
insert into Customer  values  (1,'Brown','Rob','123 Main St.','New York',NY,10001,'USA', '316-636-5555','316-636-5554','brob@yahoo.com');
insert into Customer  values   (2,'Charles','John','237 Ash Avenue','Manapalan','NJ',07726,'USA', '316-689-5555',
                                  '316-689-5554','jcharles@mail.ru');
insert into Account  values   (1,2,'MARG','ACTIVE',1000.00);
insert into Account  values   (2,2,'REGL','ACTIVE',555.60);
insert into Account  values  (3,2,'MRKT','FROZEN',0.12);
insert into Account  values   (4,1,'MARG','ACTIVE',999.00);
insert into Account  values   (5,1,'REGL','ACTIVE',55.10);

Resources

1. EJB Tutorial:  

2. EJB specification version 2.0:  
   http://java.sun.com/products/ejb/2.0.html

3. EJB Technologies Fundamentals by jGuru:  
   http://developer.java.sun.com/developer/onlineTraining/EJBIntro/
Lesson 19

Session Beans (Cont.)

Stateful Session Beans

In a regular Java class you can call several methods in row and store the returns of each method in the member variables of the class - they hold “the state of the class”. Since stateless session beans do not guarantee that the methods called by a client will be processed by the same exact bean – the instance variables have very limited use in stateless beans.

If the business processing consists of several method calls and their results have to be available for the client for some period of time – use stateful session beans which allow to hold client specific data in the bean. The bad news is that it’s much more expensive to keep one bean instance per client than to pick any available stateless bean from a pool.

The code of stateful and stateless session beans looks the same: both implement the same interfaces and include the same callback methods. As a matter of fact, the EJB container knows what type of a bean is it only after reading its properties from the deployment descriptor file.

Lifecycle of a Stateful Session Bean

A client application works with the stateful bean the same way as with the stateless one: it performs the JNDI lookup of its home, creates the remote interface and calls its business methods.

Since the instance of a stateful bean “belongs” to a client (i.e. shopping cart), it remembers the state of the current session (i.e. the goods that were added to the shopping cart).

Here’s the sequence of method calls on a typical stateful session bean:

- The client calls a method create() on the home interface.
- The EJB container calls the method setSessionContext() on the bean.
• The EJB container calls bean’s method `ejbCreate()` that has the same signature as the method `create()`.

• **The client** calls one or more business methods on the same instance of the bean.

• The EJB container might call the method `ejbPassivate()` to free up some memory.

• The EJB container might call the method `ejbActivate()`, if the bean has been passivated earlier.

• **The client** calls the method `remove()` when the bean is not needed anymore.

• The EJB container calls the method `ejbRemove()` on the bean.

**Creating Stateful Beans**

Stateful beans may have overloaded methods `create()` with different arguments and each method `create()` on the remote interface must have the matching `ejbCreate()` on the bean.

You may write code in the method `ejbCreate()` – something that you would put in the constructor of a regular Java class – initialization of instance variables, creation of connections to some outside systems, etc. The sample code below shows the method `ejbCreate()` that takes the instance of some utility class `Customer` as an argument.

```java
public class OrderService implements SessionBean{
    int cust id;
    float balance;
    MyJMSConnection connection;

    public void ejbCreate( Customer cust)
        throws SPIException{

        // initialization of the member variables
        custID= cust.getID();
        balance = cust.getBalance();
    }
}
```
// Creation of additional beans
Context ctx = new InitialContext();

// Connect to some outside system
connection = new MyJMSConnection();
}
}

class Customer implements Serializable{
    private int id;
    private double balance;
    public int getID(){ return id;}
    public double getBalance(){return balance;}
    public void setID(int id){this.id=id;}
    public void setBalance(double balance) {
        this.balance=balance;
    }
}

In the world of distributed applications it's better to pass multiple values to
the bean in one shot by using utility classes than making several remote
calls of the setter methods. Our class Customer is being used for this
purpose and represents a design pattern called Value Object. Each remote
call generates network traffic, that's why it's better to pre-populate the
instance of a class Customer and make one remote call like this:

myBean.setCustomer(myCustomer);// the coarse-grained method

than multiple remote calls like these:

myBean.setCustID(1234);       // the fine-grained methods
myBean.setCustBalance(500.00);

Bean Activation, Passivation And Removal

Passivation is a process of temporary removal of the bean from memory.
This is done by the EJB container to free up some room for other beans, if
needed.

Activation is a process of loading the previously passivated bean back into
container's memory.
When the EJB container decides to passivate a stateful session bean, Java serialization routines are being used to persist its state in a safe place (usually on a disk). Later on, when this bean will be activated again, all its variables will be de-serialized by the container.

You should always release the resources that can be easily re-created, i.e. connection variable from the class OrderService above. The following code fragments demonstrate the use of methods ejbPassivate(), ejbActivate() and ejbRemove() that could be used in our stateful session bean:

```java
public void ejbPassivate(){
    if (connection != null){
        connection.close();
    }
}

public void ejbActivate(){
    if (connection == null){
        connection = new MyJMSConnection();
    }
}

public void ejbRemove(){
    if (connection != null){
        connection.close();
    }
}
```

Bean’s Session Context

An EJB container passes the information about the operational environment to the bean by calling its method setSessionContext():

```java
public class OrderService implements SessionBean{
    SessionContext sessionContext;
    ...

    public void setSessionContext(SessionContext sc) {
        this.sessionContext = sc;
    }
    ...
}
```
One of the common uses of the session context is getting a reference to the class that provides transaction processing: UserTransaction:

```java
class MyBean implements SessionBean{
    public void myMethod(){
        UserTransaction txn =
            sessionContext.getUserTransaction();
        txn.begin();
        // your SQL, entity bean manipulations, JMS, or other 
        // code goes here
        txn.commit();
    }
}
```

Only session beans with **bean-managed** transactions (see below) can use the interface UserTransaction that provides an access to Java Transactional Services and allows to treat multi-database operations as one transaction (so called two-phase commit). The two-phase commit is available in application servers which support J2EE specification 1.3 or above.

The SessionContext also comes handy when you need to pass a reference to this bean’s instance to another bean, for example:

```java
public void myMethod(){
    //getEJBObject() returns a reference to the bean’s instance
    myBean.ValidateOrder(context.getEJBObject());
}
```

**Transaction Attributes**

Each session bean has its transaction attribute defines in the deployment descriptor. These are the valid transaction attributes used with container-managed transactions:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never
Session beans could create other session beans which also have transaction attributes. It’s easier to understand them using example of two-bean interaction, say a BeanA calls a method on a BeanB and some transactional operations could occur in each of them.

**Required:**
If the BeanA is already a part of transaction, the BeanB does not create a new one, but uses existing transaction. If the BeanA is not in any transaction, the BeanB will create one.

**RequiresNew:**
This one ensures that the BeanB will always run in its own transaction, but commits and rollbacks should not be propagated to the calling BeanA. This mode is a little slower than the previous one.

**Mandatory:**
This mode ensures that the BeanB runs in the transaction of the client which is the BeanA in our example. If the client did not have an open transaction, the BeanB would throw the exception TransactionRequiredException.

**NotSupported:**
Use it to ensure that the BeanB is never a part of any transaction and its possible failures will not affect the BeanA. For example, if the BeanB just logs informational messages of the application, we could afford to ignore possible logging errors.

**Supports:**
The BeanB does not care about transactions - if there is an open transaction in the BeanA, it’ll support it, if not – it’s also fine with the BeanB. This is the fastest mode – there is no transaction related overhead in the BeanB.

**Never:**
Use it if you want to ensure that the BeanB never runs in the transaction of the client. Should the BeanA had an open transaction, the BeanB will throw a RemoteException.
Bean and Container Managed Transactions

Here is a well known definition: **transaction is a logical unit of work.**

This definition has the same meaning in EJB as in database management and other systems. The classic example of transaction used in database tutorials is the process of money transfer from checking to savings account. Let’s say a customer transfers $500 from saving to checking account. Here’s the SQL version of transaction:

```
Begin transaction
// If both statements are successful we have to commit
// transaction, otherwise it has to be rolled back
update savings set balance=(balance-500) where acctId =123
update checking set balance=(balance+500) where acctId=456
End transaction
```

When entity beans are called from session beans (or from a web layer), there may be a need to treat several database operation as one transaction that has to be either 100% successful or rolled back to the original state in case of any error.

You have a choice of using either **bean-managed** or **container-managed** transactions.

In case of bean-managed transactions, developer has to take care of transaction processing. Here’s how you can rollback a transaction in the bean-managed mode:

```java
sessionContext.setRollbackOnly();
...
myTrans.rollback();
```

In case of container-managed transactions, developer just need to specify transaction attribute in the bean’s deployment descriptor.

**More on EJB Deployment Descriptors**

Deployment descriptor is an XML file(s) that contains names of the beans, their home and remote interfaces, transactional attributes, security roles and other deployment parameters.
According to EJB specification every vendor must have at least one deployment descriptor – **ejb-jar.xml**. In case of the EJB specification 2.0, the valid properties are defined in the file called `ejb-jar_2_0.dtd`:

```xml
<?xml version="1.0" ?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>PortfolioBean</ejb-name>
      <home>
        com.sdp.sts.beans.session.PortfolioHome
      </home>
      <remote>
        com.sdp.sts.beans.session.PortfolioUser
      </remote>
      <ejb-class>
        com.sdp.sts.beans.session.PortfolioBean
      </ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
    <session>
      <ejb-name>OrderManagerBean</ejb-name>
      <home>
        com.sdp.sts.beans.session.OrderManagerHome
      </home>
      <remote>
        com.sdp.sts.beans.session.OrderManager
      </remote>
      <ejb-class>
        com.sdp.sts.beans.session.OrderManagerBean
      </ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Usually server vendors have additional XML files with specific to their server properties such as JNDI names, number of beans in the pool, JDBC
parameters, etc. Below is a sample of WebLogic descriptor file called **weblogic-ejb-jar.xml**:

```xml
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>PortfolioBean</ejb-name>
    <jndi-name>Portfolio</jndi-name>
  </weblogic-enterprise-bean>
  <weblogic-enterprise-bean>
    <ejb-name>TradeOrder</ejb-name>
    <jndi-name>Order</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

WebLogic server also requires a file **weblogic-cmp-rdbms-jar.xml** for entity beans with container-managed persistence (CMP). This file contains mappings of the bean’s fields to database table columns.

To deploy a bean you must have the JAR file containing all of the compiled classes for your EJB home interface, remote interface, and implementation classes. The JAR must also have a META-INF directory containing a valid file **ejb-jar.xml**.

Optionally, the deployment JAR could contain the weblogic-ejb-jar.xml and weblogic-cmp-rdbms-jar.xml files.

