Lab 8: Java Networking

In this lab, you will learn about TCP/IP protocols and Java networking.

Note to the instructor -- before the students do this lab, you need to create a Dynamic Web project named **testWeb** and in its WebContent folder, a simple HTML page named **index.html**. Then right-click on that HTML file and choose *Run - Run on Server*.

Objectives:

- To examine the ICMP and HTTP protocols
- To devise a TCP/IP protocol
- To write a simple HTTP server

Part 1: Investigating ICMP and HTTP

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

Instructor computer IP address: _____

If your computer has the standard lab setup on Windows XP, the directories will be something like:

```
Lab Installation Directory:       c:\javaclass-p2
```

On Windows Vista or later:

```
Lab Installation Directory:       c:\Users\username\javaclass-p2
```

_2. To get started, open a command prompt window by choosing *Run* from the Windows Start menu and typing:

```
cmd
```

Then press Enter.

_3. Now let's determine your computer's configuration for TCP/IP. Enter the following into your command prompt window:

```
ipconfig
```

This Windows command displays the configuration of the network card bound to TCP/IP on your

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

computer. The key points are your computer's IP adress, the *subnet mask* that configures your local network and the *default gateway*, which is the device that routes TCP/IP packets from the local network eventually to the Internet (if you're connected to the Internet).

Note the name of the connnection (e.g. *Ethernet adapter Local Area Connection)*.

Now type:

```
ipconfig /all
```

Find the detailed information for the connection displayed in the previous step. Can you determine the Domain Name Service (DNS) server(s) for your computer?

_4. The first protocol you will investigate is the *ICMP* protocol used by the "ping" program.

In your command prompt window, enter the following command:

```
ping xx.xx.xx.xx              - use the Instructor's IP address
```

The ping program is useful for network trouble-shooting. It sends ICMP packets to the destination computer, which is supposed to send a reply. If you don't get a reply, then you know that either you used an incorrect IP address or the network is down or congested.

The Windows ping program sends four requests and for each, reports the number of milliseconds the request took as well as the packet's *time to live* (TTL). The TTL shows the number of gateways or routers a packet is allowed to go through before being destroyed to avoid looping. Each intermediary decrements the TTL when it relays the packet. Generally, when a host answers the ping, it sets the return packet's TTL to 255. Subtract the TTL as shown from 255 to get the number of gateways and routers crossed by the packet. This number is sometimes referred to as *hops* -- i.e. the number of hops is how many gateways the packet needed to pass through.

_5. Use ping to send ICMP requests to *www.google.com*. What is Google's IP address? How many hops did it take?

Note: If your lab computer is behind a firewall, the ping program may not work.

_6. Now let's use another useful trouble-shooting program. In your command prompt window, enter:

```
tracert xx.xx.xx.xx                     - Use the Instructor's IP address
```

The tracert program shows you the routers/gateways that a packet traverses to reach the destination. It's useful for identifying bottlenecks in the route from your computer to the destination. It displays the hop number, the number of milliseconds for each of three attempts and the router/ gateway's DNS info.

Try using tracert with www.google.com. Again, be aware that if your lab computer is behind a firewall, this might not work.

_7. The *telnet* protocol lets a user on a computer with a TCP/IP stack log into and execute commands on a remote computer. Most modern operating systems, including Windows, ship with a Telnet client program.

Though telnet is useful in its own right (though not very secure), it also allows us to investigate other protocols -- that's what you will use it for in this lab.

To ensure that telnet is installed, enter at the command line:

```
telnet
```


Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

If that worked, you can enter **quit** to terminate telnet and skip the rest of this step.

If the above command fails with "telnet is not recognized...", and your computer is running Windows 7 or later, you will need to install telnet using the following procedure:

a. From the Windows Start menu, go to the Control Panel.

b. Under *Programs and Features*, choose **Turn Windows Features on or off**.

c. Find *Telnet Client* in the list and put a checkmark next to it, then press OK and wait for telnet to be installed.

_8. The instructor has a simple Web server running on his/her computer. To see how the HTTP protocol works, you can use telnet. Enter the following into your command prompt window:

```
telnet xx.xx.xx.xx 9080              - Use Instructor's IP address
```

You should see a blank screen -- you will enter text in a moment.

The telnet command accepts the IP address or hostname of the remote computer and the port number on which to send the request.

Telnet usually works on port 23, but in this case, the instructor's computer's Web server is listening on port 9080 (the usual HTTP port is 80).

