# *Seidenberg Institute for Computing Innovation 2011*

# *Lab Manual*



**Dr. Lixin Tao and Dr. Li-Chiou Chen**
**http://seidenberg.pace.edu**
**Pace University**

**5/15/2011**

PDF Version: http://community.seidenberg.pace.edu/files/sici2011/SICI-2011-lab-manual.pdf

SICI Home: http://community.seidenberg.pace.edu/innovation

# Table of Contents


1 Installing *VMware Player* ..... 1
2 Setting Up a Basic *Ubuntu* Virtual Machine ..... 4
2.1 Creating a Ubuntu Virtual Machine ..... 4
2.2 Creating Shared Folder C ..... 10
2.3 Sharing Files by Drag and Drop ..... 14
2.4 Creating Password for Administration Account *root* ..... 15
2.5 Updating *apt-get* Package Information ..... 15
2.6 Automatic Login and Screen Saver ..... 16
2.6.1 Enable Automatic Login ..... 16
2.6.2 Disable Screen Saver ..... 17
2.7 Shutting down or Restarting Ubuntu ..... 18
3 Linux Commands Used in SICI 2011 Labs ..... 19
4 Installing Applications on a Basic *Ubuntu* Virtual Machine ..... 20
4.1 Overview ..... 20
4.2 Launching the VM ..... 20
4.3 Running Command *sudo* without Needing Password ..... 22
4.4 Adding Menu Item “Open in Terminal” to File Browser Popup Menu ..... 23
4.5 Installing 7z ..... 24
4.6 Installing Java JDK and NetBeans ..... 24
4.6.1 Installing Java JDK ..... 24
4.6.2 Installing JRE Plugin for Firefox ..... 30
4.6.1 Testing Java Commands **javac** and **java** ..... 33
4.6.2 Testing NetBeans v7 ..... 34
4.7 Installing Apache Tomcat ..... 38
4.7.1 Installing Tomcat ..... 38
4.7.2 Tomcat File Organization ..... 40
4.8 Installing *MySQL* Database Server ..... 41
4.9 Deploying Sample Tomcat Web Applications ..... 42
4.10 Installing *Apache* Web Server and PHP ..... 45
4.10.1 Installing Apache 2 ..... 45
4.10.2 Installing PHP ..... 47
4.11 Installing *Eclipse* ..... 48
4.11.1 Eclipse Setup ..... 48
4.11.2 Developing a Sample Java Program ..... 48
4.12 Installing Glassfish Application Server v 3.1 and Oracle’s Java EE Tutorials ..... 54
4.12.1 Adding GlassFish and Tomcat Servers to NetBeans ..... 54
4.12.2 Installing Java EE Tutorials through Java Update Tool ..... 59
4.13 Setting Up a Drupal Website on Apache ..... 60
4.14 Networking between Host and VM ..... 74
5 Getting Ready for the Other SICI 2011 Labs ..... 76
5.1 Setting Up for the “Introduction to Web Services” Lab ..... 76
5.2 Setting Up for the “Introduction to Cryptography” Lab ..... 76
5.3 Setting Up for the “Introduction to Web Technologies” Lab ..... 77
5.4 Setting Up for the “Introduction to Java Security” Lab ..... 77
6 Introduction to Web Technologies ..... 78
6.1 Concepts ..... 78

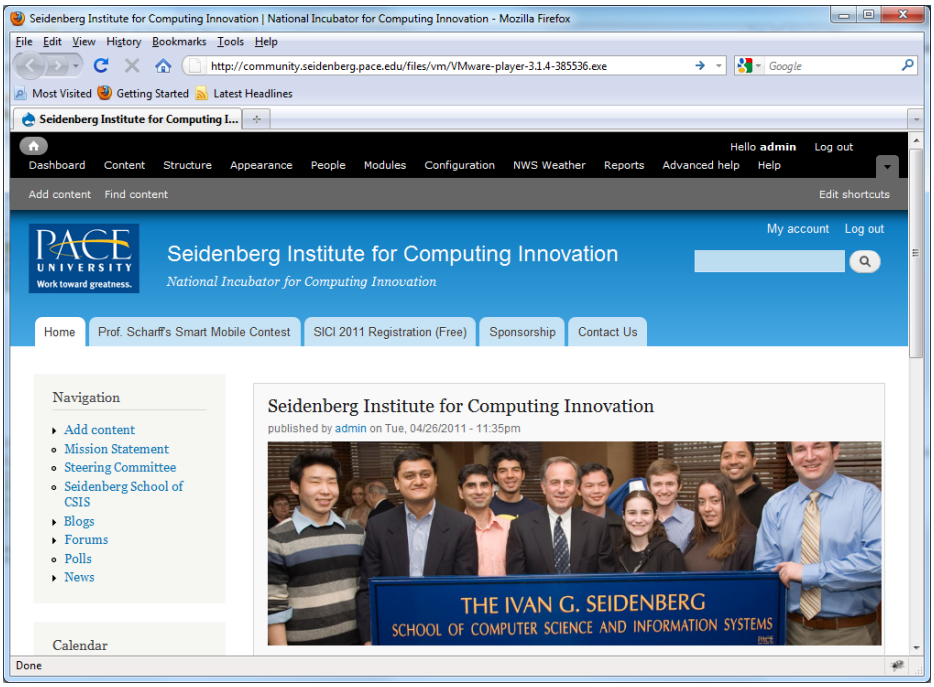
6.1.1 Web Architecture ..... 78
6.1.2 Uniform Resource Locators (URL) ..... 80
6.1.3 HTML Basics ..... 81
6.1.4 HTTP Protocol ..... 84
6.1.5 Session Data Management ..... 86
6.2 Lab Objectives ..... 88
6.3 Lab Guide ..... 88
6.3.1 Comparing HTTP GET and HTTP POST Requests ..... 88
6.3.2 Observing HTTP Communications with Paros ..... 89
6.3.3 Working with Cookies ..... 91
6.3.4 Submitting Data with HTML Form and Hyperlink ..... 92
6.3.5 Validating Form Data with JavaScript ..... 92
6.3.6 Creating Your First JavaServer Page Web Application ..... 94
6.3.7 Creating Your First Servlet Web Application ..... 96
6.4 Review Questions ..... 100
7 Introduction to Web Services ..... 101
7.1 Concepts ..... 101
7.1.1 Challenges that Web Services Address ..... 101
7.1.2 Fundamental Concepts of Web Services ..... 103
7.1.3 Major Components of Web Services ..... 105
7.1.4 Web Service's Role in System Integration ..... 106
7.1.5 Web Service Security ..... 107
7.2 Lab Objectives ..... 107
7.3 Lab Guide ..... 107
7.3.1 Development of a Java Web Service for Squaring an Integer ..... 107
7.3.2 Development of a Java Client Application for Consuming the Local Java Web Service 111
7.3.3 Running Web Service Client and Server on a LAN ..... 113
7.3.4 Developing a Java Client Application to Consume a Public Remote ASP .NET Web Service 115
7.4 Review Projects ..... 118
8 Introduction to Cryptography ..... 119
8.1 Concepts ..... 119
8.1.1 Symmetric Secret Key Ciphers ..... 119
8.1.2 Public Key Ciphers ..... 119
8.1.3 Hash Function and Digital Signature ..... 120
8.1.4 Digital Certificates ..... 122
8.2 Lab Objectives ..... 123
8.3 Lab Guide ..... 123
8.3.1 Hashing Files with MD5 and SHA-1 ..... 123
8.3.2 Symmetric Key Encryption/Decryption with GPG ..... 125
8.3.3 Public/Private Key Creation and Encryption/Decryption ..... 127
8.4 Review Questions ..... 130
9 Introduction to Java Security ..... 131
9.1 Concepts ..... 131
9.1.1 Basic Terms ..... 131
9.1.2 Java Security Framework ..... 133
9.1.3 Key Management ..... 134
9.2 Lab Objectives ..... 135
9.3 Lab Guide ..... 135
9.3.1 Reviewing Java Security Framework ..... 135

9.3.2 Creating Public/Private Keys and Digital Certificates ..... 136
9.3.3 Ensuring Data Confidentiality with Cryptography ..... 139
9.3.4 Securing File Exchange with Java Security Utilities ..... 139
9.3.5 Granting Special Rights to Applets Based on Code Location ..... 141
9.3.6 Granting Special Rights to Applets Based on Code Signing ..... 149
9.3.7 Creating a Certificate Chain to Implement a Trust Chain ..... 156
9.3.8 Protecting Your Computer from Insecure Java Applications ..... 161
9.3.9 Securing File Exchange with Java Security API and Newly Created Keys ..... 162
9.3.10 Securing File Exchange with Java Security API and Keys in Files ..... 165
9.3.11 Securing File Exchange with Java Security API and Keys in a Keystore ..... 166
9.4 Review Questions ..... 167
9.5 Appendix ..... 168
9.5.1 File “GetProperties.java” ..... 168
9.5.2 File “WriteFile.java” ..... 168
9.5.3 File “appletWrite.html” ..... 169
9.5.4 File “appletWrite2.html” ..... 169
9.5.5 File “my.java.policy” ..... 169
9.5.6 File “GenerateSignature.java” ..... 169
9.5.7 File “VerifySignature.java” ..... 170
9.5.8 File “GenerateKeys.java” ..... 171
9.5.9 File “GenerateSignature2.java” ..... 172
9.5.10 File “GenerateSignature2.java” ..... 173
9.5.11 File “GenerateSignature3.java” ..... 173
9.5.12 File “VerifySignature2.java” ..... 174
10 Acknowledgement ..... 176

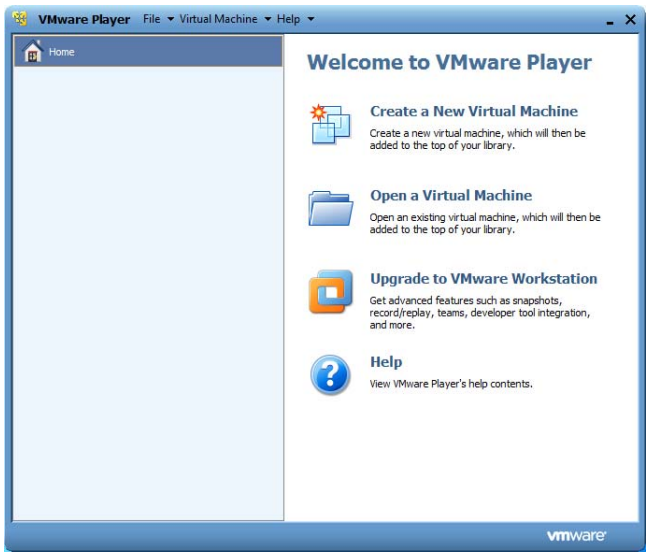
# 1 Installing *VMware Player*

*VMware Player* is a free utility for creating and running *VMware Virtual Machines* (VMs). It can run multiple VMs concurrently therefore supporting a computer cluster on a single PC. *VMware Fusion* is the counterpart of *VMware Player* for *MacBook* but you need to buy a license to use it (around $80). There are *VMware Player* versions for both Linux and Windows, and you can download them directly from http://www.vmware.com/download/player/. Here we explain how to set up *VMware Player* on a Windows PC using the latest copy of *VMware Player* that we have prepared for you.

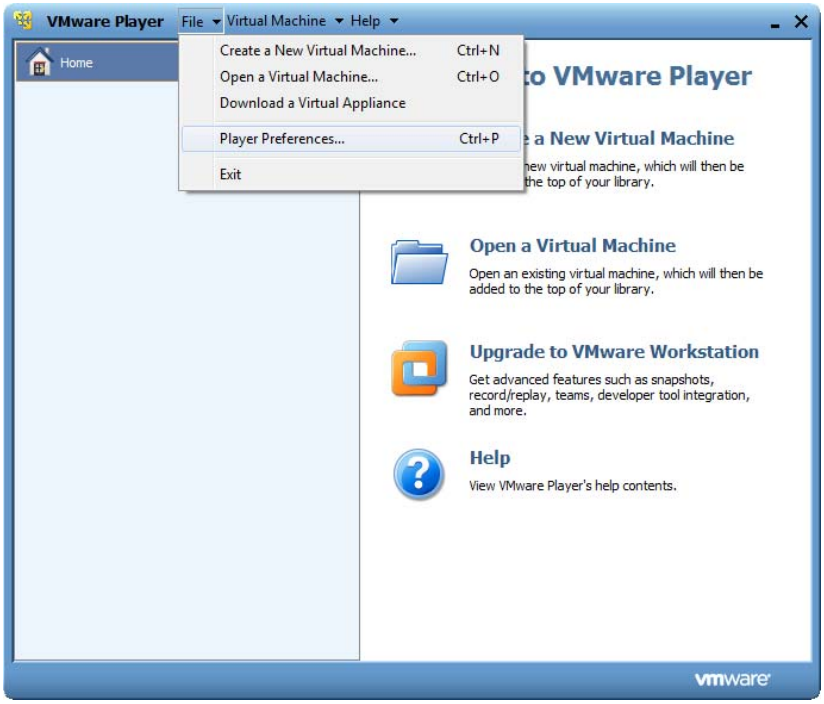
1. If you have installed *VMware Player* versions earlier than V3.1, uninstall it and reboot your PC.
2. Use a web browser to visit http://community.seidenberg.pace.edu/files/vm/VMware-player-3.1.4-385536.exe and save file "VMware-player-3.1.4-385536.exe" on your PC.



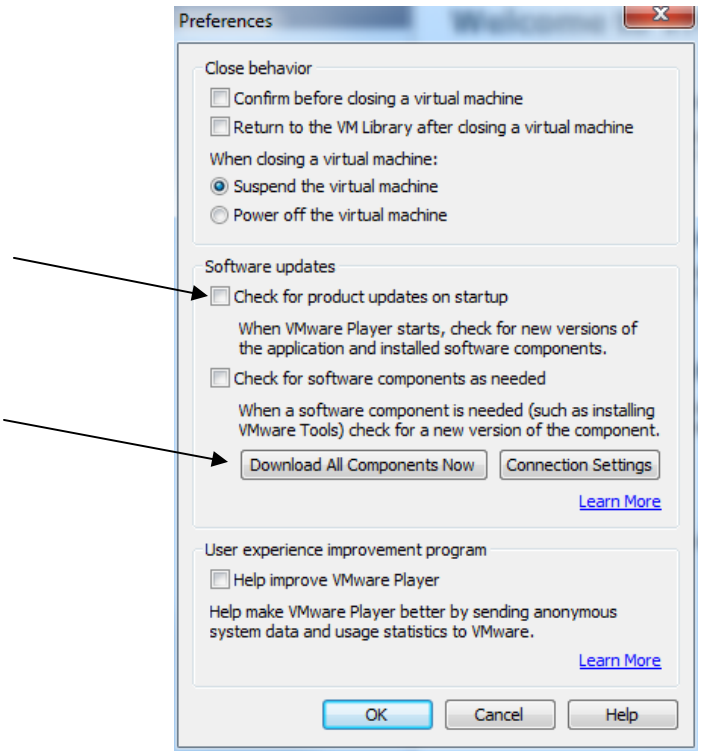

3. In *Windows Explorer*, double-click file "VMware-player-3.1.4-385536.exe" to install *VMware Player* with default values.
4. Reboot your PC.
5. Start *VMware Player*, and you will see a window like the following.