The table below lists some XML elements that are being used in EJB Deployment Descriptor files:

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
<th>Data example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejb-jar</td>
<td>Root element of EJB Deployment Descriptor</td>
<td></td>
</tr>
<tr>
<td>enterprise-beans</td>
<td>Section for declaring one or more beans</td>
<td></td>
</tr>
<tr>
<td>session</td>
<td>Section for declaring session beans</td>
<td></td>
</tr>
<tr>
<td>entity</td>
<td>Section for declaring entity beans</td>
<td></td>
</tr>
</tbody>
</table>
Deploying EJB in WebLogic Server

Perform the following steps to deploy EJB in WebLogic Server

Step 1. Set EJB deployment properties

The EJB provider should create an `ejb-jar.xml` file and other XML deployment files. These files usually reside the JAR in a top-level of the directory `META-INF`.

Step 2. Generate EJB container classes

The `ejbc` compiler generates classes based on user-created classes and deployment properties provided in deployment descriptors. For example, if you indicate that your EJBs will be used in a cluster, `ejbc` creates special cluster-aware classes that will be used for bean deployment.

Step 3. Make the EJB jar(s) available to the WebLogic server.

This could be done using **one** of the following methods:
a) Copy the jar into the application directory (see the previous lesson). Weblogic will find and deploy its beans during start-up (The WebLogic server has to be started in the development mode - see Appendix B).

b) Use the Administrative Console – select the section Deployments | EJB on the left and click on the link Install New EJB on the right side.

c) Use so called Hot Deployment - deploy beans without restarting a WebLogic Server. Below are sample commands that use Weblogic’s utility deploy and should be entered from the command window. The first command deploys a new bean, the second one removes beans that were deployed earlier, and the third one re-deploys a newer version of the beans.

c:\>java weblogic.deploy -port 7001 -host localhost -component MyTestBean:myserver deploy mypassword MyTestBean c:\myproject\myserver\MyBeans.jar

c:\>java weblogic.deploy -port 7001 -host localhost -component MyTestBean:myserver undeploy mypassword MyTestBean c:\myproject\myserver\MyBeans.jar

c:\>java weblogic.deploy -port 7001 -host localhost -component MyTestBean:myserver update mypassword MyTestBean c:\myproject\myserver\MyBeans.jar

Database Connection Pools

Let’s recall the process of working with databases using JDBC: load a JDBC driver, get the Connection object, create one of the Statement objects, execute the SQL query or update, process the result set, if any, and close the connection. The most expensive steps in terms of time are creation and closing of the database connections. When a program closes a connection, this object becomes a candidate for the garbage collection. It’s not a good idea to repeat this process again and again for every database request.

Java application servers support so called Connection Pools - the objects that allow to create some pre-defined number of database connections and re-use them instead of re-creating a new connection for each request. No additional programming is required - just use the appropriate application server’s utility to create a connection pool by giving it a name, for example OraclePool, minimum and maximum number of connections and the name
of the database driver class. In Weblogic 6.1 you create a connection pool using administration console. Below is a snapshot of a screen that shows you sample configuration for JDBC Oracle drivers (type 2).

![JDBC Data Sources](image)

**JDBC Data Sources**

JDBC specification 2.0 has a class `javax.sql.DataSource` that is used to work with database connections pools. The datasource also should be created using the application server's administrative tool. In case of Weblogic do the following:

- select the option **JDBC | Datasources** (see the screen shot above).
- enter the name of the data source (i.e. `oracleDS`) and the name of the pool to be used for database connections (i.e. `oraclePool`).
After this is done, you can access the data source from your session bean, servlet or other Java program by a JNDI lookup:

```java
Connection con = null;
ResultSet rs   = null;
Statement stmt = null;
try {
    ctx = new InitialContext();
    DataSource ds  = (DataSource) ctx.lookup("oracleDS");
    con = ds.getConnection();

    stmt = con.createStatement();
    stmt.execute("select symbol, price, quantity from Portfolio");

    rs = stmt.getResultSet();
    while (rs.next){
        ...
    }
} catch(Exception e){
    ...
}
finally {
    try{ rs.close();} catch(Exception e) {...}
    try{ stmt.close();} catch(Exception e) {...}

    // return the Connection object to the pool.
    // It won’t be garbage collected!
    try{ con.close();} catch(Exception e) {...}
}
```

Consider the use of the precompiled SQL (PreparedStatement) or database stored procedures (CallableStatement) for better performance.

Do not forget to change the `<transaction-type>` attribute in the deployment descriptor to `Bean` for bean-managed transactions.

**Implementing STS (Session Beans Version)**

In this lesson we’ll start writing a simple Stock Trading System using session beans based on the specification from the previous lesson. It has one stateful and one stateless beans. Here’s some highlights of this application:

- This version will have 2 session beans: `OrderManager` and `CustomerProcessor`.  

The OrderManager has one method `placeOrder()`.

The `CustomerProcessor` has 2 methods: `getClientData()` and `validateUser()`.

Two value objects are being used: `OrderData` and `ClientData`.

A utility class `InputValidator` provides tests user's input for digits, length, etc.

The client prints a menu in the dos window and reacts on the user’s input.

I’d like to keep this book thin, so only some of the classes or code fragments will be shown here, but complete code of this application is available online.

This is a code fragment from the class `JavaClient` that performs user validation:

```java
public void doLogon() {
    boolean cont = true;
    try {
        input = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("         *** STS Logon ***");
        while (cont) {
            System.out.print("               name:     ");
            user = input.readLine();
            System.out.print("               password: ");
            password = input.readLine();
            try {
                Properties properties = new Properties();
                properties.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
                properties.put(Context.PROVIDER_URL, "t3://localhost:7001");
                ctx = new InitialContext (properties);
                CustomerProcessorHome cpHome = (CustomerProcessorHome) ctx.lookup("CustomerProcessorHome");
            }
        }
    }
}
```
CustomerProcessor officer = cpHome.create () ;

System.out.println(" Validating User... ");

if (officer.validateUser(user,password)) {
   // A valid customer. Since the officer
   // represents a Stateful session bean, its
   // instance is bound to this client and
   // that's why the next call gets the data
   // populated by the validateUser() method.
   ClientData myClient =
      officer.getClientData () ;

   System.out.println(" Hello " +
      myClient.getFullName () ;

   sqoutes = new StockQuotes ();
   cont =false;
} else {
   System.out.println("Unknown User - " +
      " try again");
}

} catch (NamingException e) {
   System.err.println ("Could not find " +
      " CustomerProcessorHome: " +e.toString());
} catch (CreateException e) {
   System.err.println ("Could not create " +
      " CustomerProcessor: " +e.toString());
} catch (Exception e) {
   System.err.println ("Could validate user: " +
      e.toString());
}

} // end while

fwd = enterLetter(
   "Show orders/Enter orders/Quit:" ,"SEQ");
} catch (Exception e) {e.printStackTrace();}
}


The class **ClientData** is used for the data exchange between the client and the server:

package com.sdp.sts.utils;

import java.util.Vector;
public class ClientData implements java.io.Serializable {
    // try to remove “implements Serializable”, re-deploy
    // and see what happens

    private String userName;
    private String role;
    private String customerID;
    private String lastName;
    private String firstName;
    private String phone;
    private String eMail;

    private Vector accounts = new Vector();
    public ClientData () {}

    public Vector getAccounts() {
        return accounts;
    }
    public void setAccountID(Long acctID){
        accounts.add(acctID);
    }

    public void setUsername(String value){
        userName=value;
    }
    public void setRole(String value){
        role =value;
    }
    public void setCustomerID(String value){
        customerID=value;
    }
    public void setLastName(String value){
        lastName=value;
    }
    public void setFirstName(String value){
        firstName=value;
    }
    public void setPhone(String value){
        phone=value;
    }
    public void setEMail(String value){
        eMail=value;
    }
    public String getFullName(){
        return firstName + " " + lastName;
    }
    }
}
The interface **CustomerHome**:  

```java
package com.sdp.sts.beans.session;

import java.rmi.*;
import javax.ejb.*;

public interface CustomerProcessorHome extends EJBHome {
    public CustomerProcessor create()
        throws RemoteException, CreateException;
}
```

The remote interface **CustomerProcessor**:  

```java
package com.sdp.sts.beans.session;
import java.util.Hashtable;
import java.rmi.*;
import javax.ejb.*;
import com.sdp.sts.utils.ClientData;
import com.sdp.sts.beans.session;

public interface CustomerProcessor extends EJBObject {
    public boolean validateUser(String user, String pwd)
        throws RemoteException;
    public ClientData getClientData() throws RemoteException;
}
```

The **class CustomerProcessorBean** connects to the database using the data source called `oracleDS` and validate the user against the database table `Users`:

```java
package com.sdp.sts.beans.session;

import java.io.*;
import java.sql.*;
import com.sdp.sts.utils.ClientData;
import java.rmi.*;
import java.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class CustomerProcessorBean implements SessionBean {
    private SessionContext sessionContext;
    private ClientData clientData = null;
    public void ejbCreate() {  }
    public void ejbRemove() {  }
    public void ejbActivate() {  }
```
public void ejbPassivate() { }

public void setSessionContext(SessionContext context) {
    sessionContext = context;
}

public boolean validateUser(String username, String password) throws RemoteException {
    ResultSet rs = null;
    Statement stmt = null;
    Connection con = null;
    try {
        Context ctx = new InitialContext();
        javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup("oracleDS");
        con = ds.getConnection();
        stmt = con.createStatement();
        String sqltorun = "SELECT * FROM Users a,Customer b," + " Account c WHERE a.username='" + username + "' and a.password='" + password + "' and a.userid=b.customerid and " + " b.customerid=c.customerid";
        rs = stmt.executeQuery(sqltorun);
        boolean isEmpty = true;
        while (rs.next()) {
            clientData = new ClientData();
            clientData.setUsername(rs.getString("username"));
            clientData.setRole(rs.getString("role"));
            clientData.setCustomerID(rs.getString("customerID"));
            clientData.setLastName(rs.getString("lastname"));
            clientData.setFirstName(rs.getString("firstname"));
            clientData.setPhone(rs.getString("phone"));
            clientData.setEMail(rs.getString("email"));
            clientData.setAccountID(new Long(rs.getString("accountID")));
            isEmpty = false;
        }
        if (isEmpty) {
            // did not find the user
            return false;
        }
    } catch (SQLException ex) {
        System.out.println("CustomerProcessor.validateUser." + " Database error: ");
        ex.printStackTrace();
        throw new RemoteException(ex.toString());
    }
}
catch(Exception ex) {System.out.println(
    "CustomerProcessor.validateUser error: ");
ex.printStackTrace();
throw new RemoteException(ex.toString());
}
finally{
    try{
        rs.close();
        stmt.close();
        con.close();
    }catch (Exception e) {e.printStackTrace();
}
System.out.println("In CustomerSessionBean. The user "+ username + " is valid***\n");
return true;
}
public ClientData getClientData() throws RemoteException{
    return clientData;
};

How to Run the Stock Trading System

Perform the following steps to run the session beans version of the STS:

Step 1. Open two command windows and get to the Weblogic’s application directory (i.e. c:\bea\wlsserver6.1\config\mydomain).