Complete the following steps to send a sample HTTP request:

a. Into the blank screen, type:

```
GET /testWeb HTTP/1.0
```

Note: You will not see what you are typing, so be careful! Be sure to use the correct case of characters.

Warning: You will need to type fairly quickly or else the request will fail with a timeout error. You might find it easier to enter the above text into Notepad and then copy and paste from there.

b. Press the Enter key twice.

You should see a display that looks something like:

```
HTTP/1.0 302 Found
X-Powered-By: Servlet/3.0
Location: http://localhost:9080/testWeb/
Content-Language: en-US
Content-Length: 0
Date: Tue, 12 Jun 2012 19:36:05 GMT
Server: WebSphere Application Server/8.0


Connection to host lost.
```

This response is a good example of how the HTTP protocol works. The first line shows the version of HTTP on the server and the "302" is an HTTP response code (more on that in a moment). Most of the rest of the message is made up of *HTTP response headers* which give information about the response. Note that *Content-Length* is zero -- this response essentially returned no data.

c. HTTP response code 302 means that the requested resource is available at another address as

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

indicated in the *Location* response header. Note that the Location header shows the full URL of the new location including the protocol (HTTP), the server address, the port number following the colon and the *resource location*.

Then use telnet again and re-enter the GET command with the full path of a file on the server.

```
GET /testWeb/index.html HTTP/1.0
```

Be sure to specify HTTP/1.0 and press the Enter key twice. You should see a display like:

```
HTTP/1.0 200 OK
X-Powered-By: Servlet/3.0
Date: Wed, 13 Jun 2012 14:12:01 GMT
Content-Type: text/html
Last-Modified: Tue, 12 Jun 2012 19:22:46 GMT
Content-Length: 263
Content-Language: en-US
Server: WebSphere Application Server/8.0

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.
g/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

</body>
</html>

Connection to host lost.
```

This is a valid HTTP response as indicated by the 200 response code, which means OK. Note the *Content-length* header now shows a nonzero value and that following the last response header, the actual HTML content is passed. Can you make any sense of the HTML?

Open Internet Explorer from the Windows Start menu and enter this URL:

```
http://xx.xx.xx.xx:9080/testWeb/index.html     - Use Instructor's IP addres
```

You should see a minimal (blank) Web page.

Use Internet Explorer's View - Source menu to examine the HTML source for the page. Does this make sense given the HTTP response you looked at earlier?

d. Now let's try fetching a non-existent page. Use telnet and the GET command to retrieve the resource */testWeb/xyz*. Can you interpret the result? What's the HTTP response code for "resource not found"?

e. Repeat the above, but use Internet Explorer. Does the display make sense?

_9. If your computer has non-firewalled Internet access, use telnet to retrieve Google's home page at www.google.com, port 80 (use / in the GET command). Then browse to Google's home page in Internet Explorer and compare the results.

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 2: Devising a Protocol

In this lab, you will write a client-server application that uses sockets. The server program will report the amount of total and free memory in its JVM. The client will retrieve this information and display it to the console.

Steps:

_1. Create a new Java project named **Lab08**.

_2. Create a class named **FreeMemServer** in the *part2* package. The class should have a *main* method.

In *main*, open a server socket on some port, then enter an infinite loop. In the loop, call the *accept* method to listen for client requests and recieve a transfer socket. Then query the amount of total and free memory in the JVM and send these Strings back via a BufferedWriter and the socket back to the client. Here are some hints:

- After accept returns the transfer socket, "wrap" its output stream in a BufferedWriter
- To query the total and free memory, examine the *java.lang.Runtime* class
- Be sure to put a new line between the Strings (e.g. call *newLine()* on the Writer)
- Be sure to close the writer and the transfer socket after each request
- It's a good idea to print diagnostic messages to the console so that you can see the server's activity when you run it

_3. Create a class named **FreeMemClient** in the *part2* package with a *main* method.

This client should connect to the FreeMemServer using the server's hostname and the same port that you used in the server. The client should retrieve the two Strings from the server using a BufferedReader and display the Strings to the console and then exit. Here are some hints:

- Retrieve the server's hostname as a command-line argument -- use *localhost* if no command-line argument is provided
- Be sure to close the Reader and the socket before exiting

_4. To run the application, first run the server as a Java application, then run the client. You will need to edit the *Run configuration* for the client and provide the command-line argument of *localhost*. You should then see the memory-usage strings.