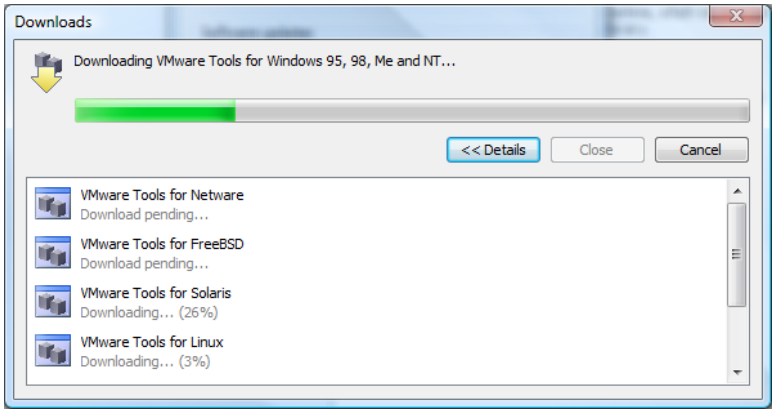



6. Click on menu item “File|Player Preferences…”




7. In the “Preferences” window, uncheck for software updates, and click on the “Download All Components Now” button so you can later install *VMware Tools* in your new VMs without Internet access. This step is optional.




8. With this same “Preferences” window you can also set close behavior (when you click on the VM’s close icon, should the VM suspend or power off the VM, and whether you need to provide a confirmation), and whether *VMware Player* should download all available optional components now or on demand.
9. Click on the *OK* button to close the “Player Preferences” window.
10. Click on the “File|Exit” menu item to exit the “VMware Player” application.

If you are using this tutorial to learn how to install *Ubuntu* and applications on a *Ubuntu* VM, then *VMware Player* is the only software that you need to install on your PC. Your software installation on the VM will have no impact on your PC’s work environment.

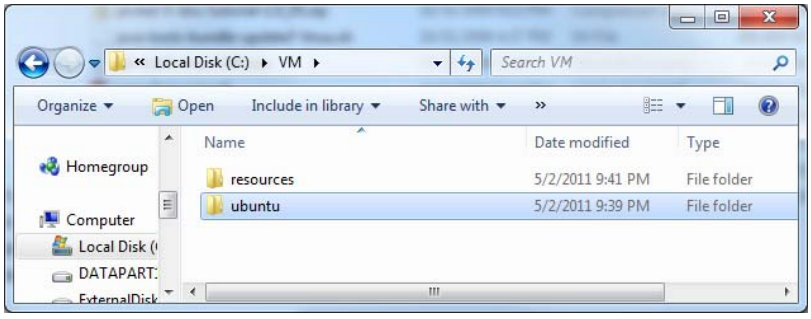
# 2 Setting Up a Basic *Ubuntu* Virtual Machine

In this section we create a basic *Ubuntu* V10.10 virtual machine using *VMware Player* v3.1.4.

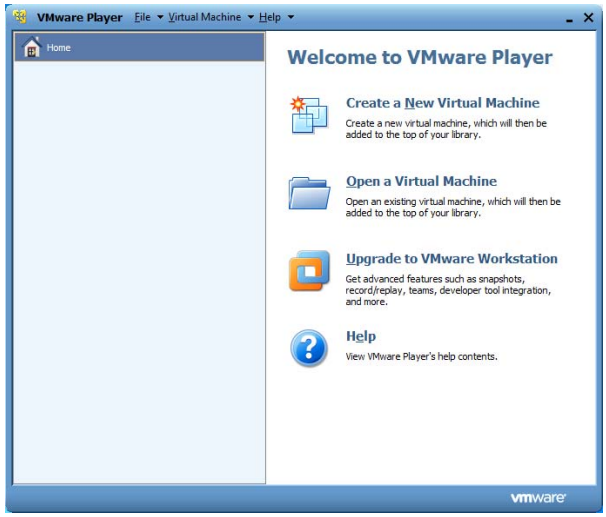
For brevity, when we say to *click on menu item "A|B"*, we mean clicking on menu "A" to see its popup menu, and then further clicking on menu item "B" on the popup menu. When we say to *run "command"*, we mean to start a terminal window (one way is to use *Ubuntu* menu item "Applications|Accessories|Terminal"), and type *command* in that terminal window followed by an *Enter* key, to run the *command*. When we say to *visit "http://url"*, we mean to use a web browser to visit the web page with URL (Uniform Resource Locator) *http://url*. When we say to *click/double-click on a file or folder*, we mean to click/double-click the file or folder in a file explorer.

## 2.1 Creating a Ubuntu Virtual Machine

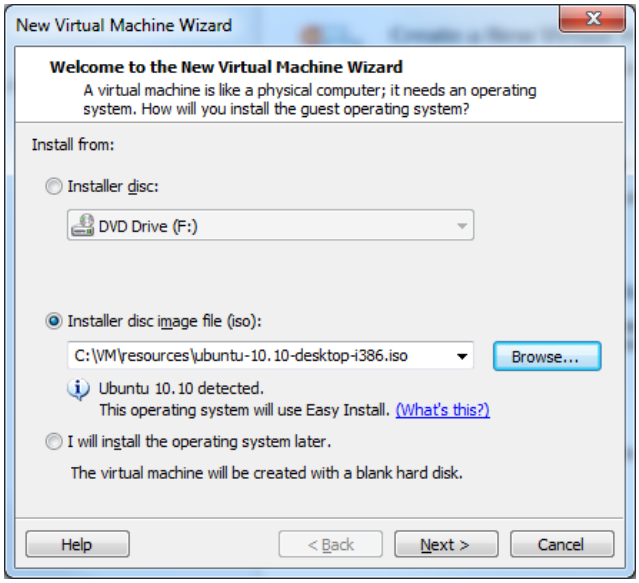
1. In your PC, create a folder "C:\VM" with *Windows Explorer*.
2. Use a web browser to download file "resources.exe" at http://community.seidenberg.pace.edu/files/sici2011/resources.exe into your PC's folder "C:\VM".
3. Double click the downloaded file "resources.exe" to extract folder "C:\VM\resources". You will find the latest *Ubuntu* installer "ubuntu-10.10-desktop-i386.iso" is in this folder.
4. Create a new folder "C:\VM\ubuntu" as the folder for the new VM.



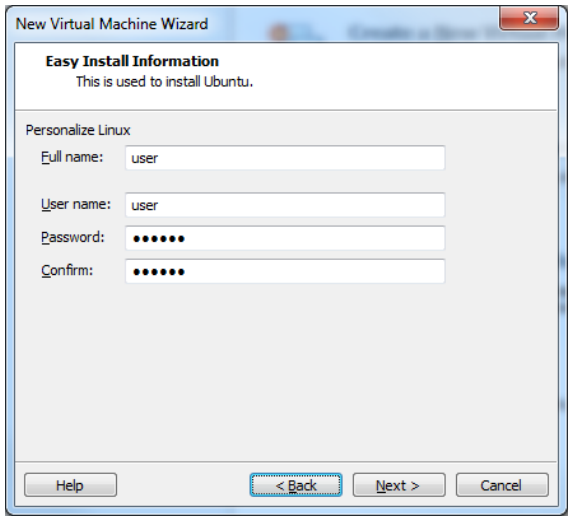

5. Launch *VMware Player*.



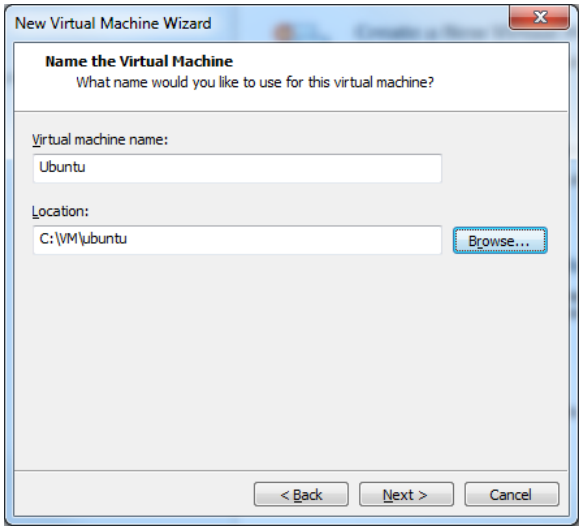

6. Click the link for “Create a New Virtual Machine”.



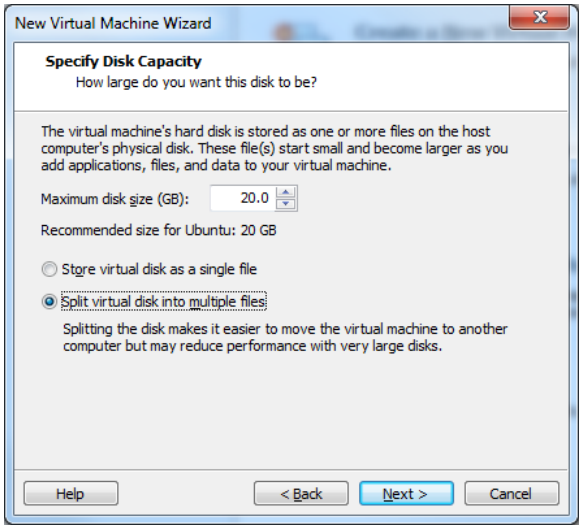

7. Check for “Installer disc image file (iso)”, and browse for your downloaded *Ubuntu* v10.10 ISO file (C:\VM\resources\ubuntu-10.10-desktop-i386.iso). Click the *Next* button.



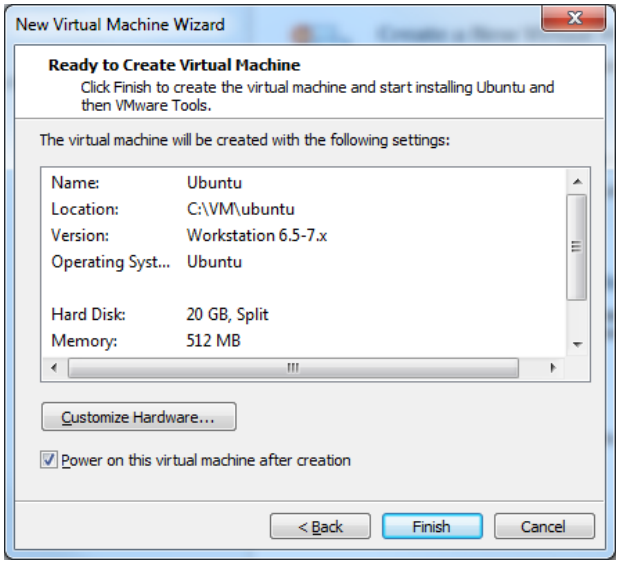

8. In the popup “New Virtual Machine Wizard” window, enter “user” for full name and user name, and 123456 for password and confirmation. Then click the *Next* button.



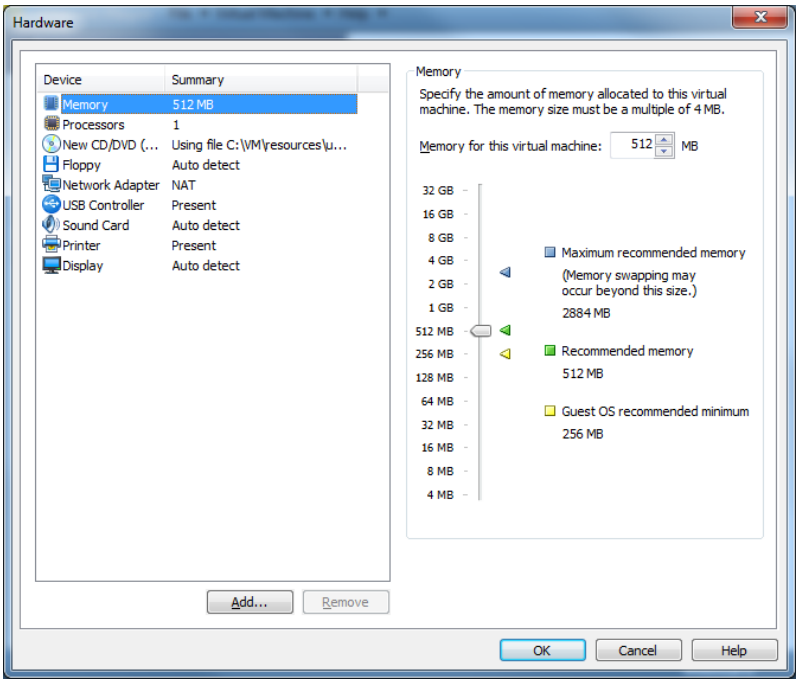

9. Use “Ubuntu” as the virtual machine name, and “C:\VM\ubuntu” as the location value. Then click the *Next* button.




10. Enter 20 for maximum disk size, and check for "Split virtual disk into multiple files" so you could later easily save the VM on DVD disks. Then click the *Next* button.

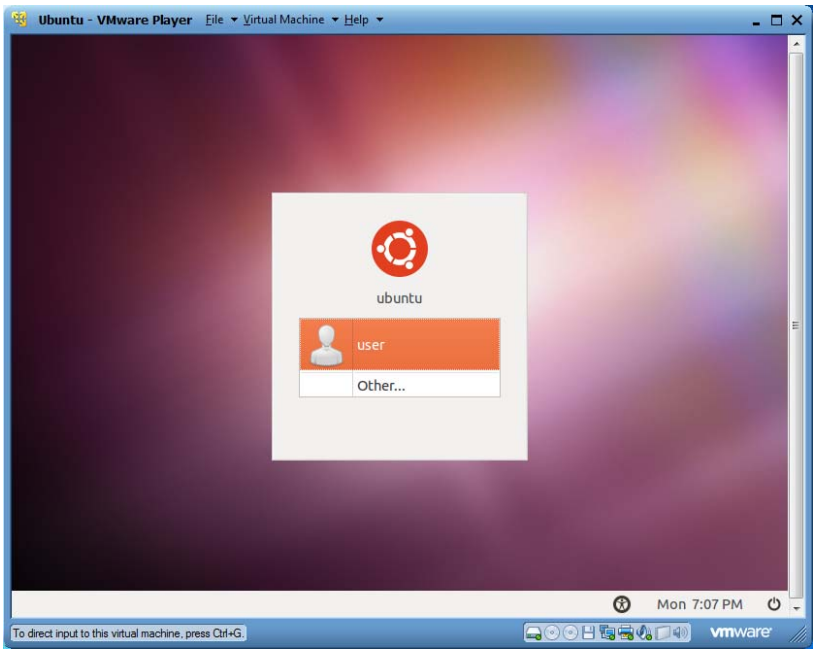



11. Review the settings for the new VM. Click the "Customize hardware…" button to review what you can further customize. Notice that by default the new VM uses 512 MB physical memory. Since your PC's OS also needs physical memory, your PC needs to have one GB memory to work well. If you have more than one GB memory, you could increase the VM's memory size so it could work smoother. If you plan to run multiple VMs at the same time, make sure the total memory used by these VMs and your PC would not exceed your available physical memory size. You may let a Linux VM work with only 256 MB memory.

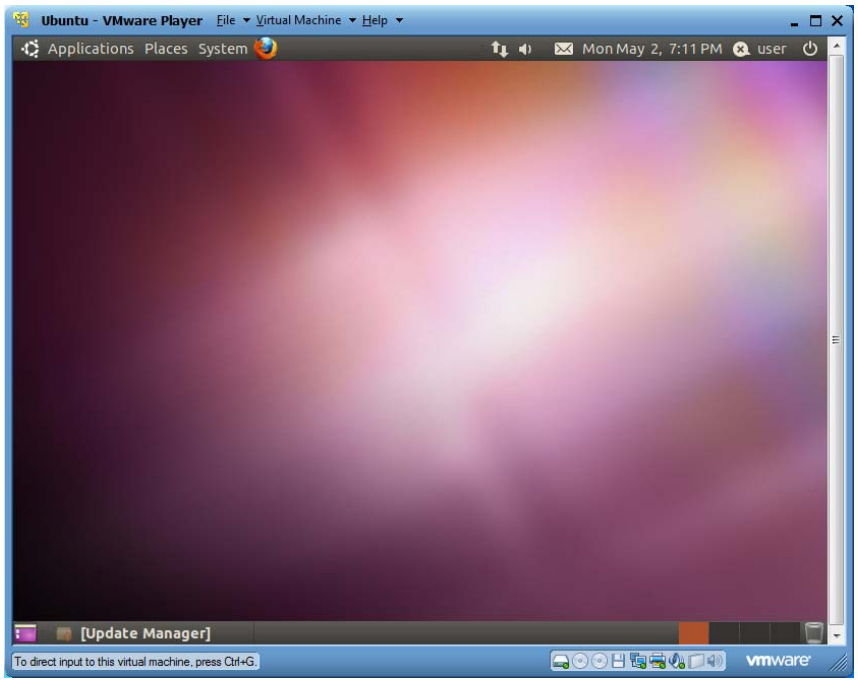



12. Click the *OK* button to exit the "Hardware" window. Click the *Finish* button to start the Ubuntu installation process. The installation may take 15 minutes.
13. If you see that the *VMware Player* is installing *VMware Tools* and let you login to use the new VM at the same time, you don't need to do anything. Just wait.