Step 2. Run the command setEnv in both windows to set environment variables:

    c:\bea\wlsserver6.1\config\mydomain>setEnv

Step 3. Switch to your working directory in the first window and run the command file to build and deploy the EJB:

    c:\practice>build

Step 4. Start the WebLogic server in the second command window. Check the administrator console to see if the bean has been deployed.

    c:\bea\wlserver6.1\config\mydomain>startWebLogic

Step 5. Open the WebLogic’s console in your browser at
http://localhost:7001/console/. Define a connection pool named oraclePool. After entering pool parameters in the General and Connections tabs, click on the tab Targets and move myServer from Available to Chosen box. If you are using Oracle 8 oci drivers, add directories with WebLogic oci DLLs (i.e. bea/wlserver6.1\bin\oci816_8) and Oracle oci client (i.e. Oracle\Ora81\bin) to the PATH variable of your OS. In case of connection problems copy all DLLs from the oci816_8 to the directory Oracle\Ora81\bin.

Step 6. Create a datasource named oracleDS by selecting on the console
Datasources | Create a New DataSource. Enter the following:

Name: Oracle Data Source
JNDI Name: oracleDS
Pool Name: oraclePool

After entering the general pool parameters, click on the tab Targets and move myServer from the box Available to the box Chosen.

Step 7. Create the database table using Oracle SQL*Plus utility and provided scripts:

SQL>@c:\practice\TablesCreate
SQL>@c:\practice\TablesData

Step 8. Compile and run the EJB client in the first command window:

c:\practice>run

Logon to the system by entering a valid user ID and Password that exist in the database table Users from the Lesson 18 and follow the instructions to create your orders, check portfolio, etc.
Lesson 20

Entity Beans

Introduction

Entity beans represent persistent data that are usually stored in database tables, but also could be stored in the flat files or some other form. These beans survive the server crash because the data are saved on disk.

While entity beans could be accessed from various types of clients, preferable clients are session beans. Since the entity beans represent the database model of your application, it’s not a good idea to expose their structure to the clients. If you did this, the client’s code should have been changed whenever the data model have changed. That’s why it’s better to hide the entity bean processing behind a session bean. Such design is called a façade pattern.

An instance of an entity bean represents one row from a database table or a result set produced by an SQL statement or stored procedure.

Persistence is a process of saving data on some media (hard disk, tape, etc.). Every time the values of the entity bean are changed in memory, appropriate updates are performed in the database or a file.

Two types of bean persistence will be explained later in this lesson: Container-Managed Persistence (CMP) and Bean-Managed Persistence (BMP).

As with session beans, EJB container maintains a pool of instances of the same entity bean. For example, several instances of the bean Employee exist in the pool, but only when a particular person is being processed, the bean gets initialized with specific values, i.e. “John Smith”.

Fields of an entity bean are usually mapped to the corresponding columns of a database table. For example, here’s the bean:

```java
public class EmployeeBean implements EntityBean {
    public String employeeID;
    public String lastName;
    public String firstName;
}
```
and this is the corresponding database table:

<table>
<thead>
<tr>
<th>ID</th>
<th>Lname</th>
<th>Fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-45-6789</td>
<td>Smith</td>
<td>Peter</td>
</tr>
<tr>
<td>789-45-1230</td>
<td>Fain</td>
<td>Yakov</td>
</tr>
<tr>
<td>234-45-2121</td>
<td>Smith</td>
<td>Olga</td>
</tr>
</tbody>
</table>

Application servers usually provide tools to map bean’s fields to the database table columns. The mapping could also be done by editing the deployment descriptor file.

Types of Persistence

Even though relational databases are not the only places for bean’s persistence, we’ll be using SQL-based examples. There are two options of saving bean’s state in a database:

- **Container-Managed Persistence**

  In this mode developer just maps the fields of a bean to the corresponding database table columns. You do not need to write SQL – the EJB Container will do it for you. For each setter method on a bean container will generate an **update** SQL statement. For example, the following code:

  ```java
  employee.setSalary(50000);
  ```

  could lead to **automatic generation** of the following SQL statement:

  ```sql
  update Employee set salary=50000
  where ID='123-45-6789'
  ```

  The ‘123-45-6789’ is a unique employee ID.

- **Bean-Managed Persistence**

  For this type of persistence developer has to write some code in the bean’s methods `ejbCreate()`, `ejbStore()`, `ejbLoad()` or `ejbRemove()`. It could be SQL statements, stored procedure calls, Java I/O streams, etc.
Primary Key of an Entity Bean

The chances are that the database has more than one employee with the last name “Smith”. How would you tell them apart? This could be done by defining a primary key for each entity bean.

Primary key is one or more fields that uniquely identify an entity bean.

If one field can uniquely identify your bean, i.e. employeeID, you can use String, Long or other Java classes to define the primary key variable.

If the bean needs multiple fields to uniquely identify itself, create a primary key class for this bean. For example, the EmployeeHistory bean may have multiple records for the same employee and each of the records has unique combination of the employeeID and titleID fields. In this case you should create the class EmployeeHistoryPK containing both - employeeID and titleID values. First, the client should create an instance of the class EmployeeHistoryPK, assign some values to its member variables and only then create the entity bean supplying the instance of the EmployeeHistoryPK to one of the create() or find() methods.

Two entity beans are considered to be identical if they have the same data in the primary key fields.

Below is a fragment from the file ejb-jar.xml that defines the primary key class name:

<enterprise-bean>
  <entity>
    ...
    <prim-key-class>EmployeeHistoryPK</prim-key-class>
  ...
  </entity>
</enterprise-bean>

In case of single-field primary keys, one of the Java wrapper classes could be used. For example, if a String variable is being used as your primary key, here’s how its data type looks in the descriptor file:

<enterprise-bean>
  <entity>
    ...
    <primkey-field>String</primkey-field>
  ...
  </entity>
</enterprise-bean>
The primary key class must implement `Serializable` interface, have a no-argument constructor and use only the public variables.

If your application server can not automatically generate primary key classes – create them manually and include two methods in this class - `equals()` and `hashCode()`. These methods are used to check if two beans represent the same data. For example:

```java
public class EmployeeHistoryPK implements Serializable{
  public String employeeID;
  public String titleID;

  // constructors
  public EmployeeHistoryPK(){
  }
  public EmployeeHistoryPK(String id, String title){
    employeeID = id;
    titleID = title;
  }

  public boolean equals(Object obj){
    if (obj == null || !(obj instanceof EmployeeHistoryPK)){
      return false;
    } else if (((EmployeeHistoryPK)obj).employeeID.equals(employeeID) &&
      ((EmployeeHistoryPK)obj).titleID.equals(titleID)) {
      return true;
    } else {
      return false;
    }
  }

  public int hashCode(){
    StringBuffer strBuf = new StringBuffer();
    strBuf.append(employeeID);
    strBuf.append(titleID);
    String str = strBuf.toString();
    int hashCode = str.hashCode();
    return hashCode;
  }
}
```
Finding Entity Beans

Every entity bean must implement interface `javax.ejb.EntityBean` which declares such methods as `ejbLoad()`, `ejbStore()`, `ejbPostCreate()` and others.

The Home interface of an entity bean must include the method `findByPrimaryKey()` and may have other finder methods supporting search by values, for example:

```java
public interface EmployeeHistoryHome extends EJBHome {
    public Employee create(EmployeeData emp)
        throws RemoteException, CreateException;
    public EmployeeHistory findByPrimaryKey(Long primaryKey)
        throws RemoteException, FinderException;
    public Collection findAll()
        throws RemoteException, FinderException;
    public Collection findByLastName(String lastName)
        throws RemoteException, FinderException;
}
```

To simplify the subject, consider the old-fashioned SQL way of finding employee data in a database:

```sql
select * from EmployeeHistory
where employeeID='123-45-6789' and titleID='PA1'
```

Below is a fragment of a client’s code that finds the same employee using the primary key class and the finder method.

```java
EmployeeHistoryHome empHome = (EmployeeHistoryHome) ctx.lookup("EmployeeHistoryHome");
EmployeeHistoryPK primKey = new EmployeeHistoryPK("123-45-6789", "PA1");
EmployeeHistory emp = empHome.findByPrimaryKey(primKey);
String fname = emp.getFirstName();
...
The method `findByPrimaryKey()` for obvious reason always returns a single instance of the bean.

Finders by value return a collection of remote interfaces and you could process each of them using Java `Iterator` interface, for example:

```java
Collection empHistory = empHome.findByLastName("Smith");
Iterator iter = empHistory.iterator();
while (iter.hasNext())
{
    EmployeeHistory empRecord = (EmployeeHistory) iter.next();
    String title = empRecord.getTitleID();
    ...
}
```

Life Cycle of an Entity Bean

When a client calls the method `create()` on a Home interface, the entity bean receives a reference to its `EntityContext` from the EJB container through the callback `setEntityContext()`. Right after that, the corresponding `ejbCreate()` is being invoked on the bean. Typically this also creates a row in a database table.

Right after the `ejbCreate()` container invokes the method `ejbPostCreate()`, which could be used by developers for some initialization routines, if needed.

EJB container maintains a pool of instances of the entity beans which do not represent any particular bean until the `ejbCreate()` or `ejbActivate()` brings them to life. At this point the bean is being moved from the pooled to the ready state.

In entity beans methods `ejbActivate()` and `ejbPassivate()` work differently than in stateful session beans. Activation here means retrieval of a bean from the pool and initialization of its member variables with specific values (“John”, “Smith”), etc. Passivation process erases the state of the bean and returns it back to the pool.

Applications like Stock Trading Systems require a real-time transaction processing, where the speed of execution is the most important requirement. It really helps if your application server provides so called caching of entity beans. Entity beans are created and stay in the cache memory and all subsequent client calls `create()` or `find()` do not hit the database and work with the cached versions.
A bean stays in the cache for a time period specified in the deployment descriptor’s cache clearing policy settings. The application server PowerTier from Persistence Software should be considered for the systems which require fast performance.

**Entity Beans with BMP**

BMP means that the bean itself has the code required to save, update and destroy itself. Guess what, you are the one who has to write this code!

The methods `find()`, `ejbLoad()`, `ejbStore()`, `ejbCreate()` and `ejbRemove()` have to be implemented. You might also program methods `ejbPostCreate()`, `ejbActivate()` and `ejbPassivate()`.

When the client calls the method `create()` on the remote interface, the `ejbCreate()` is being invoked on the bean, and usually performs an SQL Insert.

When the client calls the method `find()`, the `ejbLoad()` is being invoked on the bean, and typically performs an SQL Select.

When the client calls one of the setter methods, the `ejbStore()` is being invoked and typically performs an SQL Update.

When the client calls the method `remove()` method, the `ejbRemove()` is being invoked and typically performs SQL Delete.

These are the corresponding actions in entity beans and relational database:

<table>
<thead>
<tr>
<th>Entity Bean Method</th>
<th>SQL statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ejbCreate()</code></td>
<td>Insert</td>
</tr>
<tr>
<td><code>ejbLoad()</code></td>
<td>Select</td>
</tr>
<tr>
<td><code>ejbRemove()</code></td>
<td>Delete</td>
</tr>
<tr>
<td><code>ejbStore()</code></td>
<td>Update</td>
</tr>
</tbody>
</table>

The code snippet below gives an example of the method `ejbCreate` of the `EmployeeBean`:

```java
public class EmployeeBean implements EntityBean {
    public String employeeID;
    public String lastName;
    public String firstName;
```
public String ejbCreate(String empID, String lName,
                       String fName){
    employeeID=empID;
    lastName=lName;
    firstName=fName;
    ...
    Connection con = myDataSource.getConnection();
    Statement stmt = con.createStatement(
        "insert into EMP values('' + employeeID + 
        , '' + lastName + "," + firstName + "'') ");
    stmt.executeUpdate();
    ...
}

Entity Beans with CMP

CMP means that you do not need to write code to save, update and destroy the bean – the EJB container will do it.

How the container knows in where in the database save the bean EmployeeHistory, what's the table name and which column corresponds to the bean's variable lastName? This information has to be provided in the deployment descriptor file which has to contain the names of your CMP fields, name of the database table and the name of the corresponding table column for each field. When the table or column names change, you'll have to modify the deployment descriptor and re-generate the bean.

Name of the xml file containing the mappings between bean fields and table column names is vendor specific.

This is how it could be done in WebLogic application server:

- Specify bean's property such as primary key class and CMP fields in the ejb-jar.xml.

- Create the file weblogic-ejb-jar.xml and include there a name of the xml file that has mappings between bean's CMP fields and database table columns. See the property WebLogic_CMP_RDBMS on the next page – it has a value META-INF/weblogic-cmp-dbms-jar.xml which is a name of the file with mappings.
• Create the file `weblogic-cmp-dbms-jar.xml` with mappings between fields and table columns.

The EJB specification 2.0 includes substantial changes in the CMP model which will be discussed later in the section EJB-QL.

Below is a fragment of the `ejb-jar.xml` for the EmployeeHistory bean with CMP. This file has to be located in the directory META-INF.

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>EmployeeHistory</ejb-name>
      <home>EmployeeHistoryHome</home>
      <remote>EmployeeHistory</remote>
      <ejb-class>EmployeeHistoryBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>EmployeeHistoryPK</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field>
        <field-name>employeeID</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>titleID</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>lastName</field-name>
      </cmp-field>
    </entity>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>EmployeeHistory</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>

</ejb-jar>
```

Here is a fragment from the `weblogic-ejb-jar.xml` for the EmployeeHistory bean with CMP:
Below is a fragment from the file `weblogic-cmp-dbms-jar.xml` that contains mappings between the bean's CMP fields and database table columns:

```xml
<?xml version="1.0"?>

<weblogic-rdbms-bean>
  <pool-name>oraclePool</pool-name>
  <table-name>Emp_Hist</table-name>
  <attribute-map>
    <object-link>
      <bean-field>employeeID</bean-field>
      <dbms-column>id</dbms-column>
    </object-link>
    <object-link>
      <bean-field>titleID</bean-field>
      <dbms-column>title_id</dbms-column>
    </object-link>
  </attribute-map>
</weblogic-rdbms-bean>
```
Entity Bean Relations

Objects in a real world have some relations between each other, for example, one Customer can have many Orders - we call it one-to-many relations. Other types of relations are one-to-one and many-to-many. If you've had a chance to work with relational databases, you know that you can write SQL queries (joins) that will extract data from more than one database table. Since entity beans usually map to database tables, the obvious question is how to specify bean relations and what's the equivalent to joins in the beans universe.

The application server PowerTier had the ability to specify bean relations and automatically generate supporting Java code even prior to EJB specification 2.0. But now this process becomes standardized. With CMP beans container will learn about container managed relations (CMR) from the Deployment descriptor ejb-jar.xml file. Base on the relations we could have, for example the method getOrders() on the CustomerBean. Since this method can return multiple objects, its return type could be a Java Collection. For the Customer-Order pair, the following xml would be required:
Below is a fragment from the file `weblogic-cmp-dbms-jar.xml` that contains mappings between the bean’s CMR fields and database table columns. It introduces the tags `<key-column>` and `<foreign-key-column>` which have similar to relational databases meaning. The sample below assumes that the table `Customer` has a primary key column called `id` and the table `Orders` has the foreign key `custID` referring to the table `Customer`.

```xml
<weblogic-rdbms-relation>
  <relation-name>Customer-Orders</relation-name>
  <weblogic-relationship-role>
    <relationship-role-name>customer-has-orders</relationship-role-name>
    <column-map>
      <key-column>id</key-column>
      <foreign-key-column>custID</foreign-key-column>
    </column-map>
  </weblogic-relationship-role>
</weblogic-rdbms-relation>
```

**Query Language - EJB-QL**

The query language similar to SQL has been introduced in EJB 2.0 specification to minimize dependency between CMP entity beans and underlying database tables. The query statements are specified in the xml deployment descriptors for the finders defined in the Home interface of the bean. For example, if the `OrderHome` defines the following finder:

```java
Collection findByStockSymbol(String symbol);
```

the `ejb-jar.xml` can have the following EJB-QL query:
In this example the `<ejb-ql>` expression is included in the CDATA xml block which tells XML parses that there is no need to parse this piece. This is done to avoid possible conflicts between the EJB-QL and XML special characters. The `?1` means the first parameter and it will be replaced with `IBM` if the following code is executed:

```java
Collection orders = OrdHome.findByStockSymbol("IBM");
if(!orders.isEmpty){
    Iterator iter = orders.iterator();
    while (iter.hasNext()){
        Order ord = (Order)iter.next();
        System.out.println("Order price=" + ord.getPrice());
    }
}
```

Since the query is specified in the xml descriptor, it could be easily modified without the need to recompile the Java code.

### The Final Project Assignment

Start working on the final project now and complete it after learning the material of the Lesson 21.

1. Create an entity bean with BMP called `OrderBean` and move there SQL from the `OrderManager` session bean. The `OrderManager` bean has to create the `OrderBean` instances and call appropriate methods - `createOrder()`, `modifyOrder()` or `cancelOrder()`. The method `cancelOrders()` should not delete the order from the database.

2. Create a class that will send a message (an instance of the `OrderData` object) to a queue `StockExchangeIn`. 
3. Create an independent Java class called StockExchange. This class has to implement MessageListener interface and perform the following operations:

- Get the message from the queue StockExchangeIn
- Extract the OrderData object from the message and assign a random execution price to it.
- Put the OrderData object into the queue StockExchangeOut.

4. Create a message-driven bean that will be listening to the queue StockExchangeOut and modify the database record with the received execution price.

5. Create a GUI interface to replace the class JavaClient – use Java Swing applet, application or JSP. The GUI screen has to allow order input and display of the open orders. As soon as the execution comes in from the Stock Exchange, the appropriate order data has to be refreshed on the screen.

   If all of the above is done and you're still bored – perform the next step.

6. Download and install evaluation copy of one of the following servers: IBM's WebShpere from http://www7b.boulder.ibm.com/wsdd/downloads/ or Macromedia's JRun from http://macromedia.com/downloads/. Read the relevant documentation and re-deploy our Stock Trading System in the new environment. First you can keep using WebLogic as the Messaging server, but eventually it has to be replaced by the one of the following MOM products: FioranoMQ, SonicMQ or MQSeries.

Resources

1. EJB-QL Tutorial
   http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html

2. Plenty of EJB resources:
   http://www.ejbean.com/
Lesson 21

Java Messaging Service

Introduction

People are sending messages to each other using e-mail. Applications could send messages to each other using Message Oriented Middleware (MOM).

MOM is not the only way for data exchange. Two applications might also talk to each other using CORBA, RPC, Java RMI, Microsoft’s DCOM, etc. Suppose you place an order to buy some stocks by calling the method `placeOrder()` — this is a synchronous or blocking call. The calling program can’t continue until the code in the `placeOrder()` is finished.

One of the major advantages of MOM is that it provides asynchronous communication. It somewhat similar to e-mail operations — you do not have to be online when someone sends you a message — you could read it later.

The process of placing an order comes down to placing an object that describes your order into a message queue. The calling program may continue its execution without waiting until the processing of the order is finished. Another program on the other end of the queue should de-queue and process the messages. When your order will be processed, appropriate message will be put into another queue and, if your application is active at that time, it will de-queue the message immediately on its arrival. In case of guaranteed delivery (see below) the messages will remain in the queue if your application is not running.

JMS is a standard API to work with MOM. JMS itself does not transport messages. JMS to MOM is the same as JDBC to a relational DBMS. Java applications could use the same JMS classes with any MOM vendor. These are some popular messaging products:

- MQSeries (IBM)
- Tibco Rendezvous (Tibco Software)
- SonicMQ (Progress Software)
- SpiritWave (SpiritSoft)
- FioranoMQ (Fiorano)
Two Modes of Message Delivery

A program could either send or publish a message. If it sends a message to a particular queue, we call it **Point-to-Point** (PtP) messaging. In this mode a message is deleted from a queue as soon as it is successfully received.

If a program publishes a message, it’s called **Publish/Subscribe** (Pub/Sub) mode. A message is published to a particular topic and many subscribers could subscribe for it. Some topic examples are PriceDropAlert, BreakingNews, etc. In this mode a message is deleted from a queue as soon as all subscribers received it.

Message delivery could be **guaranteed** - MOM will keep the message in a queue until the receiver gets it. In this mode messages are persistent – they are saved by a MOM vendor.

In a **non-guaranteed** mode – MOM will deliver a message only to active receivers.

JMS Classes And Terms

Below are the names and a short description of major JMS classes. All these classes could be found in the package `javax.jms`.

- **Queue** - a place to put/get your messages. The messages will be retrieved using the First In First Out (FIFO) rule. A message **producer** (sender) puts messages in a queue and a message **consumer** (receiver) de-queues them.

- **QueueConnection** - an object that represents a particular connection to MOM (similar to the JDBC class `Connection`).

- **QueueConnectionFactory**
  An object that creates `Connection` objects (similar to the JDBC class `DataSource`).

- **QueueSession** - an object that represents a particular session between the client and MOM server.