Hint: In the RAD console, you can switch between the console output of multiple programs by clicking the drop-down arrow on the "computer" icon on the console titlebar:

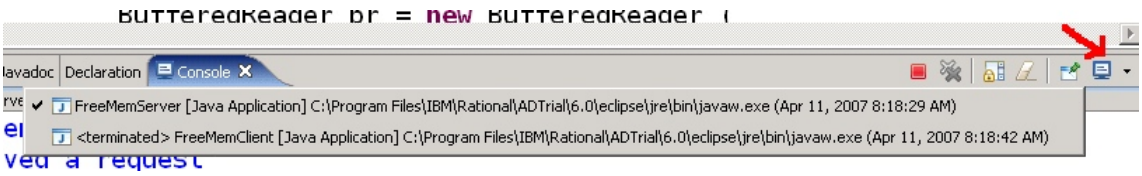


Figure 1

_5. If the classroom hardware supports it, try running the client and servers on different computers. The client will need to specify the hostname of the server as a command-line argument.

Reinforcement: Writing a Simple HTTP Server

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

In this part, you will practice the concepts you learned in lecture and in the first two parts of lab by writing a server that implements part of the HTTP protocol. You will not attempt to write a full implementation, since that is too large of undertaking for a lab. Here are the specifications for your HTTP server:

- The server should continuously listen on the standard HTTP port
- The server should only process *GET* requests for simple files returning *HTTP/1.1 200 OK*
- The server should return *HTTP/1.1 404 Not Found* for non-existent files
- The server should return *400 Bad Request* for all HTTP methods other than *GET*
- The server should return the response headers shown in the 200 OK example from Part 1 of this lab
- Be sure to wait for a blank line after the GET request and to insert a blank line between the response headers and response body
- Be sure to close the Writer and the request socket after satisfying each request

Steps:

_1. Create two simple HTML files in the *Lab08* root folder: **file1.html** and **file2.html**. These are the files that your HTTP server will serve.

Note: We recommend that you use the *Source* tab at the bottom of RAD's HTML editor so you can enter "raw" HTML into these files.

_2. Create a class named **MyHttpServer** in the *reinforce* package with a *main* method.

In *main*, implement the HTTP server as described above.

_3. To test, run your server, then use a standard Web browser to browse to *http://localhost/file1.html*. Then try testing with a non-existent file URL.

If you are having problems using the Web browser, try using the telnet method you used in the first part of lab to diagnose problems.

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 9: Remote Method Invocation

In this lab, you will practice using Java's RMI facility.

Objectives:

- To write a remoteable object and RMI server
- To write an RMI client
- To configure the RMI runtime infrastructure

Part 1: Writing an RMI Server

In this part, you will write an RMI server. The server will publish the date and time of a fictional meeting. In the next part, you will write a client that will retrieve the info and display it.

Steps:

_1. Create a new Java project named **Lab09**.

_2. Create an interface named **Meeting** in the *server* package, and in it, write an RMI remote interface that contains two methods: one to retrieve a String that contains the meeting date, and another that returns a String containing the meeting time. Neither method needs any input parameters but should follow the rules for RMI remotable methods -- i.e. the interface should extend *java.rmi.Remote* and each method should throw the *java.rmi.RemoteException*.

_3. Create a class named **MeetingServer** that extends *UnicastRemoteObject* and implements the *Meeting* interface in the *server* package with a *main* method. Note that you will need to define an appropriate zero-argument constructor to get rid of compile errors. The *main* method should perform the following:

   a. Set the *RMISecurityManager* so that the server can download classes for serialized objects.

   b. Create an instance of the MeetingServer class.

   c. Construct a String that contains the object's name. The String should be of the form *rmi://xxxxx/Meeting* where xxxxx is your computer's hostname. Hint: Use the *InetAddress* class to determine the hostname.

   d. Bind the instance into the RMI namespace using the name.

   Note: We recommend that you write System.out.println calls after each step in main. That will make it easier to debug and will allow you to see the server's progress as it initializes. It's especially helpful to print the object's name so you can see if you constructed it correctly.

_4. Implement the two remoteable methods by returning hard-coded Strings for a date and a time.