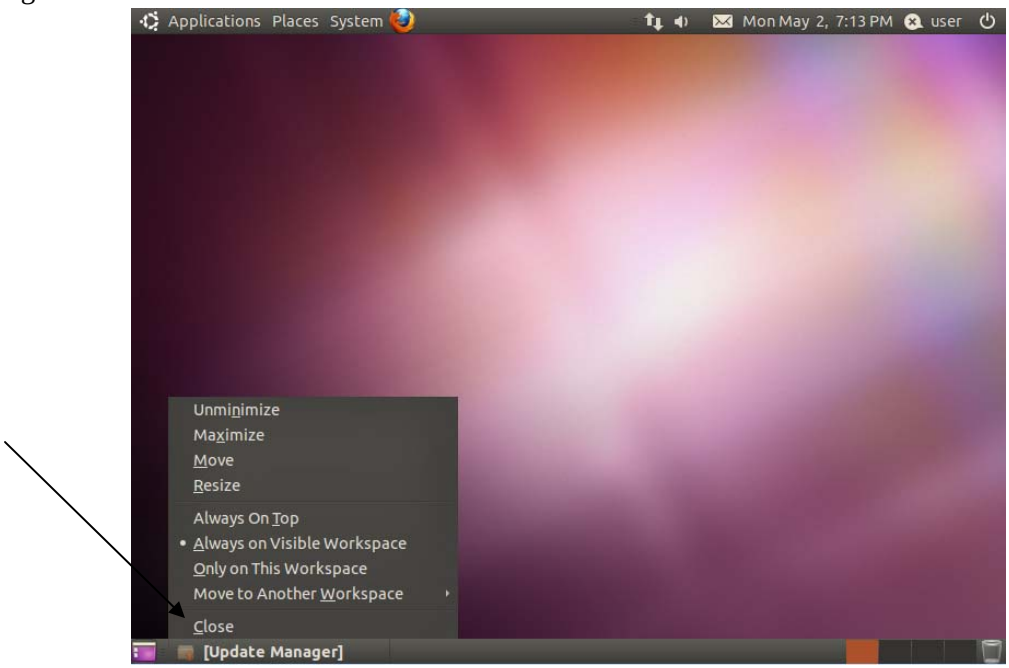

14. When the installation is complete you will see the following screen:




15. Click “user”, and enter 123456 as password. You will see the following screen. You are now logged in your new VM.

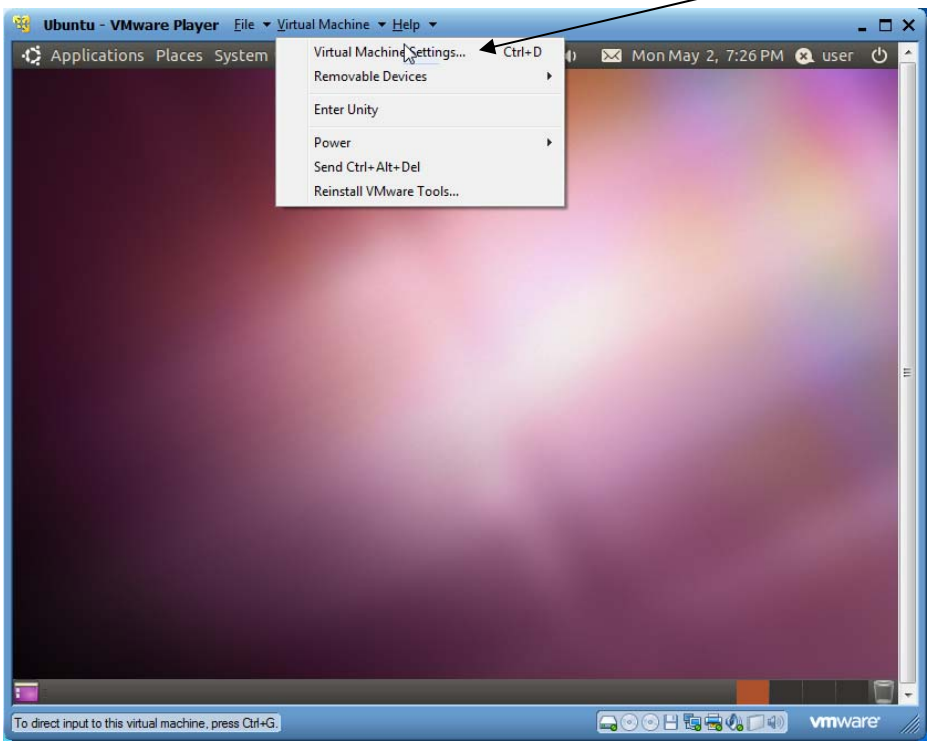



16. When asked whether to upgrade your Ubuntu 11.04 installation, click the button for “Don’t Upgrade”. You should do the update when you are working at home.
17. To save the lab time, you will not perform system update at this time. Right click the bottom-left icon for “Update Manager” and choose “Close” to shut down the update manager.

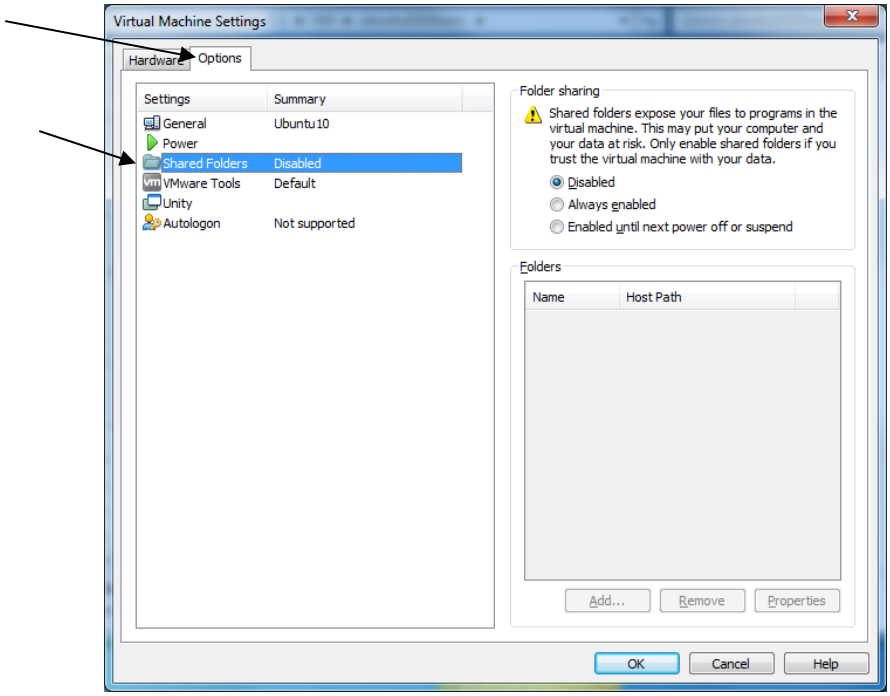



## 2.2 Creating Shared Folder C

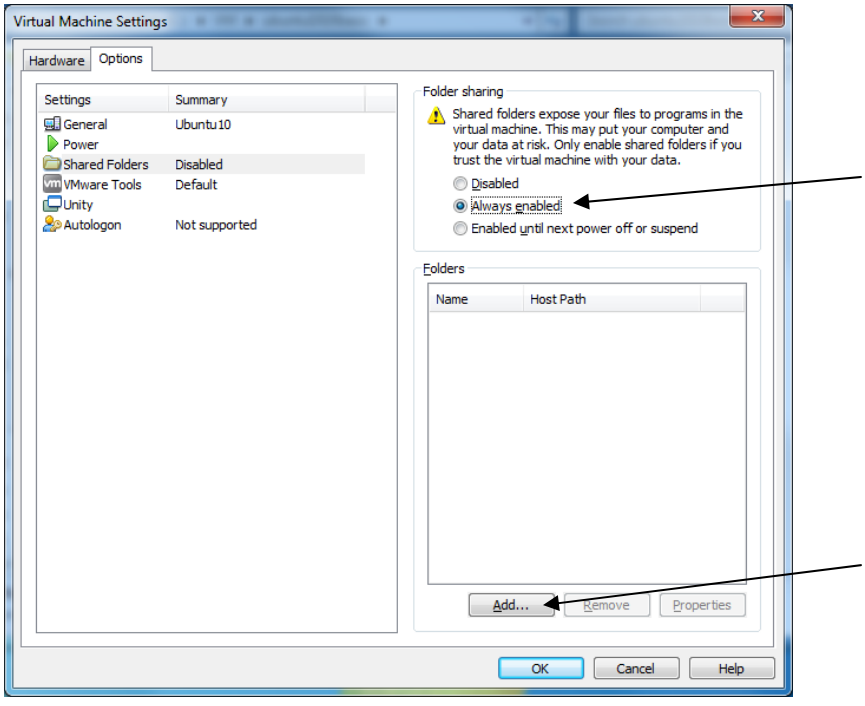
18. Click menu item “Virtual Machine|Virtual Machine Settings…”.