- **QueueSender** - an object that actually sends messages.

- **QueueReceiver** - an object that receives messages.
• **TopicPublisher** - an object that publishes messages (similar to the QueueSender).

• **TopicSubscriber** - an object that receives messages (similar to the QueueReceiver).

• **Topic** - an object that is used in Pub/Sub mode to represent some important event.

• **TopicPublisher** - publishes messages to a topic so the TopicSubscribers could subscribe for it.

• **Message** - an object that contains a user’s message. It could be placed into a queue or published to a topic.

---

**Types of Messages**

Every message contains a header, properties and a body.

The header contains the message identification (message ID, destination, type, etc).

The optional properties could be set by a program to “mark” a message from a business point of view, for example UrgentOrder.

The optional body contains a message that has to be delivered. Below are the names and descriptions of JMS classes that could be placed in a message body. All these classes are inherited from the class `Message`:

• **TextMessage** - this could be any Java String.
• **ObjectMessage** – this could be any serializable Java object.

• **BytesMessage** – an array of bytes.

• **StreamMessage** – a stream of Java primitives.

• **MapMessage** – any key/value pairs, for example id=123.

## How to Send a Message

Queues have to be created before a program start sending messages. In real-world applications it's done by the server administrator. When Java developers get to know the queue parameters, they have a choice of either creating message objects (`QueueConnectionFactory`, `Queue`, etc.) programmatically every time when they need to send/receive a message, or create these objects once, bind them to a naming tree and just perform a JNDI lookup. The latter solution could be more practical. You can find a sample code for binding objects to a naming tree in the Lesson 17.

While some application servers (WebLogic) provide an administrative GUI tool to create and bind queues and topics to the JNDI tree, this either may not be the case with other vendors, or you may decide to use a third-party naming server. In any case it’s good to know how to bind JMS objects to JNDI trees programmatically.

The following steps have to be performed to send a message:

**Step 1.** Create (or get from JNDI) a `QueueConnectionFactory` object.

**Step 2.** Create a `Connection` object and call its method `start()`.

**Step 3.** Create a `Session` object.

**Step 4.** Create a `Queue` object.

**Step 5.** Create a `QueueSender` object.

**Step 6.** Create one of the `Message` objects (i.e. `TextMessage`) and put some data in it.

**Step 7.** Call the method `send()` on the `QueueSender`.

**Step 8.** Close the `QueueSender`, `Session` and `Connection` objects to release system resources.
Since creation and closing of connections and senders are slow operations, you may want to consider writing code for JMS Connection pools. Some MOM products may provide such pools, i.e. MQSeries starting from version 5.3. If you are using session beans as message producers, store the JMS Connection object in its member variable. This way you’ll have a pool of connections, because EJB container automatically creates pools of beans.

Below is a code fragment of a method that sends a message to a queue called TestQueue:

```java
public static void main(String args[]){
    try
        QueueConnectionFactory factory = new QueueConnectionFactory();

        //QueueConnectionFactory factory=ctx.lookup("MyQCF");
        QueueConnection connection = factory.createQueueConnection();
        connection.start();

        Session session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        Queue ioQueue = session.createQueue( "TestQueue" );
        QueueSender queueSender = session.createSender(ioQueue);
        TextMessage outMsg = session.createTextMessage();

        // Buy 200 shares of IBM at market price
        outMsg.setText("IBM 200 Mkt");
        queueSender.send(outMsg);
        queueSender.close();
    } catch (JMSException e){
        System.out.println("Error: " + e.getMessage());
    } finally{
        try{
            session.close();
            connection.close();
        } catch (Exception e) {...
    }
}
```
How to Receive a Message

You can receive messages either synchronously using the method receive(), or asynchronously by implementing the MessageListener interface and programming a callback onMessage().

The method receive() uses polling mechanism constantly asking for a message. It blocks the program which can not continue until the message is received or the specified time has expired:

```java
QueueReceiver queueReceiver = Session.createReceiver(ioQueue);
Message myMessage = queueReceiver.receive();
```

This is how to set the timeout interval of 500 milliseconds:

```java
Message myMessage = queueReceiver.receive(500);
```

An asynchronous callback onMessage() is a preferable way of receiving messages, because the message consumer is not sending multiple request just to see if the message is in the queue. The method onMessage() will be called immediately when a message put in the queue. The following steps have to be performed to receive messages:

Step 1. Create (or get from JNDI) the QueueConnectionFactory object.

Step 2. Create a Connection object and call its method start().

Step 3. Create a Session object.

Step 4. Create a Queue object.

Step 5. Create a QueueReceiver object.

Step 6. If your class implements MessageListener (see below) write implementation for the callback method onMessage(). If you decide to get messages synchronously, just call the method QueueReceiver.receive(). In this case implementation of the MessageListener interface is not needed.

Step 7. Close the Session and Connection objects to release the system resources.

The sample class myReceiver shows how to consume messages asynchronously. Its constructor creates JMS objects and registers itself as a
The message acknowledgment mode is defined at the time of creation of the Session object. The method createSession() has 2 arguments – if the first argument is true, the session is transacted, the value of the second argument is irrelevant and the message could be either committed, or rolled back by the consumer. If the method commit() has been called, the message is removed from the queue. The method rollback() leaves the message in
the queue. If the session is non-transacted as in our sample above, the second argument defines the acknowledgement mode.

- **AUTO_ACKNOWLEDGE** mode sends the acknowledgement back as soon as the method `onMessage()` is successfully finished.

- **CLIENT_ACKNOWLEDGE** mode requires explicit acknowledgement, i.e. `msg.acknowledge()`. This is a permission to delete the message from the queue.

- **DUP_OK_ACKNOWLEDGE** – in case of server’s failure the same message may be delivered more than once.

If more than one message is processed by the same `Session` object, acknowledgement of one message affects all messages from the same session.

### How to Publish a Message

Programs publish messages to topics, which should have been created in advance by the MOM system administrator. Multiple subscribers can get messages published to the same topic (this is a "one-to-many" mode).

Message publishing is very similar to the message sending, but the program should create a `Topic` instead of a `Queue`, a `Publisher` instead of a `Sender` and the method `publish()` should be called instead of a `send()`:

```java
TopicConnectionFactory conFactory = (TopicConnectionFactory) ctx.lookup("cn=primaryTCF");
TopicConnection connection = conFactory.createTopicConnection();
TopicSession pubSession = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
Topic myTopic = (Topic)ctx.lookup("Price_Drop_Alerts");
TopicPublisher publisher = pubSession.createPublisher(myTopic);
connection.start();
TextMessage message = pubSession.createTextMessage();
message.setText("Sale in ‘Century 21’ starts tomorrow");
```
How to Subscribe for a Topic

Subscribers could be **durable** and **non-durable**.

Durable subscribers are guaranteed to receive their messages – they do not have to be active at the time when a message comes.

Non-durable subscribers will be receiving messages only when they (subscribers) are active. This mode is similar to the way the chat rooms operate – you must be online to get the messages.

The code snippet below creates a non-durable subscriber. Two modifications have to made to this code to create a durable one: the client ID has to be assigned to the connection - `connection.setClientID(username);` and the method `createDurableSubscriber(topic)` should be used instead of the `createSubscriber(topic)`.

```java
TopicSession subSession =
    connection.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);

Topic topic = (Topic) ctx.lookup("Price_Drop_Alerts");

TopicSubscriber subscriber =
    subSession.createSubscriber(topic);

connection.start();

subscriber.setMessageListener(this);

public void onMessage(Message message) {
    String msgText;
    try{
        if (msg instanceof TextMessage){
            msgText = ((TextMessage) msg).getText();
            System.out.println("Got " + msgText);
        }else{
            System.out.println("Got a non-text message");
        }
    }catch (JMSException e){
        System.out.println("Error: " + e.getMessage());
    }
}
```
Message Selectors

If you have to share a queue with some other applications or developers from your team, use message selectors (filters) to avoid "stealing" somebody else's messages, for example:

```java
String selector = "StoreName=Century21";
session.createReceiver(queue, selector);
```

In this case the queue listener will de-queue only those messages that have a String property StoreName with the value “Century21”. Message producers have to set this property:

```java
TextMessage outMsg = session.createTextMessage();
outMsg.setText("Super sale starts tomorrow");
outMsg.setStringProperty("StoreName", "Century21");
```

If the message producers are not written in Java, they may not be able to set properties of the message. In this case the field correlationID from the message header could be used as a workaround.

Please remember that message selectors slow down the process of retrieval. The messages stay in a queue until the listener with matching selector will pick them up.

Selectors really help if your team has a limited number of queues and everyone needs to receive messages without interfering with others. But if someone will start the queue listener without selectors, it'll just drain the queue.

Message-Driven Beans

The EJB 2.0 specification defines a bean of a new flavor - Message-Driven Bean. Prior to this a EJB could not be used to receive JMS messages. This new bean does not have neither home nor remote interfaces, because clients do not need to access the bean - it just sits in memory and listens to a particular queue or a topic. These beans must implement two interfaces: MessageDrivenBean and MessageListener. When a message appears
into the queue, the EJB container picks one of the message-driven beans from
the pool, and passes the message to its method `onMessage()`.

```java
public class OrderListener
    implements MessageDrivenBean, MessageListener{

    MessageDrivenContext ctx;

    // A no-argument constructor is required
    public MyListener() {} // The business code goes here.

    public void onMessage(Message message){
    }

    public void ejbRemove()throws javax.ejb.EJBException {}

    public void setMessageDrivenContext(
        MessageDrivenContext ctx) throws javax.ejb.EJBException {
        this.ctx = ctx;
    }

    public void ejbCreate() {}
}
```

If you use message-driven beans instead of regular message receivers EJB
container gives you excellent freebies: distributed transaction processing,
automatic pooling, number of receivers is easily configured by specifying pool
size in deployment descriptor, co-location of receivers and other beans, and
simple assignment of queues or topics to the receivers in deployment
descriptors.

Below is a sample deployment descriptor `ejb-jar.xml` for a message-driven
bean `OrderListener`.

```
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD
Enterprise JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-
jar_2_0.dtd">
<ejb-jar>
    <enterprise-beans>
        <message-driven>
            <ejb-name> OrderListener </ejb-name>
            <ejb-class> OrderListener </ejb-class>
            <transaction-type>Container</transaction-type>
            <transaction-scope>Local</transaction-scope>
            <jms-acknowledge-mode>
                auto-acknowledge
            </jms-acknowledge-mode>
        </message-driven>
    </enterprise-beans>
</ejb-jar>
```
The descriptor above does not specify queue or topic names – these values are specified in vendor-specific descriptors, for example in `weblogic-ejb-jar.xml`:

```xml
<?xml version="1.0"?>
<weblogic-enterprise-bean>
  <ejb-name>OrderListener</ejb-name>
  <message-driven-descriptor>
    <destination-jndi-name>OrderQueue</destination-jndi-name>
  </message-driven-descriptor>
</weblogic-enterprise-bean>
```

The `OrderQueue` object has to be created, configured and bound to a JNDI tree in advance (see the WebLogic procedures below).
Configuring WebLogic JMS Objects

J2EE application servers may support JMS, but do not have their own message transport layer. In this case you’d need to bind the vendor-specific administered objects (queues, topics, connection factories) to a JNDI tree. WebLogic has its own transport layer and you just need to create and configure required JMS objects with the Administrative Console by completing the following procedure:

- Create a JMS Server by selecting JMS | Servers on the left panel and Create a New JMS Server on the right (if the message persistence is required, specify either a file or a database store). Click on the Targets tab and choose your application server.