_5. Next, you need to write a *policy file* that lets your server access the RMI socket. While you could write a specific policy file, for now, write one that grants all privileges. Right-click on the Lab09 project and choose *New - File* to create a file named **serverpolicy**, and in it, enter:

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
grant
{
    permission java.security.AllPermission;
};
```

Note: Be sure to enter the trailing semicolon!

_6. Open a command-prompt window and change to the lab directory, then establish the Java environment:

a. Use the Windows Start menu: choose **Run** and enter *cmd* to open a command-prompt window.

b. In the command-prompt window, change to the *Lab09* directory. Assuming the standard lab setup on Windows XP:

```
cd \javaclass-p2\workspace\Lab09\bin
```

On Windows Vista or later:

```
cd \Users\username\javaclass-p2\workspace\Lab09\bin
```

Note: If your workspace is not in the **javaclass-p2\workspace** folder, adjust the above command as necessary.

c. Set the system PATH to reference a JDK:

```
set PATH="\Program Files\IBM\SDP\jdk\bin";%PATH%
```

_7. Start the RMI Registry in its own window:

```
start rmiregistry
```

You can minimize the registry's window since it doesn't display anything interesting, but leave it running for subsequent steps.

_8. Gentlemen and ladies: Start your server:

```
java -Djava.security.policy=../serverpolicy server.MeetingServer
```

You should see any diagnostic messages provided by your server. Leave the server and the RMI registry running for later steps.

Part 2: Writing an RMI Client

Steps:

_1. In the *Lab09* project, create a new class named **MeetingClient** in the *client* package with a *main* method.

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Complete the *main* method so that it:

a. Sets the security manager to the *RMISecurityManager* so that the client can download stubs and serialized classes from the server.

b. Looks up the remote object using the its registered name.

c. Casts the reference to the correct type.

d. Calls both methods on the remote interface and prints the results to the console.

_2. Copy the *serverpolicy* file and rename the copy to **clientpolicy**.

_3. Run the client program - it should fail with a security violation since it isn't yet using the policy file.

_4. To fix the runtime error, edit the client's *Run Configuration* (from the main menu, choose *Run - Run Configurations*) that specifies the client's class. In the Run Configuration, click the *Arguments* tab and set the *VM arguments* to **-Djava.security.policy=clientpolicy**.

Then run the client -- do you see the date and time Strings?

Part 3: Using Multiple Physical Computers

In this part, you will configure the program to work across multiple physical computers.

Note: This lab will work ONLY if you and your partner coded the RMI interface in EXACTLY the same way.

Steps:

_1. Find a partner, preferably sitting near you, who has already completed the previous part. One partner should act as the client, the other, as the server.

_2. Modify the client program so it doesn't use *localhost* for the remote object name. Instead, it should use the hostname or IP address. Look back at the Networking lab if you cannot remember how to determine the hostname or IP address.

_3. Ensure that the rmiregistry is running on both machines.

_4. Test the program.

Part 4: Reinforcement

In this part, you will create a client/server based application using RMI with minimal guidance from this write up.

Here's the scenario. You need to create a distributed "chat" application that lets participants insult each other. You should create the following remoteable interfaces:

- **InsultCoordinator** This interface defines the behavior of a distributed coordinator that maintains a String array of participant names. The names are of the the form *objName@hostname*. This interface should define methods to get the participant list and to add and remove participants.
- **InsultReceiver** This interface defines the behavior of each participant -- this interface should define a method that allows a participant to receive an insult String from some other participant.

You can define the following classes, each with a *main* method.

- **InsultCoordinatorServer** This class should maintain the list of participants and should register itself into the RMI registry.
- **Participant** This class should implement *InsultReceiver* and accept the Participant's name as a command-line argument. It then should register itself with the RMI registry and with the coordinator. The

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

class should then enter into an infinite loop. Each time through the loop, display a list of participants and let the user choose one. Then let the user enter an insult, at which point this class should look up the participant from the RMI registry and then send the insult. Any received insults should also be displayed to the console. For extra credit, cache each participant in a hash table so you don't do the RMI lookup more than once.

Steps:

_1. Switch back to the Java perspective if you're not already there.

_2. Create a new package: **reinforcement** and implement the program as described above.

_3. To test, run the InsultCoordinatorServer in its own command-prompt window and run each participant in a separate command-prompt window.

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 10: XML and XML Programming

In this lab, you will write some well-formed XML and write Java programs that process XML.

Objectives:

- To use RAD to create XML content
- To parse XML using the DOM API
- To parse XML using the SAX API

Part 1: Converting a Text Document into XML

One of the benefits of XML is that it lets you define content that can be transformed into various forms. In this lab, the fictional Cogswell Company would like to make their travel policy guidelines available on their intranet as well as printed in an employee handbook.

In this part, you will convert a text document that describes the travel policy for Cogswell to XML.

Steps:

_1. Switch RAD to the Java perspective if not already in it.

You can also stop the RMI server and registry from the last lab if you've not already done so.