19. Click the “Options” tab of the “Virtual machine Settings”, and then select “Shared Folders”

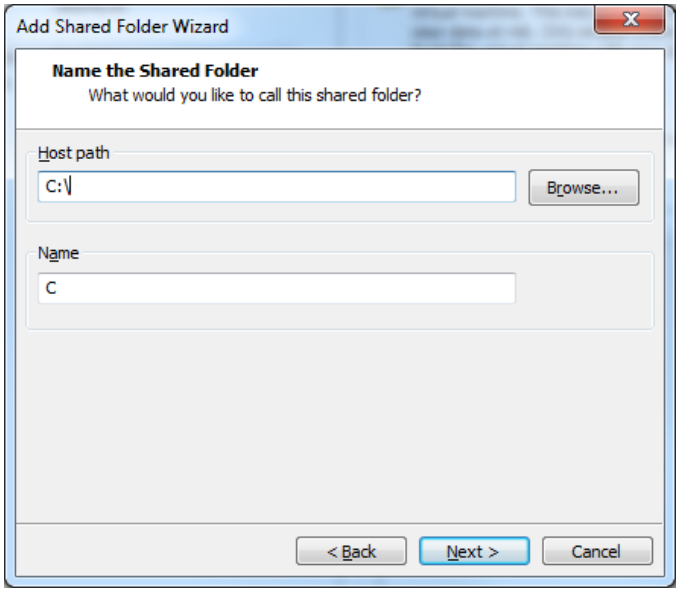



20. On the right side, check the “Always enabled” checkbox, and click the “Add…” button.

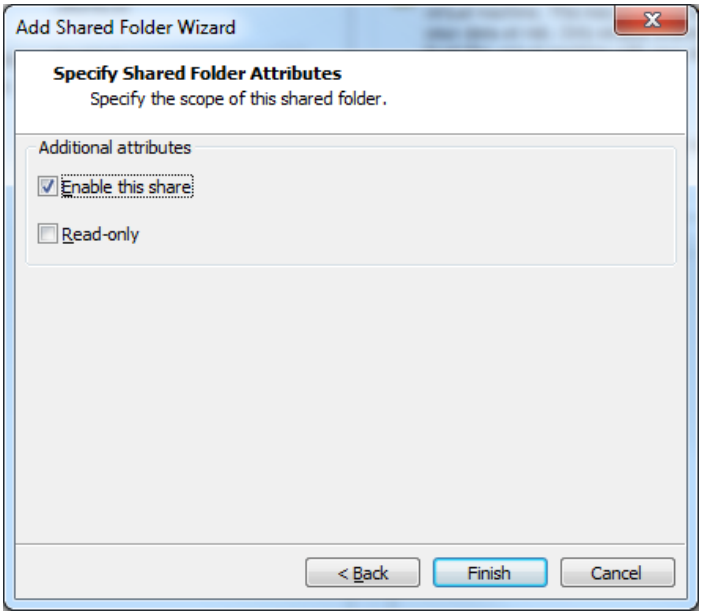



21. You will see the following window.

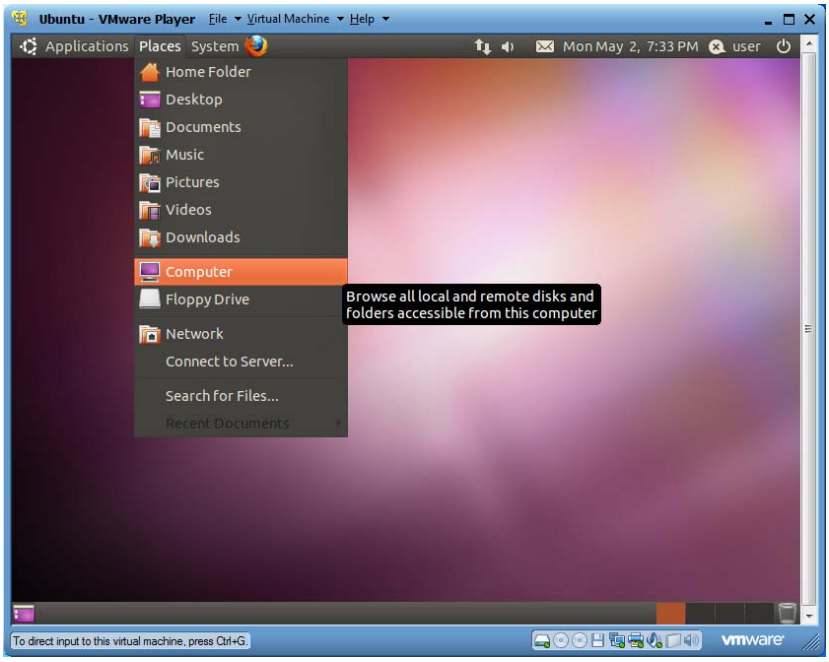

22. Click the *Next* button. Enter "C:\" for *Host Path*, and "C" for *Name*.



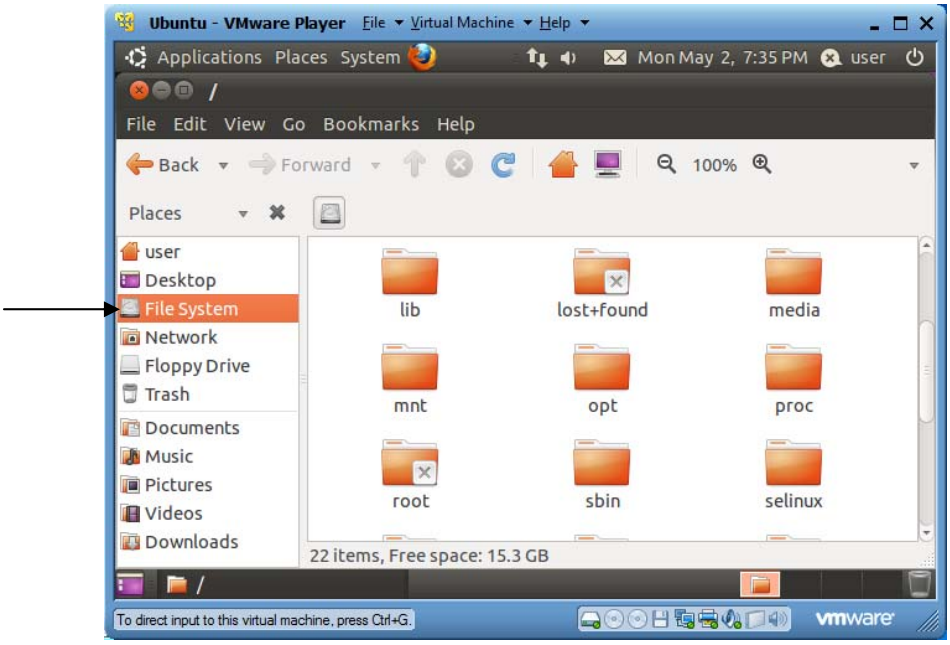

23. Click *the Next* button.



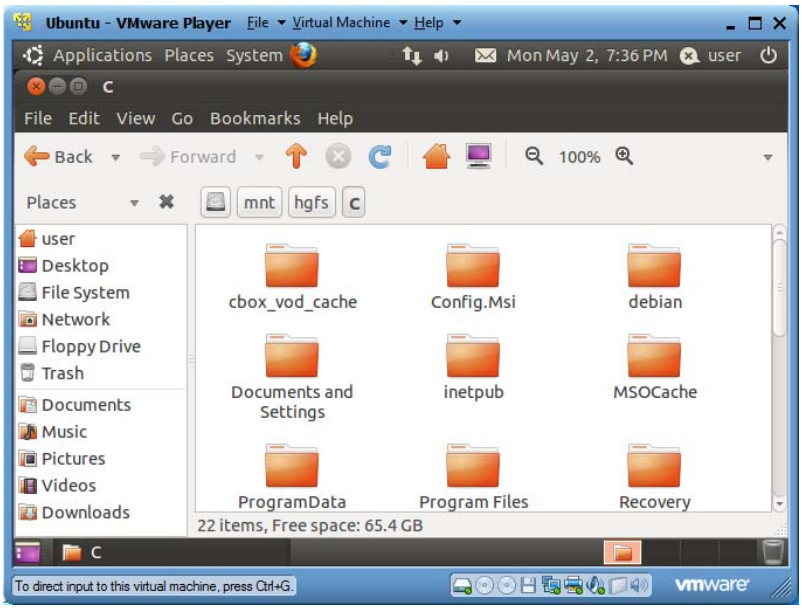

24. Make sure “Enable this share” is checked. Click the *Finish* button to shut down the “Add Shared Folder Wizard”.
25. Click the *OK* button to shut down the “Virtual Machine Settings” window.
26. Now you are ready to check out the shared folder. Click menu item “Places|Computer”.




27. Click “File System” in the left pane.




28. Click *mnt*, then *hgfs*, and then *C* to see all your files and folders on your PC's drive "C:\".



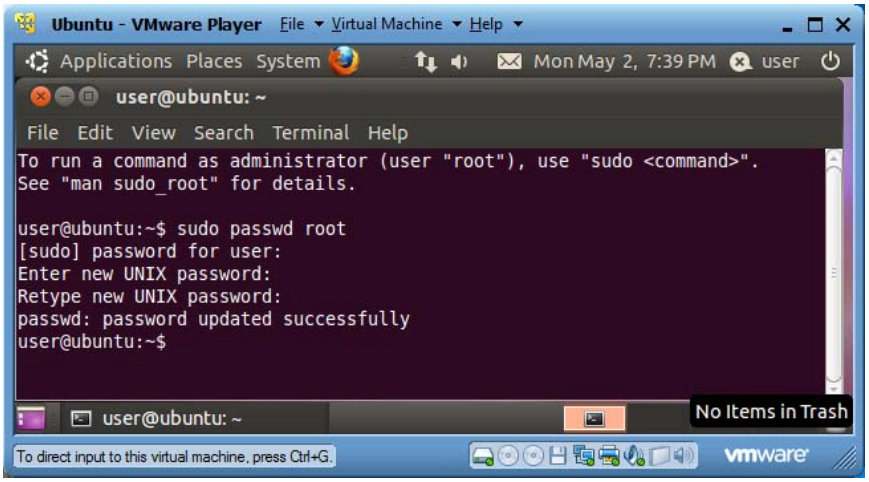

29. Congratulations and your shared folder is working. You can now use the shared folder to transfer files between the VM and your physical PC. It is faster and more reliable than drag-and-drop.

## 2.3 Sharing Files by Drag and Drop

*VMware Tools* also allows you to copy files between your PC and your VM by drag and drop. Try to copy a file from drive C or desktop of your PC to your VM's desktop, and then copy the file back to a different location of your PC.

## 2.4 Creating Password for Administration Account *root*

For security reason, by default *Ubuntu* doesn't create password for super user *root*. We'd like to create a password for *root*. Click menu item "Applications|Accessories|Terminal" to launch a terminal window. In the terminal window, run "sudo passwd root". When asked for your password, enter 123456. Then type 123456 twice as *root*'s password.




Now you have a working Ubuntu V10.10 VM. In the next section you will learn how to install various tools and servers in this VM so it can function as a powerful learning tool for all kinds of information technologies.

## 2.5 Updating *apt-get* Package Information

You will mainly use Linux utility apt-get to install utilities and applications. You need to download the latest information on the available apt-get packages. We use this chance to illustrate how to login as root.

1. Run "su -" to request to work as super user (root). The symbol - indicates that the following session will use *root*'s environment.
2. Enter your ("user") password 123456.
3. Now you notice that the command prompt symbol has changed from $ to #. This means you are working as *root*.
4. Run "apt-get update" to update the apt-get information on available packages to install.
5. Run "logout" to exit the super user session.

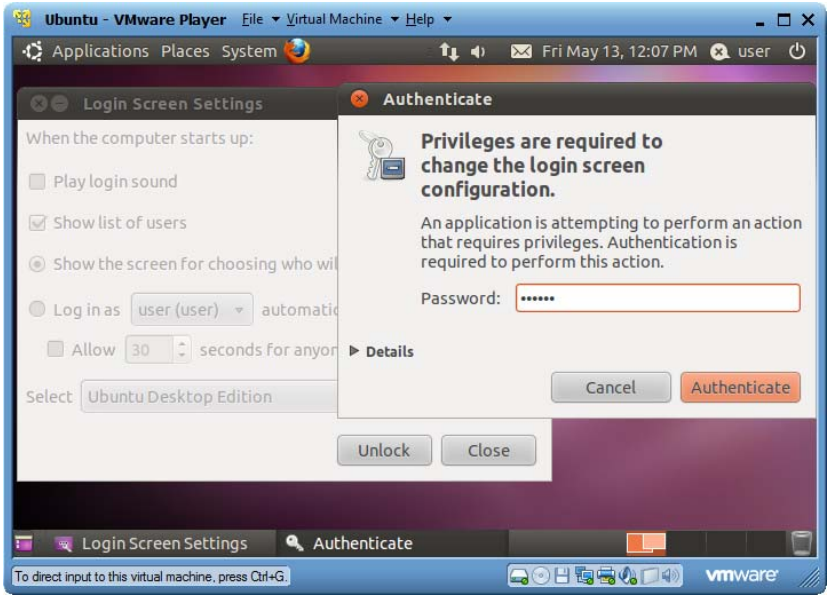
```
user@ubuntu:~$ su -
Password:
root@ubuntu:~# apt-get update
Hit http://us.archive.ubuntu.com maverick Release.gpg
Ign http://us.archive.ubuntu.com/ubuntu/ maverick/main Translation-
en
……
root@ubuntu:~# logout
user@ubuntu:~$
```

## 2.6 Automatic Login and Screen Saver

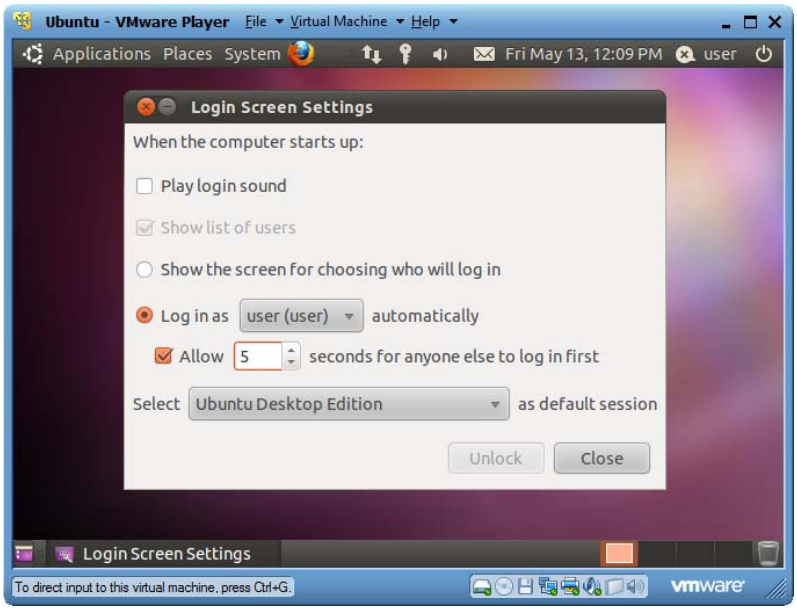
Since you will work on this VM for extended period of time, it is convenient to enable automatic login and disable the screen saver.

### *2.6.1 Enable Automatic Login*

1. Use menu item “System|Administration|Login Screen” to launch the “Login Screen Settings” window.


2. Click the *Unlock* button and then enter your password 123456. Click the *Authenticate* button.

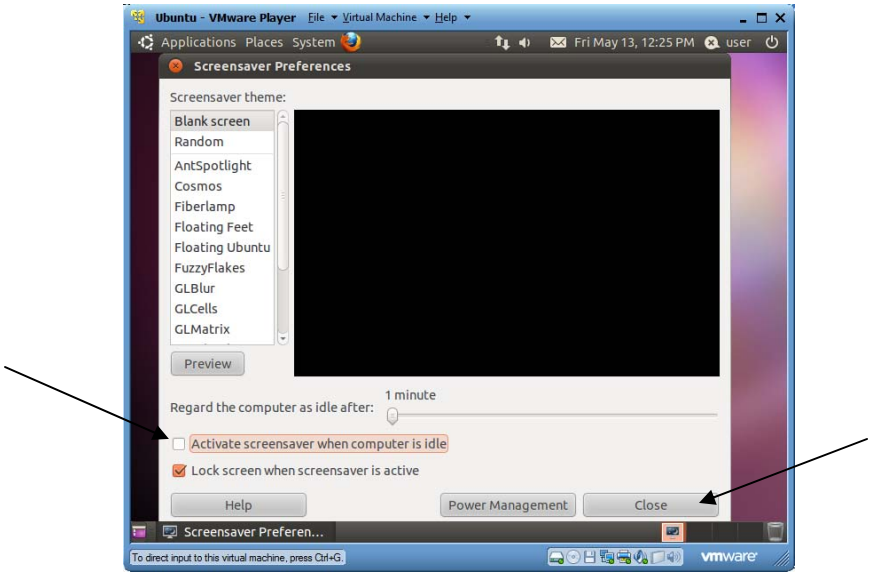



3. Check “Log in as *user* automatically”, and check and choose 5 for “Allow 5 seconds …”. Now you have 5 seconds to login as another user before the VM automatically login as *user*.
4. Click *Close* to complete the procedure.

### *2.6.2 Disable Screen Saver*

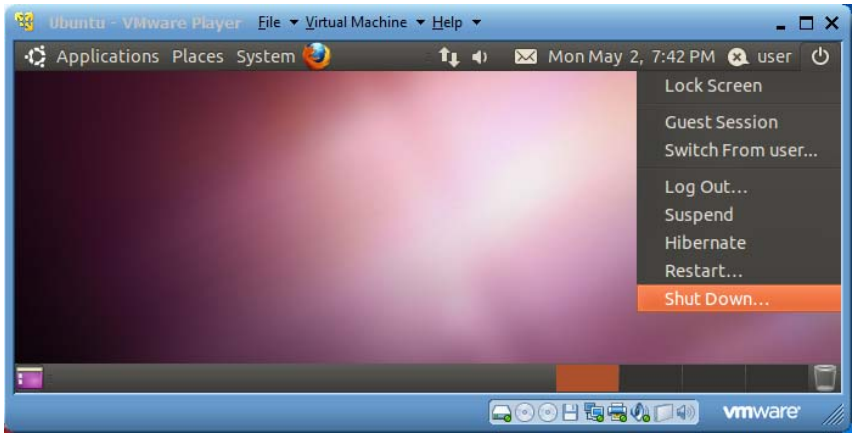
1. Use menu item “System|Preferences|Screen Saver” to launch the “Screen Saver” window.


2. Uncheck “Activate screensaver when computer is idle”, and then click *Close* to complete the procedure.




## 2.7 Shutting down or Restarting Ubuntu

To shut down or restart Ubuntu, click the upper-right icon  and then choose the right item on the pop-up menu as shown below.



# 3 Linux Commands Used in SICI 2011 Labs

| Action | Command Example |
|---|---|
| Launch terminal window | `Applications\|Accessories\|Terminal` |
| Change working folder to "~/www" | `cd ~/www` |
| Go back to home folder | `cd` |
| Go back to home folder | `cd ~` |
| Go up one level | `cd ..` |
| Find current path | `pwd` |
| List files | `ls` |
| List all files with properties | `ls -alg` |
| Copy file or folder *old* to *new* | `cp old new` |
| Move or rename file/folder *old* to *new* | `mv old new` |
| Browse contents of file *test.txt* | `more test.txt` |
| Edit file *test.txt* | `gedit test.txt` |
| Delete file *test.txt* | `rm test.txt` |
| Create folder *test* | `mkdir test` |
| Delete folder *test* (no question) | `rm -rf test` |
| Change password of *user* | `sudo passwd user` |
| Edit file /etc/sudoers | `visudo` |
| Run script file *shell.sh* in current folder | `./shell.sh` |
| Display value of environment variable PATH | `echo $PATH` |
| Process shell file ".bashrc" | `source ~/.bashrc` |
| Extract contents from *file.tar.gz* in current folder | `tar xvzf file.tar.gz –C .` |
| Install module *apache2* | `sudo apt-get install apache2` |
| Update apt-get source info | `Sudo apt-get update` |
| Create symbolic link "~/www" for folder "/var/www" | `ln -s /var/www ~/www` |
| Restart Apache web server | `sudo apache2ctl restart` |
| Change owner of all files in /var/www to *user* | `sudo chown -R user /var/www` |
| Make *file* executable by everyone | `chmod a+x file` |
| Make *file* not executable by anyone | `chmod a-x file` |
| Make *file* writable by everyone | `chmod a+w file` |
| Make *file* not writable by anyone | `chmod a-w file` |
| Make all files under folder *bin* writable by anyone | `chmod -R a+w bin` |
| Log in and work as root | `su -` |
| Run a command as root | `sudo command` |
| Download a web file at *http://url* to the current folder | `wget http://url` |
| Find the computer's IP address | `ifconfig` |
| Clear terminal screen | `clear` |
| Generate WAR file for a Java web application *test* | `jar cvf test.war * [in project folder test]` |
| Review contents of text file *test* | `more test` |

# 4 Installing Applications on a Basic *Ubuntu* Virtual Machine

## 4.1 Overview

This section teaches you how to set up most popular Linux tools and server applications on a basic version of *Ubuntu* V10.10 *VMware* virtual machine (VM), *Ubuntu*, which you learned to set up in a previous section.