- Create a default connection factory by selecting Connection Factories on the left and Create a New JMS Connection Factory on the right. Click on the tab Targets and choose your application server.

- Select the icon Destinations on the left and create queues or topics on the right.

The following screen snapshot will clarify the process.

The code snippet below shows how to find JMS objects using default WebLogic connection factories:
Context ctx = new InitialContext();
QueueConnectionFactory queueFactory =
(QueueConnectionFactory)
ctx.lookup("weblogic.jms.QueueConnectionFactory");
Queue myQueue = ctx.lookup("OrderQueue");

How to Run JMS Samples

This lesson comes with 3 sample programs: Point-to-Point, Pub/Sub and a message-driven bean. To run the samples, the WebLogic server has to be running, and it should have the JMS Server, the queue OrderQueue, the connection factory called weblogic.jms.QueueConnectionFactory, and the topic PriceDropTopic. You’ll need to have at least 3 command windows opened with the Weblogic's environment variables set.

Run the command file CompileJMS to compile the classes TestSender, TestReceiver, TestPublisher and TestSubscriber.

P2P Sample. The TestSender sends orders to buy stocks to the OrderQueue, and the TestReceiver receives them.

1. Execute the command file runReceiver in one DOS window.
2. Execute the command file runSender in another DOS window.

Pub/Sub Sample. The TestPublisher publishes messages to the topic PriceDropTopic and multiple TestSubscribers receive them. Since these subscribers are not durable, they must be active when messages are being published.

1. Execute the command file runSubscriber in two or three DOS windows
2. Execute the command file runPublisher in another DOS window

MDB Sample. The TestSender from the first sample sending message that are received by a message-driven bean.

1. Build and deploy the bean by executing the build script.
2. Re-start the WebLogic server.
3. Execute the command file `runSender` in another DOS window and watch the messages printed in Weblogic's command window.

**Resources**

1. JMS documentation and jars:
   http://java.sun.com/products/jms/docs.html

2. JMS Tutorial:
Appendix A

Java Technical Interviews.

Rules of the Game

The job interview is a game with well-defined rules. I am not going to discuss personal (non-technical) interviews here because there are plenty of good books on the subject. I'd rather share my experience of being a person who conducts technical interviews with you. Here's the usual scenario that I experienced many times: all of a sudden my boss asks me to interview a person that is already sitting “in the corner office”. I was in the middle of something else, but...

On one hand, I do not want to spend more than a half an hour asking Java questions, but on the other hand I want to give an honest, unbiased opinion about the technical skills this person has. I really do not want to make a mistake because if we hire a programmer who does not match the job requirements they would either leave soon (over-qualified), or we'll have to perform their job ourselves (under-qualified). You do not have to know everything to get the job – you need to know enough to do the required job and be comfortable with the offered salary.

I usually start the interview with general questions about the recent projects of the candidate and his or her role in these projects. After discussing technologies mentioned in the applicant’s resume, I switch our conversation to Java technical questions starting with the simple ones, and increasing complexity as we go. If the person has problems answering more complicated questions, I go back to the simpler ones. These are some hints that should help a job applicant make a better impression:

- Prepare a short speech describing your recent projects and your role in them.

- Prepare a couple of interesting technical problems that you had to resolve recently and try to switch the conversation to this area.

- Do not talk too much – keep your answers short to lower the chances of saying something wrong.
• Show your confidence in the subject – the interviewer wants to see it.

• Know the architecture of your project and understand why it has been designed this way. I was interviewing a person who said that they were using Java servlets. I asked - why servlets? He said that this design has been already made when he joined the company. This is an answer of a junior programmer. You need to be able to explain advantages and disadvantages of such architecture.

• Don’t critique the system design of your prospective employer – you’ll have a chance to do it later (if they hire you).

To Get or Not to Get Certified?

Sun Microsystems has various certification programs to rate your Java skills. The first one gives you the title “Sun Certified Programmer For Java 2 Platform”, if you pass the computer test with multiple choice type questions. Some other companies have similar programs (see Resources at the end of this appendix). I think it’s a very good idea to prepare yourself and go through the certification process, because it will definitely improve your understanding of the language. In some geographical areas it may also “improve the performance” of your resume (not in New York though, were there is an overabundance of certified programmers). It’ll also help you slip through computer screening tests that are used by job placement agencies.

The JavaPro magazine recently conducted a salary survey of Java programmers and it have shown that certifications from ... Microsoft lead to a salary increase. This proves the fact that the people who write and integrate applications using different technologies are in demand.

Technical Questions and Answers

Below are suggested technical interview questions and answers on various Java-related topics. I did not include the very basic questions such as “What’s the difference between the while and do while loops?” – there are plenty of multiple choice mock exams that will help test your knowledge of Java syntax. Below are the questions that I had to answer while working on various Java projects. Each question is marked with one or two asterisks with the following meaning:
* - questions for the beginners
** - advanced questions
*** - expert level questions

1.* Let's say you compiled a class and try to run it but JVM gives you the ClassNotFoundException. What could be the reason?
A. The variable CLASSPATH needs to have a dot (add ; in Windows and : in Unix) which is an instruction to JVM to look for the class in the current directory.

2.** What's the difference between interface and an abstract class?
A. An abstract class may contain code in method bodies, which is not allowed in an interface. With abstract classes, you have to inherit your class from it and Java does not allow multiple inheritance. On the other hand, you can implement multiple interfaces in your class. (See the Employee class discussion in the Lesson 13).

3.** How do you deploy a servlet in the application server that you currently use?
A. In most of the J2EE application servers you create a web archive (WAR) and copy it to the assigned directory, i.e applications directory in case of the WebLogic server.

4.** What are the default layout managers for frames and applets?
A. The BorderLayout is a default layout for frames and the FlowLayout is a default one for applets.

5.** What's the usage of a keyword static?
A. It's being used in declarations of methods and variables to make them available without creating an instance of the class. For example, the main() method is a static one. If a variable is static, its value is shared by all instances of this class.

6.** How can you force the garbage collection?
A. The class System has a method gc() to request the garbage collection. You can't force the garbage collection, but could request it, because JVM does not guarantee that it'll be started immediately.

7.** Describe the difference in the event model of Java 1.02 and Java 1.1 and above?
A. In Java 1.02 all events were implemented in the Component object and were triggered for all your components even if you did not need them (for example, you are not interested in the MouseMove events when the mouse is moving over a Button).
Starting from Java 1.1, listener interfaces with callback methods are used to process events and the programmer makes the decision as to which event to process and which one could be ignored. This model is called a delegation cause one class could “delegate” the processing of its events to a different one.

8.** Explain the usage of event adapters?
A. Some event listener interfaces declare multiple methods (i.e. `WindowListener` has 7 methods) and a class must implement all of them. Adapters already have all these methods predefined with empty bodies. So if a class needs to process just one of the `WindowListener` events, it has to override only this method of a `WindowAdapter`.

9.* How do you decide if an explicit casting is needed?
A. If you assign a superclass object to a variable of a subclass's data type, you need to do explicit casting. For example:

```java
Object a;
Customer b;
b = (Customer) a;
```

When it is a subclass to superclass assignment, the casting is performed automatically.

10.* Can you perform casting between objects of different types?
A. No you can’t. The objects must have a superclass - subclass relationship.

11.* What types of JDBC drivers do you know?
A. There are four types of JDBC drivers...
   Type 1: JDBC-ODBC bridge.
   Type 2: Java driver accessing native dbms driver.
   Type 3: A 3-tier driver.
   Type 4: Pure Java driver.

12.* Can a Java class be inherited from two classes?
A. No, Java does not allow multiple inheritance, but interfaces could be used as a workaround.

13.** What has to be done if a user’s Web browser has an older version of JVM than your applet needs?
A. Java plug-ins should be downloaded from the Sun MicroSystem’s site and installed on the user’s computer. After that, the HTML files have to be modified by a HTML converter program to point a browser to the new JVM.

14. * Can you write a Java class that could be used both as an applet and as an independent application?
A. Yes I can. In this case, the method `main()` has to be added to the applet.

15. * What’s the difference between applets and servlets in terms of the runtime environment?
A. Applets run on user’s machines under control of a WEB browser, they have security restrictions. Servlets run in the middle tier under control of the servlet container or an Application Server.
16.* What's the difference between constructors and the regular methods?
A. Constructors must have the same name as the class and can not return a value. They are only called once while regular methods could be called many times.

17.* What's the difference between the HTML methods Get and Post?
A. The method Get appends the parameters to the URL and the resulting URL is bookmarkable. The method Put allows sending objects as well as text data, while Get works with text only.

18.** What's a Cookie? Which Java components create them?
A. A cookie is an object that represents a name/value pair. Servlets or JSP could create and send cookies to a Web browser that saves them on the user's disk in a special directory. Cookies help a servlet identify a user. For example, a bank can store your account number in a cookie file on your machine, so you do not need to enter it on a logon screen.

19.** Can a non-abstract class have both abstract and concrete methods?
A. A class remains abstract until it has at least one abstract method.

20.** Explain the usage of Java packages.
A. This is a way to organize files when a project consists of multiple modules. It also helps resolve naming conflicts when different packages have classes with the same names. Packages access level also allows you to protect data from being used by the non-authorized classes.

21.* If a class is located in a package, what do you need to change in the OS environment to be able to use it?
A. You need to add a directory or a jar file that contains the package directories to the CLASSPATH environment variable. Let's say a class Employee starts with a statement package com.xyz.hr; and is located in the file c:\dev\com\xyz\hr\Employee.java. In this case, you’d need to add c:\dev to the variable CLASSPATH. If this class contains the method main(), you could test it from a command prompt window as follows:

```java
c:\>java com.xyz.hr.Employee
```

22.** Explain the usage of the keyword transient?
A. This keyword indicates that the value of this member variable does not have to be serialized with the object. When the class will be de-serialized,
this variable will be initialized with a default value of its data type (i.e. zero for integers).