_2. Since RAD contains so many features, some of its "advanced" capabilities are disabled by default, so as not to overwhelm new users. To enable XML development, follow this procedure:

a. Choose *Window - Preferences* and expand the *Workbench - General* category. Then highlight the *Capabilities* entry.

b. Press the *Enable All* button.

Then press Apply followed by OK.

_3. Create a new Java project named **Lab10**.

_4. Next, import all of the XML and text files from the *starters/Lab10* folder into the project.

The *policy.txt* file contains the "raw" travel policy that you will convert to well-formed XML.

_5. Open the *policy.txt* file into an editor by double-clicking on it.

Note the structure of the document: it contains a title, a set of definitions and then the set of policy rules that govern travel for Cogswell employees. Your first job is to visualize the document as a tree of content -- that will make the transition to XML easier. Sketch the tree structure of the document below:

_6. Copy the file to an XML file using this procedure:

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. In the Package Explorer, right-click on *policy.txt* and choose **Copy**.

b. Right-click on the *Lab10* project and choose **Paste** -- enter **policy.xml** for the name of the new file.

c. Double-click on the XML file to open it and press the *Source* tab at the bottom of the editor window.

_7. Your job is to add XML markup to the file so it becomes self-describing data.

Start by adding the *XML declaration* at the top of the file.

As you add markup, you can press Ctrl+S to save the file and then in the Package Explorer, right-click on the XML file and choose *Validate*. This will cause RAD to check it for well-formdness and show any problems in the *Problems* pane.

_8. Using your tree sketch from above, separate the text content into XML elements. Be sure to close each element! Here are some hints:

- The entire document should be contained in a **policy** element
- It's a good idea to create *aggregating elements* that hold child elements. For example, it would be good to create an enclosing **definitions** element that contains **definition** elements and a **rules** element enclosing **rule** elements. Remember that it's OK and common to have more than one element with the same name!
- You can write a **heading** element as a child of the **policy** element that contains the text *General Travel Policy* -- that will give you something to display as the title if you were to style the XML with CSS. Use similar logic for the **definitions** and **rules** content.

Note: We recommend that you indent the elements and text to make the XML more readable.

_9. When you are finished, you should have no errors in the Problemss pane.

You should also be able to see the tree structure of your XML by looking in the *Outline* pane. If you don't see the Outline view, you can display it by choosing *Window - Show View - Outline*.

Part 2: Converting a Scanned Form into XML

In this part, you will convert a paper-based purchase-order form into well-formed XML content.

Steps:

_1. Create a new XML file:

a. Right click on *Lab10* in the Package Explorer and choose **New - Other - XML - XML**.

b. On the first wizard page, select *Create XML file from scratch*, then press Next.

c. On the next wizard page, enter **po.xml** for the *File name*, then press Finish.

_2. In the new file, create the XML content for the scanned PO form shown here.

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Purchase Order

PO#QM2345

Issue Date: Oct 10, 2000
Valid 30 days from Issue

To:

Computer Excitement
75 Sasbro Road
Etobicoke, Ontario
CANADA M2X 3B3
Attn: Jack Wong
Phone: 416-323-3554
FAX: 416-323-2888

From

Magic Widget Development Inc.
23 Bandana Road
Carlsdale, Ontario
CANADA M5B 3M9
Attn: Joe Sharpe, Purchasing Dept
Phone: 416-343-3333 X 2345
FAX: 416-343-3324

Quantity	SKU	Description	Unit Price	Extended
3	MO3452	Mag 20 inch Monitor	495.00	1485.00
2	KB1102	Ergonomic Keyboard	99.00	198.00
			PST	117.81
			GST	134.64
			TOTAL	1935.45

Please ship all items to: *Magic Widget Research Labs.*
1543 Northern Highway
Littleton, OH
USA 32111
Attn: Tracy Boyton, Imaging Dept
Phone: 512-234-3333 X 2345
FAX: 512-234-3324

Figure 1: Purchase Order Form

_3. Save the file and make sure that it's well formed. Correct any mistakes before you continue.

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 3: DOM Programming

In this part, you will write a DOM program that parses XML that represents an online exam. Your program will perform some simple data processing on the content.

Steps:

_1. Examine the *exam.xml* file that you imported earlier and take a look at the XML file. We recommend that you draw a picture that illustrates the tree structure of the content's elements.

_2. Create a new Java class named **ExamSummaryDOM** in the *part3* package. The new class should have a *main* method.

_3. One useful technique for processing a DOM tree is to use a *recursion* loop that visits each node in the tree. We have provided you with a method that implements recursion in the **{Lab Installation Directory}/Lab10/starters/starters.txt** file. Copy and paste the *processNode* method into the new class.

Examine the provided code and note how it calls itself (recursion) and how it checks to ensure it doesn't create an infinite loop.

_4. Complete the *main* method so that it uses DOM to parse the XML and displays the following output:

- The exam name, date and professor
- A summary that contains the highest score, lowest score and average score

Here are some hints:

a. Use the JAXP API to instantiate a DOM parser.

b. Parse the *exam.xml* file.

c. Call the *processNode* method, passing the *Document* node. You will complete the *processNode* method in the next step.

_5. The real work of the program takes place in the *processNode* method. Here are some hints to help you get started:

a. After the *work with "current" node here* comment, write a test to ensure that the current node is an element node.

If it is, retrieve the element's *node name*. If the name is *date*, display the exam's date. Here is some code to help you get started:

```
if (currNode.getNodeType() == Node.ELEMENT_NODE)
{
    if (currNode.getNodeName().equals("date"))
    {
        Node txtNode = currNode.getFirstChild();
        String s = txtNode.getNodeValue();
        System.out.println("Exam Date: " + s);
    }
}
```

b. We recommend that you run the program at this point to test it to see if it works so far.

c. In a similar fashion, complete the *processNode* method so that it implements all of the requirements listed above.

_6. The final results should look something like:

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Exam Date: May 10, 2004
Exam Title: XML 101 Midterm
Professor: I.M. Mean
High score: 98
Low score: 50
Average score: 78
```

Part 4: SAX Programming

In this part, you will repeat the previous part, but use the SAX API instead of DOM.

Steps:

_1. Create a new class named **ExamSummarySAX** in the *part4* package with a *main* method.

_2. Create a private, static, inner class named **MyContentHandler** that extends *org.xml.sax.helpers.DefaultHandler*. This inner class should not have a *main* method.

_3. Complete the *main* method so that it:

- Uses the JAXP API to create a SAX parser and retrieve an *XMLReader*
- Creates an instance of the content-handler inner class and registers it with the XMLReader
- Parses the *exam.xml* content

_4. Complete the inner class by overriding appropriate methods from DefaultHandler so that it generates the same output as the DOM program.

Note: Since the Exam XML content contains no namespaces, your *startElement* method should examine the *qName* argument instead of the *localName* argument.

Reinforcement

In this part, you will practice the techniques you learned in earlier parts to implement a program with minimal instruction from this lab guide.

We have provided you with some XML content that represents an order from an online Web store. The content is in the *order.xml* file in the starters folder. Please examine it and make sure you understand it.

Your job is to:

- Parse the XML using either DOM or SAX and create a set of *domain* objects that represent the content. You will need to write the classes for the domain objects (e.g. *Order*, *LineItem* and so forth. The domain classes should be in a package named *order*.
- Each domain class should define a *toString* method that prints a nicely formatted display of the object's data.

Steps:

_1. Import the *order.xml* file from the starters folder into the project's root folder.

_2. Create a new class named **ProcessOrder** in the *reinforce* package with a *main* method. In main, instantiate a parser and start the parsing process.

Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_3. Create the domain classes in the *order* package. The domain classes should use relationships to reflect the structure of the XML content (e.g. the *Order* class might maintain a collection of *LineItems*).

Note: The domain classes should define properties of appropriate types (e.g. *int* for *quantity*) and should have get/set methods for the properties.

_4. Using either DOM or SAX techniques, parse the XML and create domain objects and initialize them from the XML content.

_5. Call the toString method(s) to display formatted output, which should look something like:

```
Order date:May 22, 2004
PO Number: 123
Customer info:
        First name: Paul
        Last name: Westerberg
        Address 1: 123 Elm St
        Address 2: null
        City: Minneapolis
        State: MN
        Zip: 45933
Line Item Info:
        SKU: 3740222
        Description: Hootenany
        Quantity: 1
        Price: 12.55
Line Item Info:
        SKU: 3740223
        Description: Let it Be
        Quantity: 3
        Price: 12.55
Line Item Info:
        SKU: 3740234
        Description: Pleased to Meet Me
        Quantity: 2
        Price: 10.99
```


Copyright © Descriptor Systems 2011. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.