Even though our instructions are on the installation of applications on a *VMware* virtual machine, they apply to the case when you install the applications on an *Ubuntu*/*Linux* installation on a physical PC. Therefore when we refer to a VM, we also implicitly refer to a physical PC if you are actually doing application installation on a physical Linux PC.

Since Linux, utility and application versions change all the time, you need to adjust the URLs, installer file names, installation folder names/paths accordingly when necessary. All the instructions are valid on May 14, 2011. The instructions will be revised when necessary.

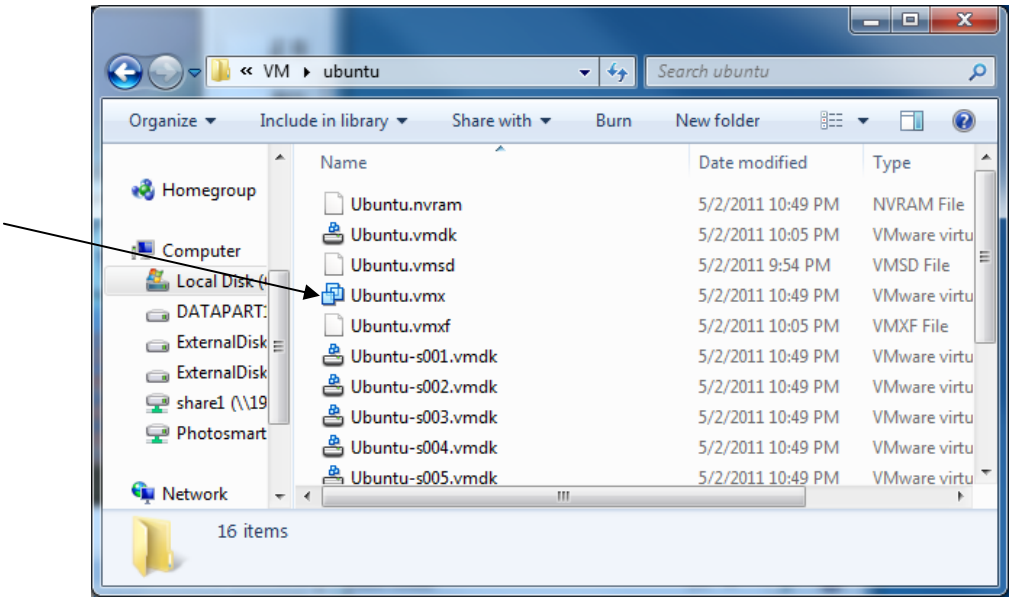
In *Ubuntu* there are three popular utilities for installing applications: *apt-get*, *aptitude*, and *Synaptic*. Utilities *apt-get* and *aptitude* are both terminal commands and very concise and flexible, while *Synaptic* has a graphic user interface (GUI) but is less flexible. In this guide we mainly use *apt-get* to install applications. In Linux, if you need to know how to use a command *cmd*, you can try to run "man *cmd*" to read *cmd*'s manual pages, or "*cmd* --help" for a short usage explanation.

While this guide uses *Ubuntu V10.10* as the base Linux system, most of the instructions apply to other flavors of Linux too, including *Red Hat*, *Fedora* (a community version of *Red Hat*) and *Debian*. On *Red Hat* Linux systems, the application installers are normally in the form of RPM format, and utility *yum* is used instead of *apt-get* or *aptitude*.

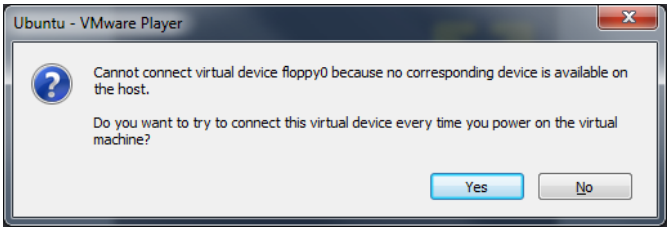
## 4.2 Launching the VM

In case you didn't complete the VM described in Section 2, you can download the VM built by following the instructions in Section 2 from http://community.seidenberg.pace.edu/files/sici2011/ubuntu1.exe. You can download file "ubuntu1.exe" into "C:\VM", and run it to generate the VM folder "C:\VM\ubuntu".

In folder "C:\VM\ubuntu", you will see the following files. Double click the VM configuration file "Ubuntu.vmx" to launch the VM. Depend on your folder property setup, you may not see the file extension "vmx". But the file has a special icon of three partially overlapped blue squares.




If you see the following message asking whether you'd like to connect the VM to a non-existent virtual device every time you power up the VM, click the "No" button.




If you are asked whether you have copied or moved your VM, answer "I copied the VM" so your VM can have a unique ID and IP address. Otherwise you cannot run multiple of the VM in a LAN.

Before we start software installation, we use menu item "Applications|Accessories|Terminal" to launch a terminal window. Test the following commands.

1. Run "pwd" to see which is the terminal windows' current working folder. A new terminal window is always opened in the current user's *home folder*. Each user has a *home folder* under "/home". Since the current user name is "user", the current working folder is "/home/user".
2. Run "ls" to see a list of files/folders in the current working folder. Files and folders are displayed in different colors.
3. Run "ls -alg" to list file/folders in the current folder with details. The left column shows the accessibility of the files/folders. The 3rd column shows the owner of the files/folders. This command also show files/folders whose names start with period (.). These files are configuration files normally hidden if you list files without using the command line switches "-alg".
4. Run "cd Downloads" to change working folder to the new folder "Downloads".
5. Run "pwd" to confirm that the current folder is "/home/user/Downloads".
6. Run "cd" to return the terminal window's working folder to the current user's home holder "/home/user".
7. Run "cd Downloads" to change working folder to the new folder "Downloads" the second time.
8. Run "cd ~" to return to "user"'s home folder "/home/user". Symbol "~" represents the home folder of the current user.

If you copy commands from this guide and paste them into VM terminal window, you need to paste the command in VM terminal window by right-clicking on the terminal window and then choosing item "Paste" on the pop-up menu. You can use the same method to paste text into text editors including *gedit* and *nano*.

## 4.3 Running Command *sudo* without Needing Password

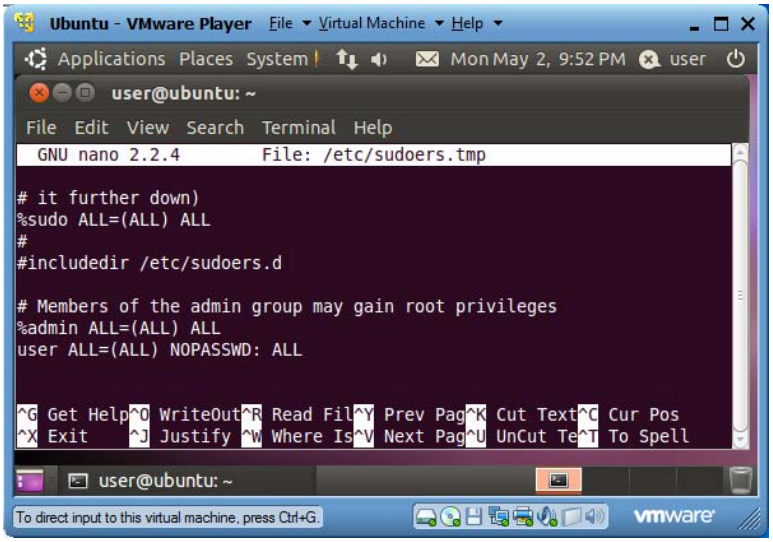
Many programs need be installed or run as super user *root*, but we don't want to share *root*'s password with other people, even if we believe them. Command "sudo" allows a user to run programs as the super user *root*. For example, "sudo ls –alg" may ask you for your Linux password (*sudo* remembers your password for 15 minutes in a terminal window), and then run "ls –alg" as *root*.

File "/etc/sudoers" defines who can assume *root*'s privilege and whether these people need to enter their Linux passwords when they use "sudo". You edit file "/etc/sudoers" by running command "visudo". To allow Linux user "john" to use "sudo" to run commands as root, add line "john ALL=(ALL) ALL" in file "sudoers". When user "john" uses "sudo", he may be prompted to enter his Linux password. If we have instead used line "john ALL=(ALL) NOPASSWD: ALL", then user "john" can use "sudo" without being prompted for his Linux password. You can find more information on "sudo" at http://www.unixtutorial.org/?s=sudoers.

Since we will use *sudo* often to run commands as *root*, we don't want to type our password each time we use *sudo*. We can avoid entering password for using *sudo* by running command "**sudo visudo**" to use editor *nano* to edit file "/etc/sudoers". Use "Page Down" key to reach the end of the file, and insert the following line:

```
user  ALL=(ALL)  NOPASSWD: ALL
```

Type key combination Ctrl+O and then key *Enter* to write the modified file (/etc/sudoers.tmp) out, and type key combination Ctrl+X to exit the editor (the basic *nano* editing commands are listed at the bottom of the editor window, where ^ means the Ctrl key). This new line in file "/etc/sudoers" specifies that user "user" can run *sudo* without providing his/her password. Now you can test by typing "sudo ls –alg" to run as *root* to list attributes of the files and folders in the current folder, and you will not be prompted to enter your password.



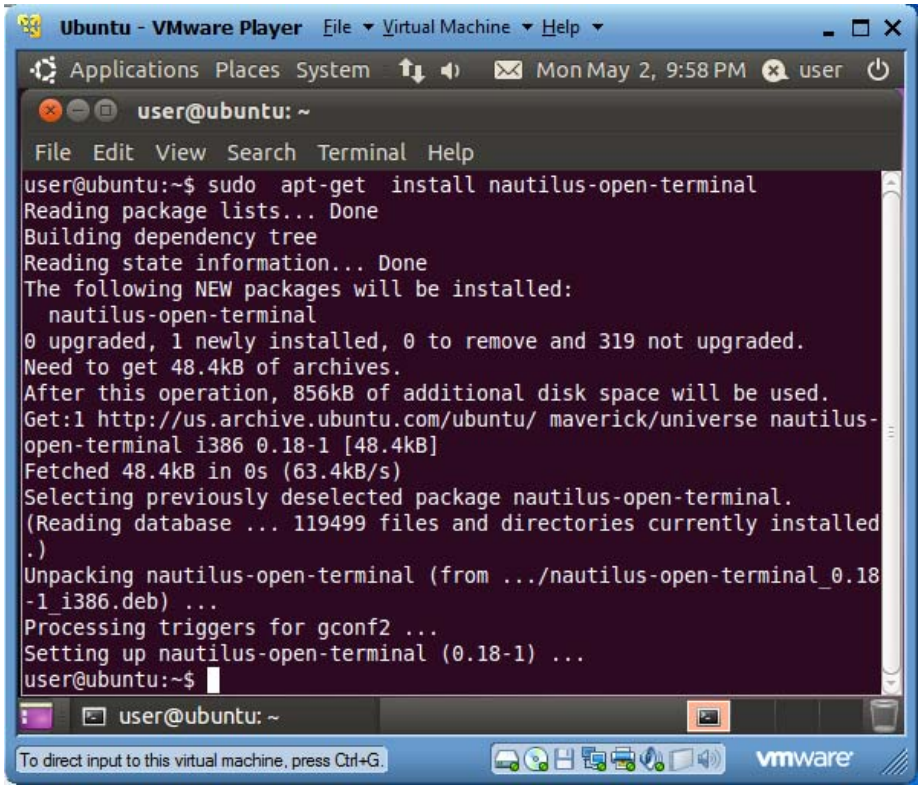
## 4.4 Adding Menu Item "Open in Terminal" to File Browser Popup Menu

Often you need to open a terminal window, change folder (directory) to a specific folder with command "cd" (*c*hange *d*irectory), and run commands there. It would be nice if we could right-click a folder in the file browser, click an "Open in Terminal" menu item in the popup menu, and have a terminal window launched in that folder. This section shows you how to add such an "Open in Terminal" menu item to the file browser popup menu.

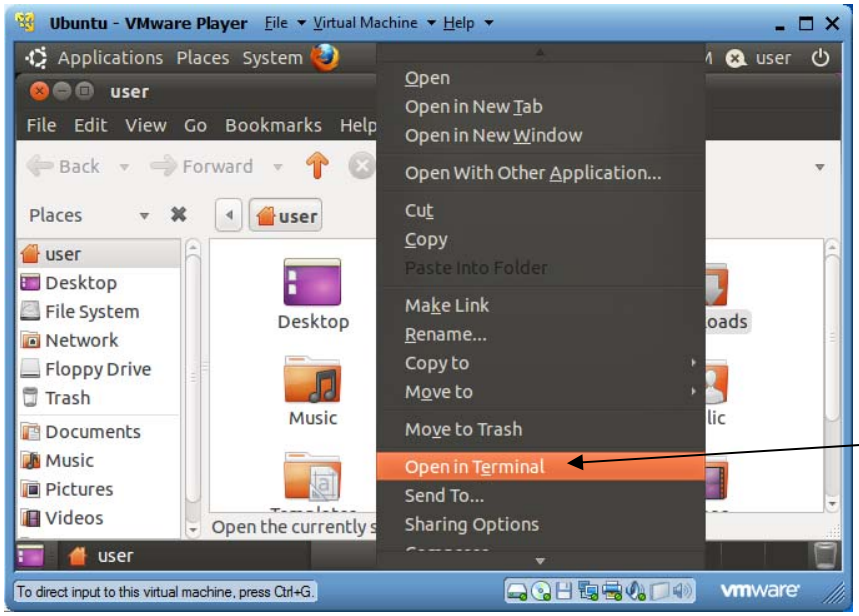
1. Start a terminal window by clicking on menu item "Applications|Accessories|Terminal".
2. To install software, you normally need be a super user, and "root" is the primary super user. You type

```
sudo  apt-get  install nautilus-open-terminal
```

to start the installation of a file explorer utility.




Reboot the VM to see the new function. In the following example, I opened the filer browser with "Places|Home Folder". The right pane shows folders and files in the home folder "/home/user" of the current user "user". If you right-click any folder or the blank space in this right pane, you will see a new menu item "Open in Terminal", as shown below. You click this menu item and a new terminal window will pop up with that folder as the current working folder.




Congratulations and you have installed your first utility in Linux!

## 4.5 Installing 7z

*7z* is a popular file zipping/unzipping utility. To install *7z*, run "sudo apt-get install p7zip-full". Run command "7z" to see how to use 7z.

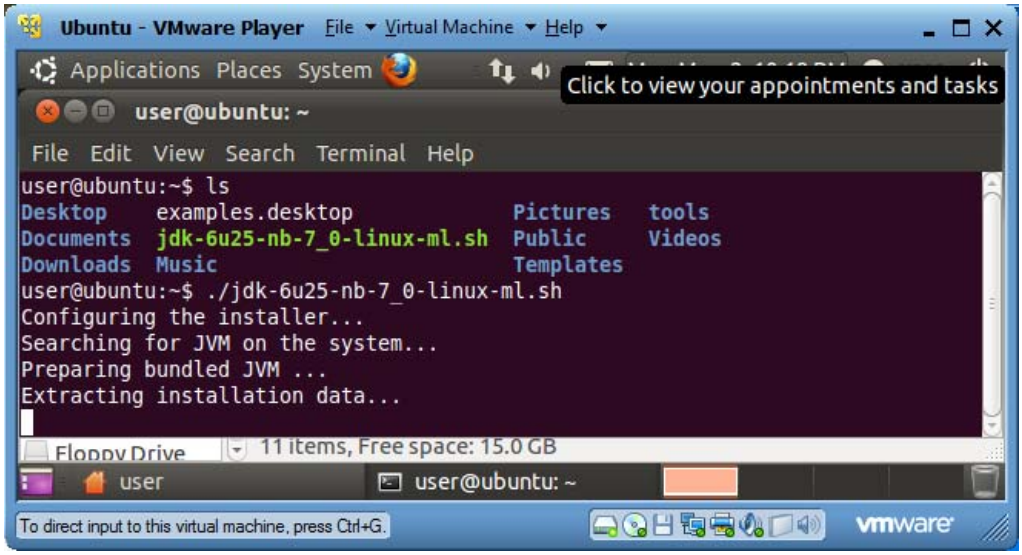
## 4.6 Installing Java JDK and NetBeans

Here I show you another software installation method on Linux. You can have more control of which version of software is installed, and the installed software files will be centralized in the installation base folder. In this subsection we install one of the latest Java JDK (Java Development Kit) packages that includes Java IDE *NetBeans* v7.0. *NetBeans* and *Eclipse* are two most popular Java IDEs.