Q23.** What do you know about thread synchronization? Explain the difference between

public void synchronized myMethod() { ... }

and

public void myMethod() {
    ... 
    synchronized (some_object) { ... }
}

A. The keyword synchronized is used to prevent race conditions when more that one thread tries to update some values. Synchronized blocks are preferable to synchronized methods because they place locks for shorter periods.

24.** What’s the difference between the methods sleep() and wait()?

A. The code sleep(1000); puts thread aside for exactly one second. The code wait(1000), causes a wait of up to one second. A thread could stop waiting earlier if it receives the notify() or notifyAll() call. The method wait() is defined in the class Object and the method sleep() is defined in the class Thread.

25.* Do all Java exceptions have to be declared or handled?
A. No, only listed exceptions that are inherited from the class Exception have to be taken care of. You do not process exceptions inherited from the class Error, which are caused by internal errors of the JVM.

26.* What is the usage of the CLASSPATH variable?
A. The CLASSPATH is an environment variable that tells the JVM where to look for Java classes during the run time. It plays the same role for Java classes as the PATH variable plays for OS executable programs.

27. When should the method invokeLater() be used?
A. This method is defined in the class SwingUtilities and is used to ensure that Swing windows will be updated through the event-dispatching thread.

28.** How could Java classes direct informational messages to the system console, but an error messages, say to a file?
A. The class `System` has a variable `out` that represents the standard output, and the variable `err` that represents the standard error device. By default, they both point at the system console:

```java
System.out.println("Please enter the password");
System.err.println("Could not logon the user...");
```

This how the standard output could be re-directed:
```java
Stream st = new Stream(new FileOutputStream("output.txt");
System.setErr(st);
System.setOut(st);
```

29.** How would you make a copy of a Java object with its state in memory during the runtime?**
A. I'd have this class implement `Cloneable` interface and call its method `clone()`.

30.* What's the difference between the events `windowClosed` and `windowClosing`?**
A. The `windowClosed` event is invoked when a window has been closed by a method `dispose()`. The `windowClosing` event is invoked when the user tries to close it from the window's system menu.

31.* What's the difference between the thread creation using the class `Thread` and the `Runnable` interface?**
A. To use the class `Thread`, your class has to be inherited from it and you just create an instance of your class. If a class implements the `Runnable` interface, you have to create the instance of your class, the instance of the `Thread` object, passing the `Runnable` class to it.

32.* What would you use to compare two `String` variables – the method `equals()`, or the operator `==`?**
A. I use the method `equals()` to compare the values of the `Strings` and the `==` to check if two variables point at the same `String`.

33**. What do you know about MVC?**
A. MVC is the abbreviation for the Model-View-Controller design pattern, which is used to separate presentation modules from the business logic and data ones. The `Model` part represents the data and the business logic of the application, the `View` is a visual representation (i.e. screens) and `Controller` accepts the data from the view and passes it to the Model. For example, JSP is a view, Servlet is a controller, and regular Java classes represent a model.

34.** How can you make the garbage collection more effective?
A. Object pooling helps lower the need of garbage collection.

35.* What HTTP error codes do you know? 
A. 404 - which means that the URL is not found, 
   500 - which means an internal error of the server program. 
   ...

36* Will the statement `File a = new File("xyz.txt");` create a file in the current directory? 
A. No, it just creates an object pointing to this file.

37.* How can a subclass call a method defined in a superclass? 
A. Java has a keyword `super`. If you need to call an overridden method use the following syntax `super.myMethod();` To call a constructor of the superclass, just write `super();` in the first line of the subclass's constructor.

38.* Write the code reacting to the `WindowClosing` event using `WindowAdapter`. 
A. `myFrame.addWindowListener (new WindowAdapter(){
    public void windowClosing( WindowEvent e)
    {System.exit(0); } });`

39.* What class access level do you need to specify to ensure that only classes from the same directory can access it? 
A. You do not need to specify any access level. In this case, Java will use the default package access level.

40.** What do you know about JNDI? 
A. It stands for Java Naming and Directory Interface – a Java API for directory servers. It is used to bind and lookup objects to a naming tree. One of the popular JNDI uses is for looking for the Enterprise Java Beans’ home interfaces.

41.* Do you have to put a call to a method that reads a file in the `try-catch` block? 
A. You have to either put it in the `try-catch` block or declare that the calling method may throw an exception, for example: `void myMethod() throws IOException{}`.

42.* Does it matter what order the `catch` statements for `FileNotFoundException` and `IOException` are written? 
A. Yes, it does. The `FileNotFoundException` is inherited from the `IOException`. The exception-subclasses have to be caught first.

43.* What's the loopback IP address?
A. It's a special IP address 127.0.0.1 that allows you to test network programs on a standalone machine (see the next question for an example).

44.* How can you test a servlet on a standalone computer?
A. The servlet engine has to be running on this machine, a servlet has to be deployed and you could specify a loop back IP address or a localhost in the Web browser, for example:

45.** Explain, in details, the data workflow between an HTML page and a servlet after the user presses the button Submit.
A. The web browser connects to the machine based on the entered URL, and if the servlet was not running there, its method init() will be called followed by the method service() which in turn calls the servlet’s doGet() or doPost() depending on the value of the action attribute in HTML tag <form>. The objects HttpServletRequest and HttpServletResponse are used for the interaction between the calling program and the servlet. The servlet’s output could be send back to the user using one of the methods of the HttpServletResponse, for example println().

46.** Explain the process of displaying the data from a database table on the screen - which JDBC classes and methods have to be used.
A. First you load the appropriated JDBC driver using the method Class.forName(), after that get the Connection object using DriverManager.getConnection() or a connection pool. Then create a Statement object and call one of its methods like executeQuery() or executeUpdate(). Process the ResultSet, if any and close the Connection, Statement and ResultSet objects.

47.* What’s the difference between the keywords final, finalize, and finally?
A. Depending on its position, the keyword final means, either that the variable is a constant, or that you can not override a method, or that you can not subclass a class.

The method finalize(), if defined, is called by a garbage collector when it’s ready to release a memory.

The finally is a clause in a try-catch block and you place there a code which has to be always executed, for example closing a stream.

48.** Can you declare variables in an interface?
A. Yes you can, but they should be final and static.
49.* What method is called when a user clicks on one of the buttons in a Frame window and how do you find out which button has been clicked?
A. The callback method actionPerformed() is being called with the argument is ActionEvent. The method ActionEvent.getSource() has the information about the component that generated the event.

50.** Is it possible to have an inner class accessing the private instance variable of the main class?
A. Yes you can, for example:

```java
public class OuterClass {
    private static String name = "Mary";
    public static class InnerClass {
        public static void printName() {
            System.out.println(name);
        }
    }
    public static void main (String args[]) {
        InnerClass.printName();
    }
}
```

51.** Can an inner class, declared inside of a method, access local variables of this method?
A. It's possible only if these variables are final.

52.* Give an example of an anonymous inner class.
A. An AWT event listener class could be created on the fly, for example:

```java
addActionListener (new ActionListener() {
    public void actionPerformed (ActionEvent e) {
        doSomething();
    }
});
```

53.* What could be used to keep track of sessions in servlets?
A. You can use cookies, URL rewriting, hidden fields, and the class HttpSession.

54.** Will session management with cookies always work?
A. No, if a user will disable cookies in the Web browser, it won’t work. In such cases, application servers usually automatically switch to URL rewriting.

55.** How can you stop a long running thread?
A. A class Thread has a deprecated method stop(), but it does not guarantees that it will do the job. Depending on the process, which is run by
this thread, you could try to close connections or open streams, if any, or use the method interrupt().

56.** When could a Java class be called a bean?
A. You call the class a bean if it has a non-argument constructor, implement Serializable interface and has public setter/getter methods for its private properties.

57.** What version of JSP and Servlets does your application server support?
A. You need to know the versions...For example, WebLogic 6.1 supports Servlets API 2.3, JSP 1.1 and EJB 2.0.

58.** What are the advantages of using JSP vs. servlets?
A. JSP allows you to separate presentation from business logic and the resulting web page could be modified by people who do not know Java.

59.** Give an example that shows the difference between the use of operators && and &?
String a=null;

if (a==null && a.length()>10) {...}

In this code the second expression in the if statement will not even be evaluated if the variable a is null. If a single ampersand would be used here, we'd get a NullPointerException.

60.** Name some predefined JSP variables.
A. request, response, out, session,...

61.** How do you deploy a JSP?
A. Usually the .jsp files have to be placed into a document root directory of the application server, or the WAR file should be created.

62.** How can you find out if a thread is not in a New or a Dead state?
A. You can call the Thread's method isAlive().

63.** What's the default port number for Web Servers?
A. 80

64.** What servlet’s method is an equivalent of a constructor’s?
A. Servlet’s method init() plays a similar role, but since all clients use the same instance of the servlet, you initialize only those variables that are allowed to have the same value for each user’s request, for example, the name of the database server.
65. Name some classes from the package java.util.
A. Vector, Hashtable, Properties, StringTokenizer, Date.

66. What's the difference between an Array and a Vector?
A. Arrays could be used if you know in advance the number of its elements. You do not need to know this to use a Vector – new elements could be added as needed. Arrays work faster because JVM only allocates memory once for all elements. Vectors may need to perform memory allocation multiple times.

67. How can you load a JSP from a servlet?
A. You should use the class RequestDispatcher class for this, for example:

```java
ServletContext sc = req.getServletContext();
RequestDispatcher rd = 
    sc.getRequestDispatcher("/MyPage.jsp");
rd.forward(req, res);
```

68. How can a servlet redirect a browser to a different URL?
A. You have to call the method sendRedirect() of the response object:
```
res.sendRedirect("http://www.xyz.com/demo.html");
```

69. What do you know about reflection?
A. It's a way of finding information about a Java class during the run time. For example, you can find out what the constructors are and what method signatures are of a particular class.
The class Class has such methods as getConstructor(), getFields(), getMethods() and others.

70. Can a Java program run out of memory?
A. Yes, it can. To prevent this, you can increase the size of the dynamic memory (heap) when you start the program. You can request, for example the minimum heap size of 64 Mb and the maximum size of 512Mb as follows:
```
c:\practice>java -Xms64 -Xmx512 MyProgram
```

71. How a servlet can send a cookie?
A. It needs to create an instance of the class Cookie and put it into the HttpServletResponse object, for example:
```
Cookie acctId = new Cookie("Account", "223322");
res.addCookie(acctId);
```
72. **How a servlet receives cookies sent by a Web browser?**
A. Cookies are packaged inside of the HttpServletRequest object and could be extracted from there as follows:

```java
Cookie[] cookie_jar = request.getCookies();
if (cookie_jar != null) {
    for (int i = 0; i < cookies.length; i++) {
        Cookie aCookie = cookie_jar[i];
        System.out.println("Name : " + aCookie.getName());
        System.out.println("Value: " + aCookie.getValue());
    }
}
```

73. **How can class A notify class B about some important event?**
A. Class A has to implement the Observer interface and class B has to be inherited from the Observable class.