### *4.6.1 Installing Java JDK*

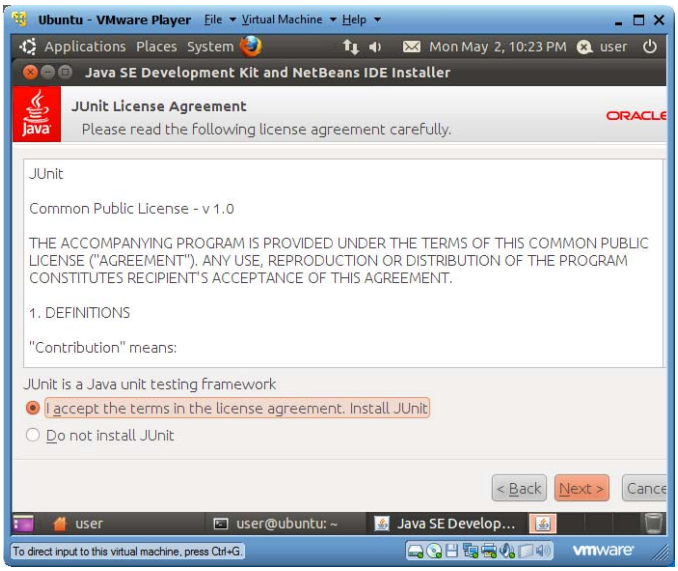
For your convenience the latest JDK installer "jdk-6u25-nb-7_0-linux-ml.sh" has been prepared for you in folder "C:\VM\resources". When you are logged in as "user", in a terminal window,

1. Drag and drop file "jdk-6u25-nb-7_0-linux-ml.sh" from your PC's folder "C:\VM\resources" to your VM's folder "/home/user".
2. Right-click any blank space in folder "user" and select "Open in Terminal" to launch a terminal window in folder "user".
3. Run "./jdk-6u25-nb-7_0-linux-ml.sh" to launch the JDK installation.

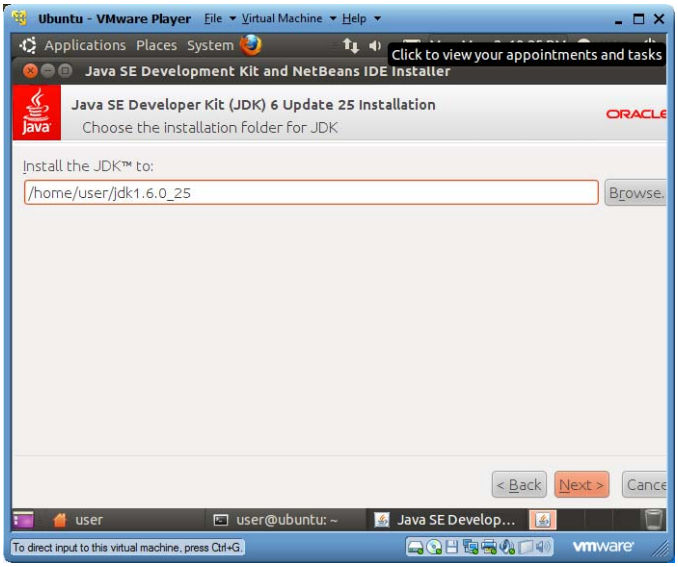




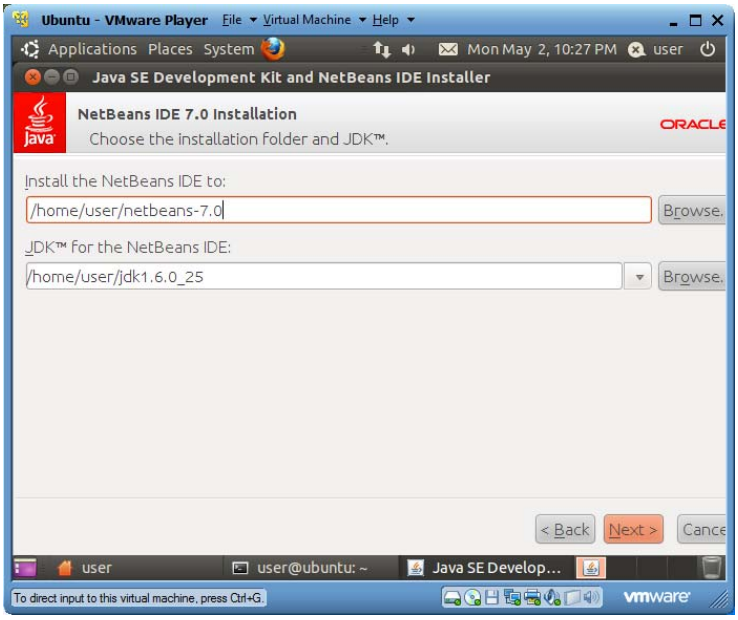
4. Click the *Next* button.



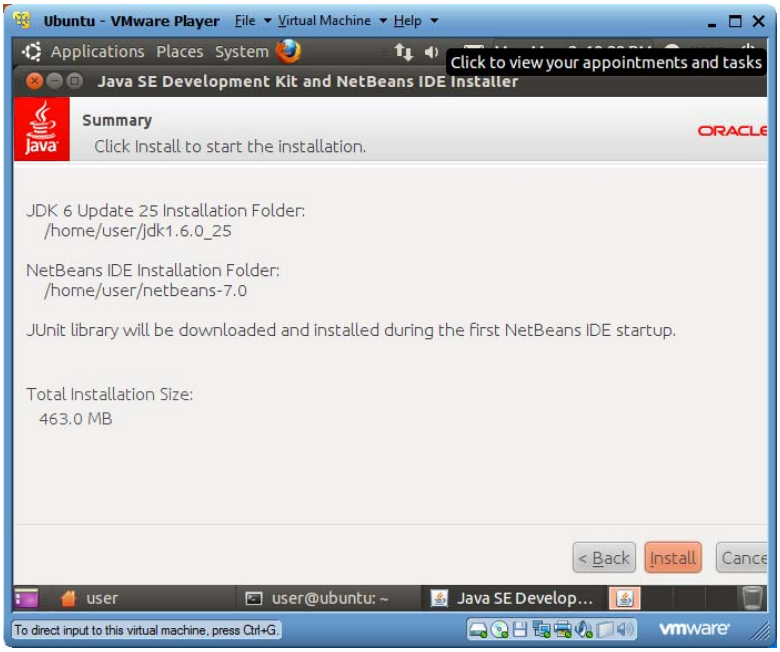

5. Check “I accept the terms …”, and then click the *Next* button.



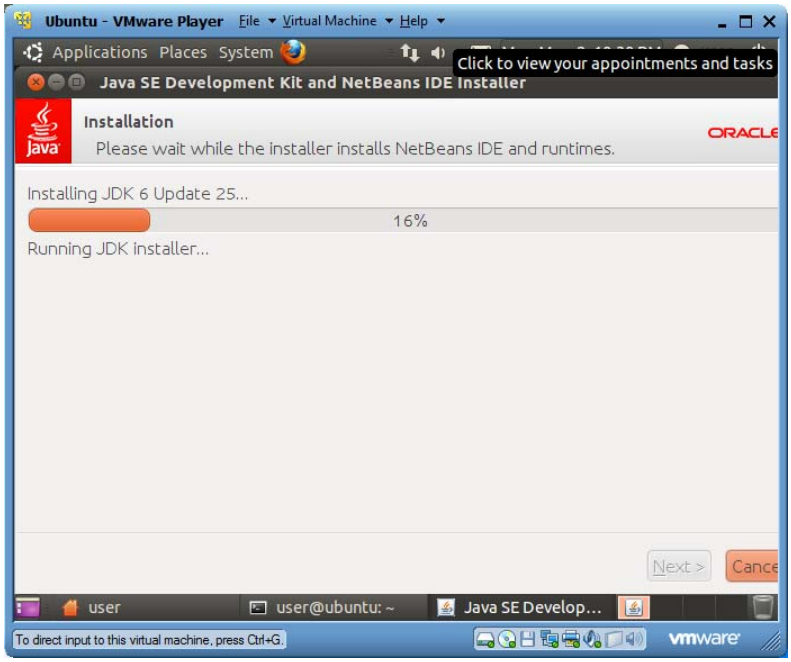

6. Click the *Next* button to accept the default installation folder.



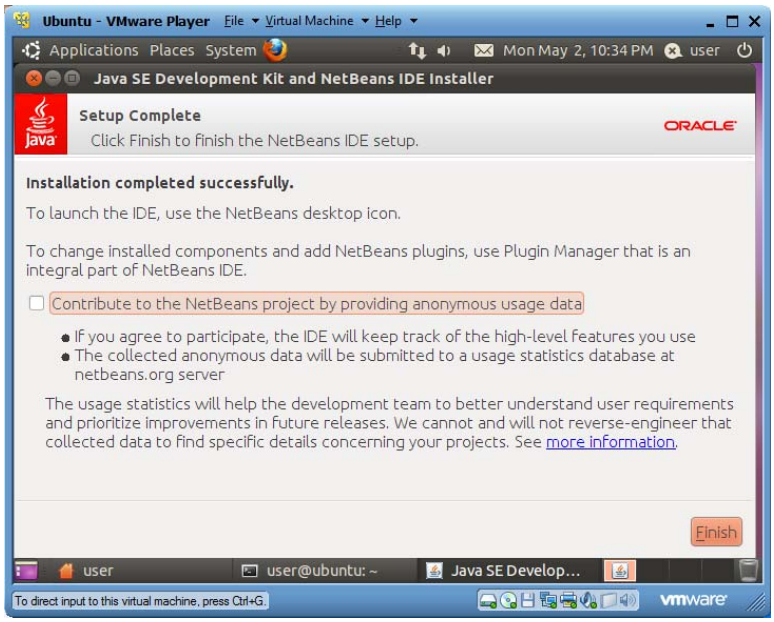

7. Click the *Next* button to accept the default folders.



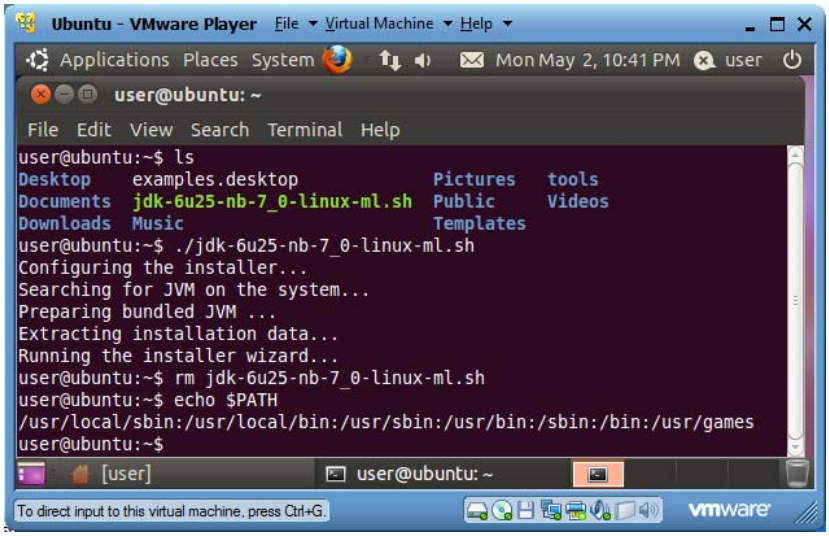

8. Click the *Install* button to start the installation.




9. When the installation come to the end, you will see the following screen:

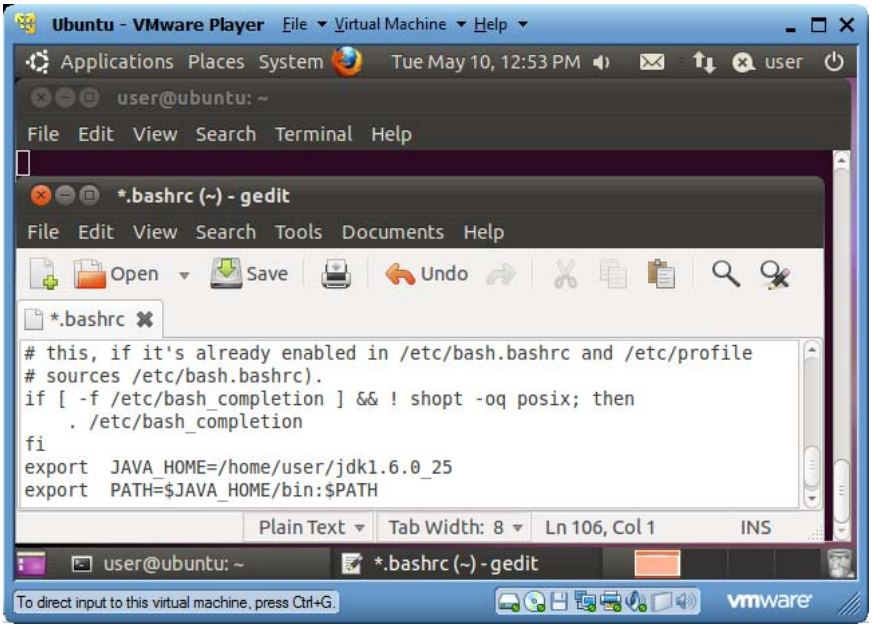



10. Click the *Finish* button.
11. You could run “rm  jdk-6u25-nb-7_0-linux-ml.sh” to delete it from folder “~”.
12. Run “echo  $PATH” to see the current definition of OS environment variable PATH. PATH is a list of folders separated by “:” in Linux and “;” in Windows. When you type a command in a terminal window, the operating system finds the corresponding execution file in the folders on the PATH, from left to right. CLASSPATH, also a sequence of folders, is another OS environment variable used by Java to find Java classes to run or compile.

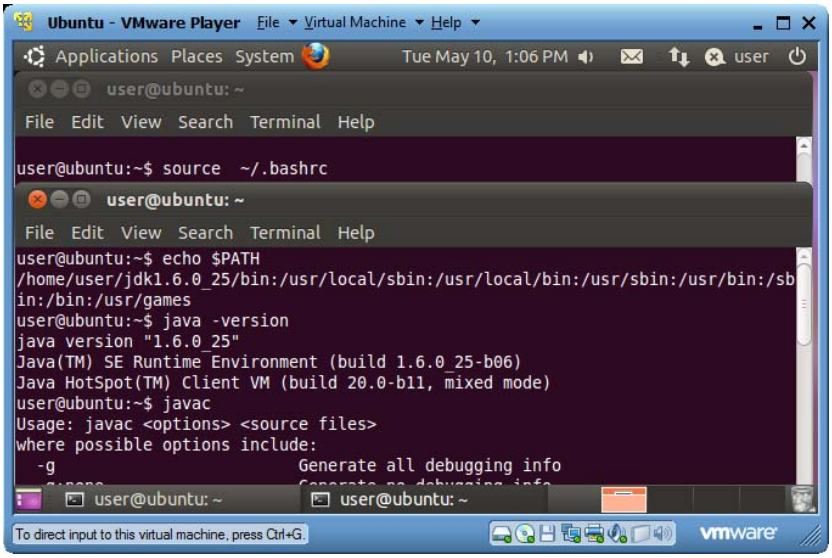



13. Run “gedit ~/.bashrc” to add the following two lines to the end of the file, which contains declarations to be processed when the current user logs in the computer. These two lines define an OS environment variable JAVA_HOME pointing to the installation folder of Java JDK, and add folder “bin” (folder for binary executable commands) of Java JDK on the PATH so the current user of this Linux PC can run those commands from any terminal window. Environment variable JAVA_HOME is needed by many Java based applications to find the Java installation on your computer. Inside Linux scripts, the value of an environment variable is represented by the $ character followed by the variable name. Therefore the second line sets the new value of variable PATH to be the JDK installation’s bin folder followed by the old value of variable PATH.

```
export  JAVA_HOME=/home/user/jdk1.6.0_25
export  PATH=$JAVA_HOME/bin:$PATH
```



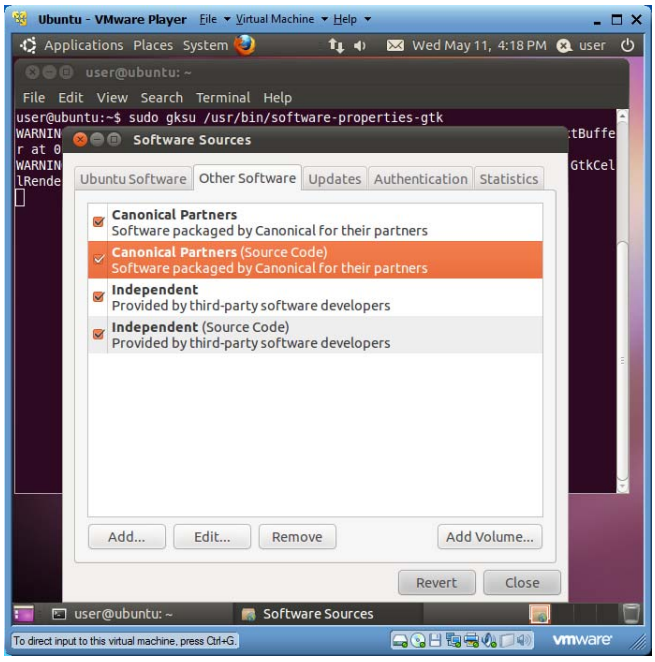
14. Save the revised file and run "source ~/.bashrc" to process the new environment variable definitions.
15. Run "echo $PATH" to see the revised definition of environment variable PATH.
16. Run command "java -version" to see the version of your Java JDK.
17. Run command "javac" to see how to use this Java compiler.



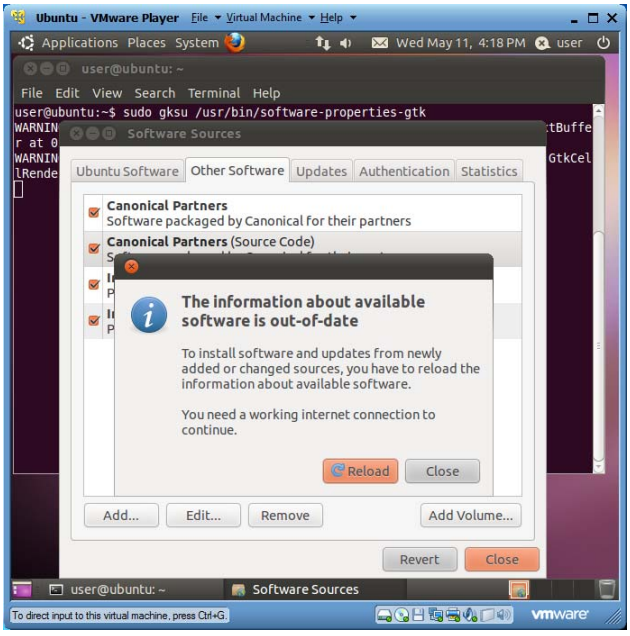

### *4.6.2 Installing JRE Plugin for Firefox*

For *Firefox* web browser to support Java applets, we need to install Java JRE Plugin.

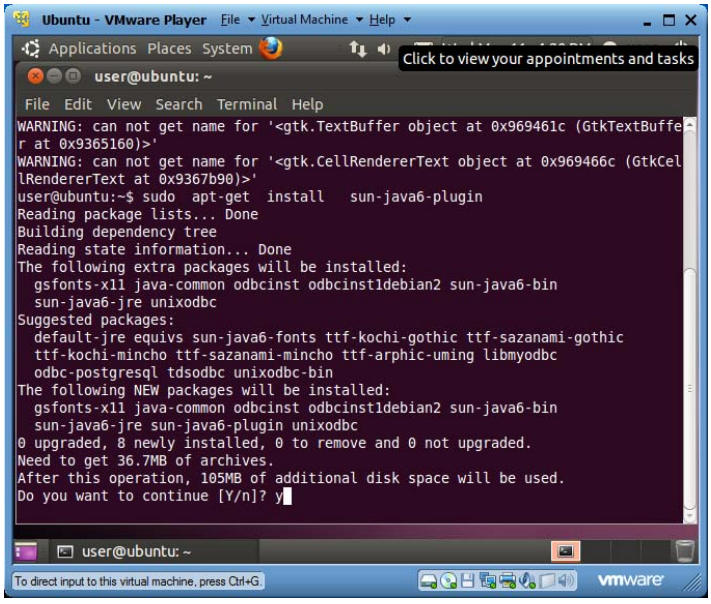
1. Run "sudo gksu /usr/bin/software-properties-gtk".



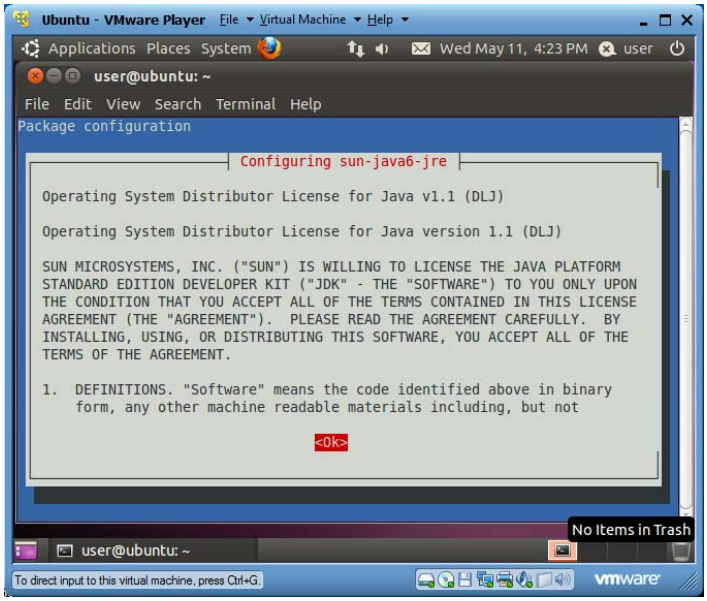
2. In the pop-up “Software Sources” window, click the “Other Software” tab and check the top two “Canonical Partners” checkboxes. Then click “Close”.



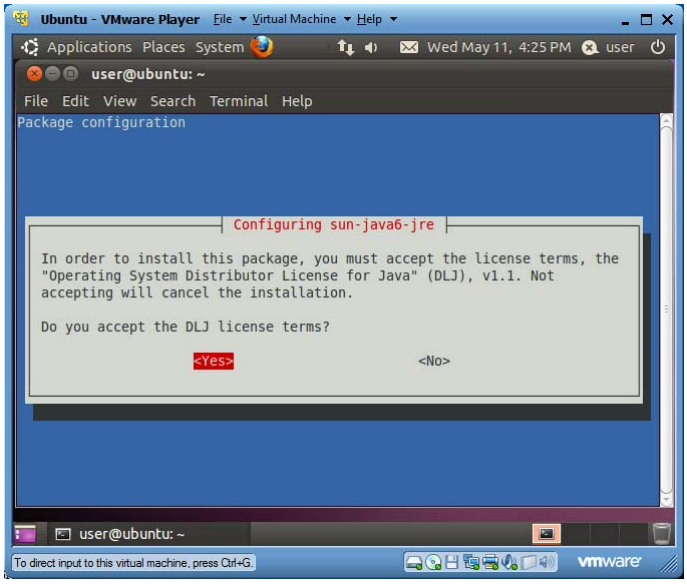

3. When prompted, click "Reload".
4. Run “sudo apt-get install sun-java6-plugin” to install JRE Plugin.



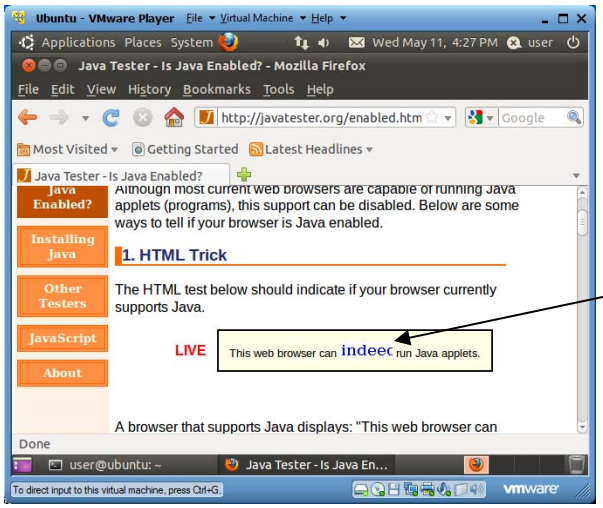

5. Upon request, respond with *y* to continue the installation.




6. Use tab key to highlight *OK* in red, then type the *ENTER* key.




7. Use tab key to highlight *Yes*, then type the *ENTER* key.
8. Use *Firefox* to display the web page at http://javatester.org/enabled.html to verify that your browser supports applet.

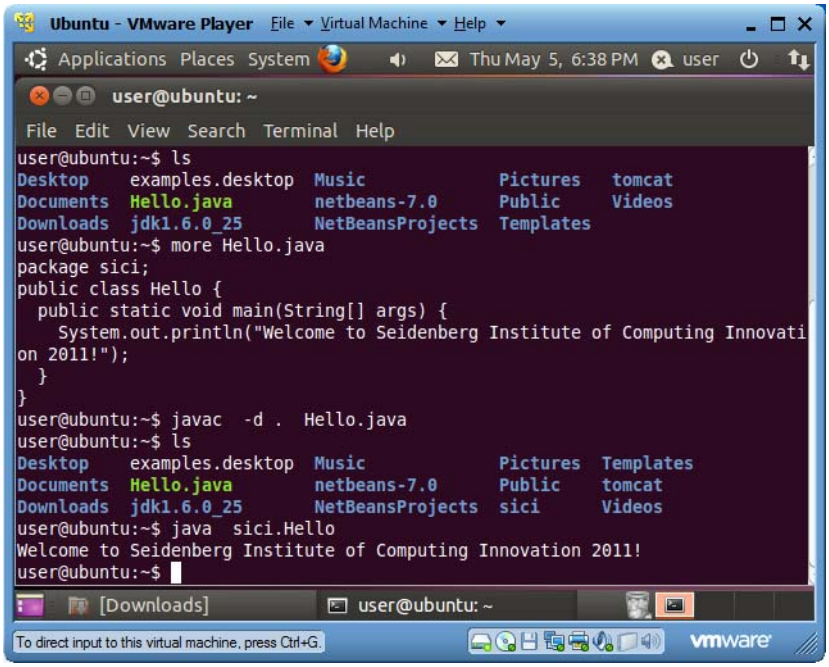



### *4.6.1 Testing Java Commands **javac** and **java***

1. Run “cd” to go back to home folder.
2. Run “gedit Hello.java” to create a new Java source file “Hello.java”. Enter the following contents and save the file. Exit the editor.

```
package  sici;
public class Hello {
  public static void main(String[] args) {
    System.out.println("Welcome to Seidenberg Institute of Computing Innovation 2011!");
  }
}
```

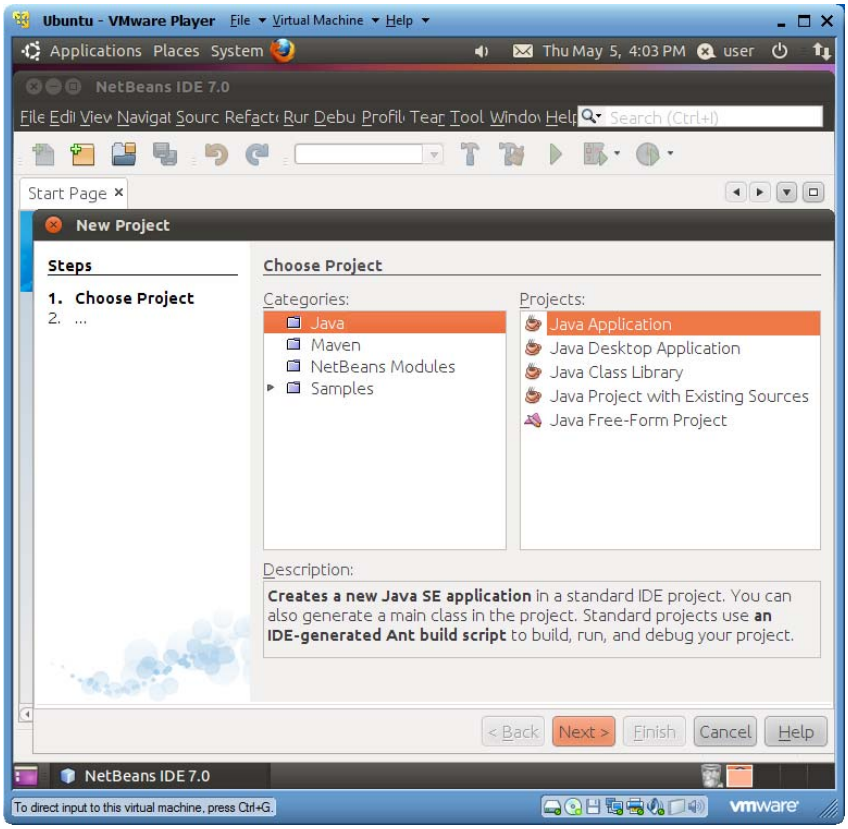
3. For your convenience, this file is also available in your PC’s folder “C:\VM\resources”, and you can drag and drop it to your home folder ~.
4. Run “javac -d . Hello.java” to compile file “Hello.java” into bytecode file “Hello.class”. Command-line switch “-d .” specifies to generate class file in the current folder. Since class *Hello* belongs to package “sici”, a new folder “sici” is automatically created to hold the new class file “Hello.class”.
5. Run “java sici.Hello” to run Java program *Hello* in Java package *sici*.



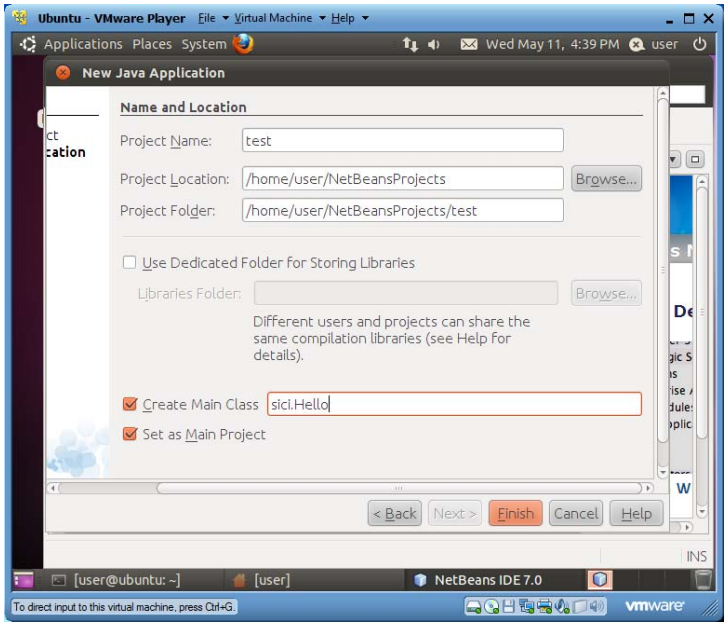

### *4.6.2 Testing NetBeans v7*

This section shows how to create the same *Hello.java* file through *NetBeans*, a powerful open-source competitor of *Eclipse* featuring a mature WYSWYG GUI designer.