74. **How can an applet exchange data with a servlet?**
A. An applet can exchange data with a servlet using either HTTP or through socket programming. HTTP protocol is easier to program, for example an applet can create an instance of the URL object, open a connection through URLConnection, define an OutputStream pointing at this connection and write some URLEncoded data to this stream.
If an applet needs to receive data from a servlet, it needs to open an InputStream on the URLConnection and read data from there.

75. **How can you ensure that only one instance of some class could be created in your application?**
A. This design pattern is called Singleton and you have to create a class with a private constructor and provide a public get method that will return the only instance of this class, something like `MyClass.getInstance()`.

76. **Name the valid bean scopes in a JSP?**
A. Page, Request, Session, and Application.

77. **How can you reset (clear) a JTable?**
A. You could either delete all the rows from the object holding data in the table model (i.e. Vector), or assign an empty model to the JTable component:

```java
MyTable.setModel(new AbstractTableModel(){
    public int getColumnCount(){return 0;}
    public int getRowCount(){return 0;}
    public Object getValueAt(int row, int col){return null;}
});
```
78***. What’s the major difference between a Hashtable and a HashMap?
A. The Hashtable class is internally synchronized, while the HashMap is not.

79***. What’s the difference in the process of activation and passivation of stateful and entity beans?
A. Activation of an entity bean means assigning values to its state variables. Activation of a stateful bean is a process of return of the bean from a persistence storage if it’s been saved there because of passivation.

80***. We are planning to develop a system in Java, which should feed data to a legacy mainframe system. What Java technology would you recommend?
A. If the mainframe system presently uses or can use messaging, i.e. MQSeries – I’d recommend JMS on the Java side. Some other candidates could be the use of Corba interface, Java Connectors, or simple XML files.

81***. What Java classes have to be installed or downloaded to the client computers? Consider the following scenarios:
a) Java Client talks to an EJB server;
b) Java Appet client talks to a Servlet;
c) HTML client talks to a Servlet.

A. a) The client’s program has to have access to home and remote interfaces, stubs and utility classes, if any that are used as method arguments of the remote interfaces.
b) There is nothing to install on the client’s machine. Utility classes that are used by both – applets and servlets have to be packaged in the archive that will be downloaded to the client’s machine with the applet.
c) This is a no-maintenance scenario – no Java classes are required on the client’s side.

82**. Name some of the design patterns.
A. Singleton, MVC, Value Object, Façade.

83**. Explain the difference in the meaning of the word “stateless” in the session EJB context vs. the HTTP protocol.
A. In case of the HTTP protocol it means that the Web browser does not hold the connection between subsequent user’s requests, while in case of an EJB, it means that the session bean can not be used to store the state of a particular client.

84**. Are Java objects passed by value or by reference?
A. Objects are passed by reference, but the their reference variables are passed by value.

Resources

1. Java certification programs by Sun Microsystems: 
   http://suned.sun.com/US/certification/java

2. Certification tests from Brainbench: 
   http://www.brainbench.com

3. Java Programmer Mock Certification Exam and tutorial by Marcus Green: 
   http://www.jchq.net/

4. Mock exams and tutorial: 
   http://www.javacaps.com

5. Javaranch - mock exams and Java forum: 
   http://www.javaranch.com

6. Article “Looking For a Job In The Greater New York”: 
   http://www.smartdataprocessing.com
Appendix B

Installing WebLogic Application Server

One of the most popular application servers on the market is WebLogic from BEA Systems, Inc. and you could download a free evaluation copy of this product. Code samples in this book were tested using WebLogic version 6.1 with Service Pack 2.

Step 1. Visit the URL http://commerce.bea.com/downloads/products.jsp, complete the registration, select the download option, and follow the instructions. If you are using a Windows platform, the name of the downloaded file could look like this:

weblogic610sp2_win.exe

Step 2. Run this program, select a home directory, for example c:\bea, and specify a name of the root directory (wlserver6.1). During the installation process you’ll accept the administrator’s id system and enter the administrator’s password. Remember the password - you’ll need it to start the server later.

You can find a detailed explanation of the installation process online: http://e-docs.bea.com/wls/docs61/install/index.html

Step 3. Open a command prompt window and set the environment variables by running the command file setEnv.cmd. Start the WebLogic server from the command prompt (run startWebLogic.cmd from the directory c:\bea\wlserver6.1\config\mydomain), or select the menu item Start Default Server. You’ll be prompted for the password that’s the same one as you’ve entered during Step 2 above. If there are no exception messages on the system console, it means that WebLogic is running and listening to the default port 7001.

Step 4. Edit the file startWebLogic.cmd and set the STARTMODE=False to ensure that the server starts in the development mode. This will enable dynamic EJB deployment.
Step 5. Start the server’s console by entering the following URL in your browser: http://localhost/console/. The WebLogic’s console is a GUI screen that allows you to see and modify the properties of various objects of the server. These properties are stored in the file named config.xml, which is located in the directory mydomain.

On the logon screen, enter the user id system and the password from Step 2 above. If your Web browser is using a proxy server to connect to the Internet, you may see an error screen instead of the WebLogic’s console. Turn off the proxy settings in your browser (check the menu Internet Options | Connections | LAN Settings in MS Internet Explorer, or Preferences | Advanced | Proxies in the Netscape Web browser).
Index

A
abstract class, 142, 274
Access Levels, 24
actionPerformed method, 67
ancestor, 15
anonymous inner class, 51
API, 25
applet, 26, 27, 185, 186
appletviewer, 30
arguments, 11, 17
array, 20, 93
ArrayList, 96
AWT, 26, 31
AWT Adapters, 50

B
Bean Activation, 225
Bean Passivation, 225
Bean-Managed Persistence, 243, 244
bean-managed transactions, 227, 229
BitSet, 99
BMP, 243, 249
BorderLayout, 37
break, 22
BufferedInputStream, 64
BufferedOutputStream, 65

C
callback method, 28
CardLayout, 40
Casting, 141, 275
certification, 273
class, 7
CLASSPATH, 4, 276, 277
Cloning, 148
CMP, 231, 243, 250, 253
CMR, 253, 254
command line arguments, 22
comments, 17
commit, 127, 128
constructor, 18, 20
container-managed transactions, 229
Container-Managed Persistence, 243, 244
content pane, 131
continue, 22
Cookie, 159, 160, 276

D
Data Types, 8, 9
Database Connection Pools, 233
DataInputStream, 68
DataOutputStream, 68
Delegation Model, 46
deployment descriptor, 217, 229
descendent, 15
document root, 170
durable subscriber, 265

E
EJB, 209
ejb-jar.xml, 217, 230, 231, 250, 253, 267
EJB-QL, 254
encapsulation, 25
Entity bean, 210, 243
Enumeration, 97
event, 42
event listeners, 42, 45
event-dispatching thread, 132
Exception, 55
extends, 15, 103
Externalizable, 75

F
façade pattern, 243
File, 72
FileDialog, 64
FileInputStream, 63, 64
FileOutputStream, 63
FileReader, 65
FileWriter, 65, 66
final, 24, 280
finalize, 280
finally, 57, 58, 280
FlowLayout, 33
for, 22, 100, 106, 123, 126
Frame, 49

G
garbage collection, 274
Get, 158, 276
global variables, 44, 99
GridLayout, 33

H
HashMap, 97
Hashtable, 96, 97
Hidden Fields, 161
Home Interface, 212
HTTP, 159

I
if statement, 13
IllegalCastException, 95, 141
implements, 46
inheritance, 15
initial context, 203
inner class, 49, 50
instance, 7
instanceOf, 95
instantiate, 12
interface, 42, 43, 44, 45, 75, 274

J
J2EE, 151, 152, 203, 269
JDBC, 117, 118, 119, 275
JDBC Data Source, 234
JDBC driver, 117
JNDI, 207, 260, 262
JSP, 170, 172, 174, 177, 190
JSP Loading, 179
JTable, 130, 133, 134

L
Layout Managers, 32
LDAP, 207
LinkedList, 100
localhost, 180
loopback IP address, 279

M
marshalling, 73
Message acknowledgement, 264
Message Selectors, 266
Message-Driven Bean, 210, 266
MessageListener, 262, 263, 267
META-INF, 166, 231
method main(), 12
method signature, 5
MOM, 257, 258
multiple inheritance, 42
Multithreading, 2
MVC, 133, 182, 190, 278

N
network programming, 76
O

object, 7
ObjectInputStream, 74
ObjectOutputStream, 73
overloading, 19
overriding, 16

P

package access level, 25
panel, 37
PATH, 3
Point-to-Point, 258
polymorphism, 144
POP, 88
port, 78, 84, 199, 282
Post, 158, 276
Primary key, 245
primitive data types, 8
private, 24, 25
Properties, 98
protected, 24
Proxy Servers, 79
public, 5, 11, 24
Publish/Subscribe, 258

Q

Queue, 258, 264

R

race condition, 107
Reflection, 146, 147
Remote Interface, 213
RequestDispatcher, 179
result set, 121, 124
return, 11
RMI, 194, 195, 200
rmiregistry, 197
rollback, 128
Runnable, 104, 132
run-time binding, 143

S

Serializable, 73, 75, 186
serialization, 72, 186, 226
ServerSocket, 83
Servlet, 153, 154, 155, 156, 168, 185
Session bean, 210
Session Tracking, 158, 162
Setting Environment, 3
skeleton, 200
SMTP, 88
socket, 84, 85
Socket, 83
SQL, 117, 118, 126
SQLException, 120
stack trace, 54
Stateful Session Bean, 223
Stateless Session Bean, 211
static, 5
stored procedures, 127
stream, 62, 68
StreamTokenizer, 69, 70
String, 266
StringTokenizer, 71
stub, 199, 200
subclass, 15
super, 23
superclass, 15
Swing, 130
switch, 14, 100

T

TableCellRender, 139
TableModel, 135
Tag Libraries, 182
TCP/IP, 76
this, 19, 46
Thread, 102, 103, 104
throw, 59, 60
Throwable, 59
throws, 57
Topic, 259, 264
transaction, 229
Transaction Attributes, 227
transient, 74
try/catch block, 55

U

URL, 76, 77, 80, 81, 154
URL rewriting, 160, 163
URLConnection, 79
user-defined exceptions, 60

V
Value Object, 225
Variable Scope, 14
variables, 10
Vector, 94, 95, 96, 97, 187
VisualAge for Java, 164, 180
void, 5, 11

W
Web application, 165, 167
Web Server, 78, 153, 159
WEB_INF, 165
WebLogic, 167, 182, 232, 269
WebLogic Installation, 287
WebSphere, 164, 180
while, 22, 80, 81, 97, 119, 120

X
XML, 166, 167, 229, 230