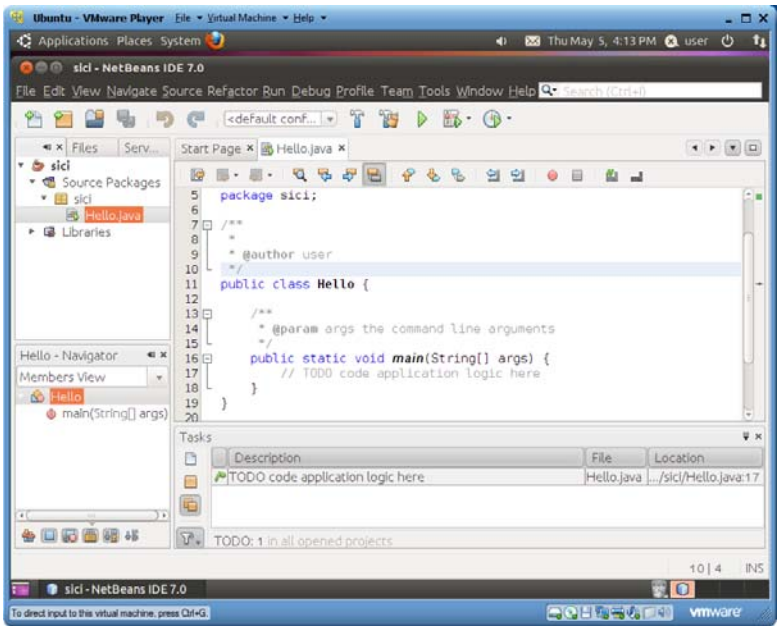
1. Use the desktop icon “NetBeans IDE 7.0” or menu item “Applications|Programming|NetBeans IDE 7.0” to launch *NetBeans*.
2. Click menu item “File|New Project …” to launch the “New Project” window. Make sure *Categories* selection chooses “Java”, and *Projects* selection chooses “Java Application”, as shown below.



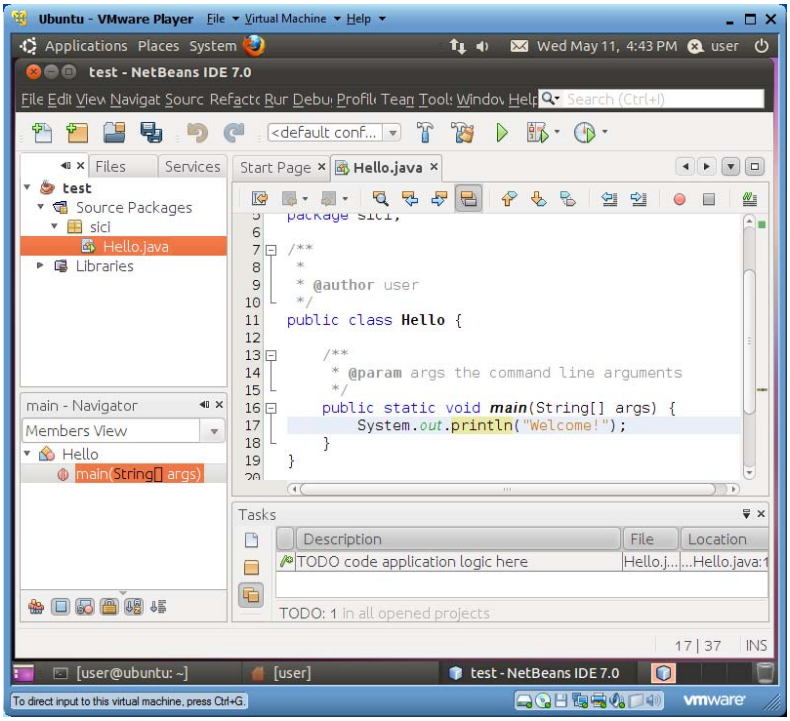

3. Click the *Next* button and see the "New Java Application" window. Type "test" in the "Project Name" text field, as shown below. Make sure that the checkboxes for "Create Main Class" and "Set as Main Project" are checked. Set the main class name to be "sici.Hello".



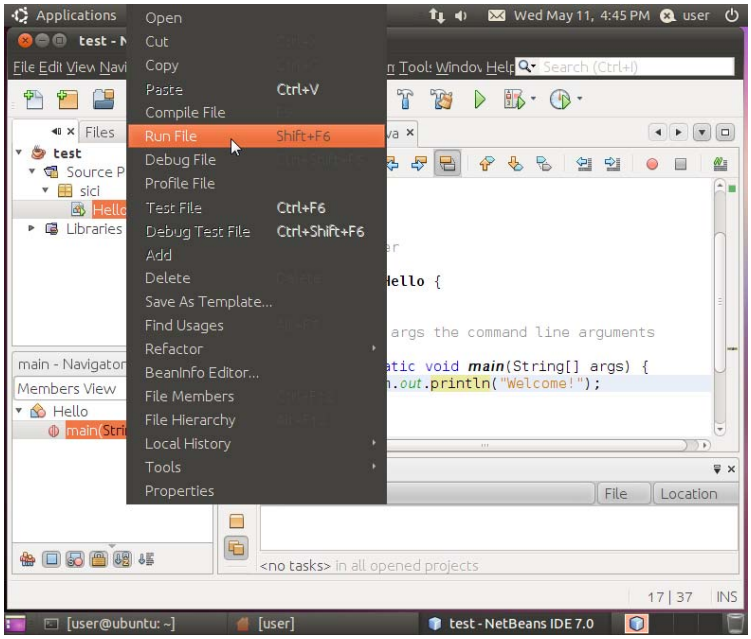

4. Click the *Finish* button, and you will see a screen similar to the following one:



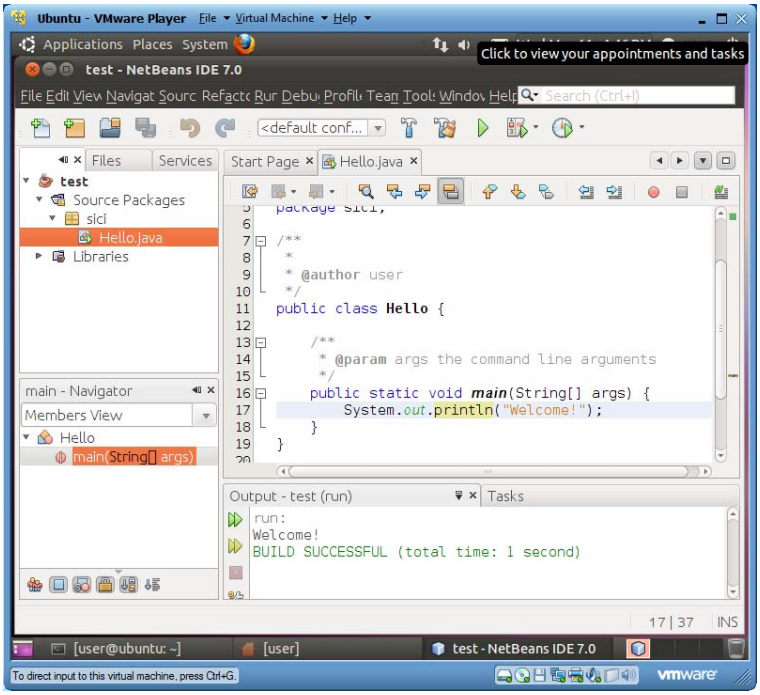

5. In the body of method *main*, enter “System.out.println("Welcome!");”, as shown below.




6. Click menu item “File|Save” to save the file. In the left-upper project pane, right-click file “Hello.java” and choose “Run File” on the popup menu, as show below:




7. The program will be compiled into a bytecode file and executed. The execution output is displayed under the "Output" tab, as shown below:




8. Click menu item "File|Close Project" to close this project, and use "File|Exit" to shut down *NetBeans* IDE.
9. Your *NetBeans* projects are saved under "/home/user/NetBeansProjects".

## 4.7 Installing Apache Tomcat

*Apache Tomcat*, later simply called *Tomcat*, is an open-source web server featuring a *Java servlet container* for using Java technologies, including servlets, *JavaServer Pages (JSP)* and *JavaServer Faces (JSF)*, to create HTML files upon client requests over the Internet. It lacks some general web server features like security settings and virtual hosts. On the other hand, *Apache* web server has richer basic web server functions for serving pre-authored (static) HTML files as well as using the older CGI technologies like Perl and PHP to generate HTML files on-the-fly. For a typical enterprise web server, *Apache* web server usually works at front-end serving clients at its default port 80, and if the client request needs be processed by Java technologies, *Apache* would forward (delegate) the request to a *Tomcat* web server behind it. By default *Tomcat* works at port 8080.

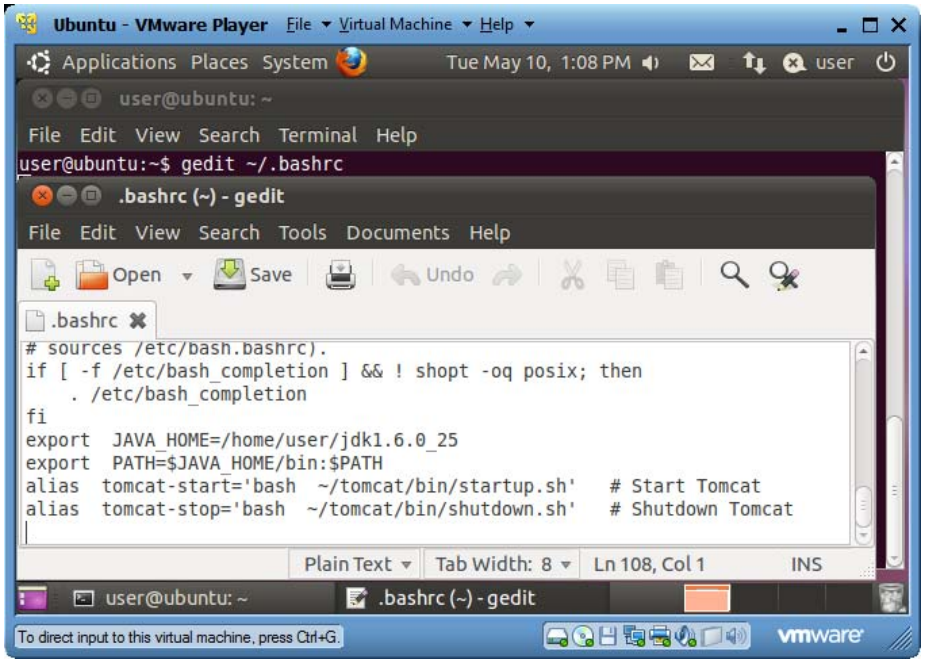
Before you can install *Tomcat*, you must have installed Java JDK or JRE.

### *4.7.1 Installing Tomcat*

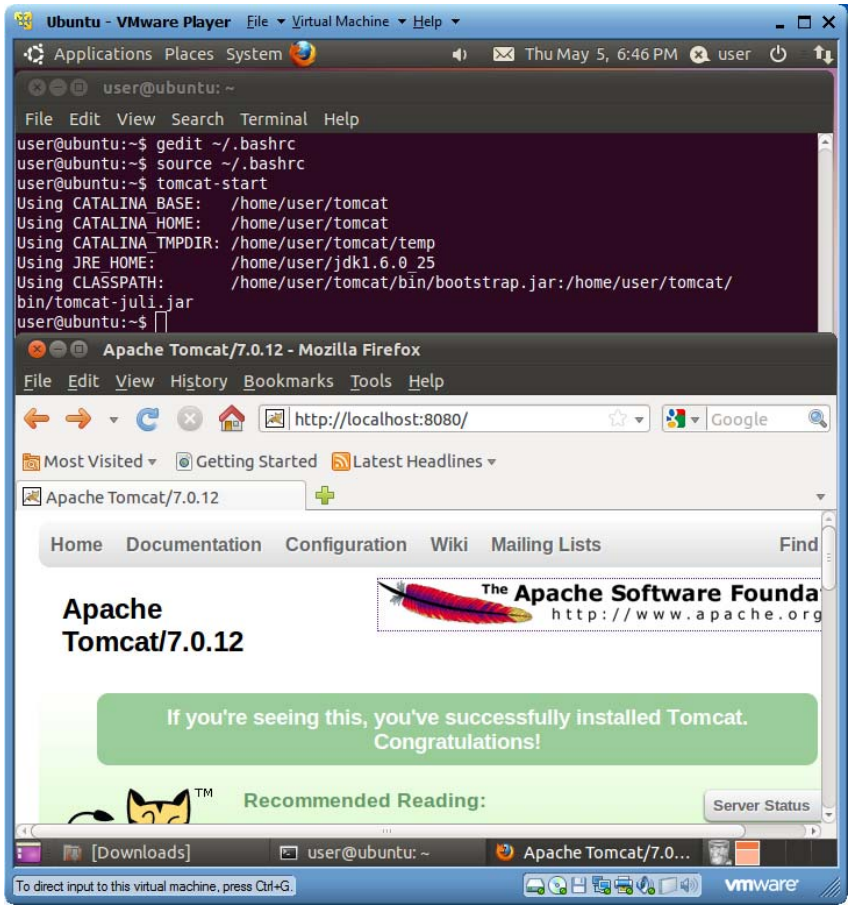
The easiest way to install Tomcat is to use "apt-get". If you use this approach, you have to wait for people to prepare the installation package for you. You cannot install the latest version of software. The installed files would be distributed and harder to follow for people new to the Linux environment. Here you learn an installation approach that can be used in any operating systems for installing the latest versions of software.

1. Start your VM's file explorer in *user*'s home folder.
2. Drag and drop *Tomcat* installer "apache-tomcat-7.0.12.tar.gz" from your PC's "C:\VM\resources" to your VM's home folder.
3. In a terminal window in *user*'s home folder, run "tar xvzf apache-tomcat-7.0.12.tar.gz -C ." to extract the *Tomcat* folder in "/home/user/ apache-tomcat-7.0.12".
4. You could now run "rm apache-tomcat-7.0.12.tar.gz" to delete the *Tomcat* installer.
5. For convenience, run "mv apache-tomcat-7.0.12 tomcat" to rename folder "apache-tomcat-7.0.12" to "tomcat".
6. Run "gedit ~/.bashrc" to use *gedit* to insert the following two lines of alias definitions at the end of file "~/.bashrc" (strings to the right of # are comments):

```
alias  tomcat-start='bash  ~/tomcat/bin/startup.sh'   # Start Tomcat
alias  tomcat-stop='bash  ~/tomcat/bin/shutdown.sh'   # Shutdown Tomcat
```




7. Run “source ~/.bashrc” or reboot the VM.
8. From now on, you can run “tomcat-start” to start the *Tomcat* web server, and run “tomcat-stop” to stop the *Tomcat* web server.
9. Run “tomcat-start” to start the *Tomcat* web server. Launch *Firefox* to visit http://localhost:8080. You will see the start page of *Tomcat* as shown below:




10. If you need to shut down *Tomcat*, run “tomcat-stop”.

### *4.7.2 Tomcat File Organization*

Our *Tomcat* installation base folder is “/home/user/tomcat”. The following discussion is relative to this folder.

1. All web applications deployed on *Tomcat* are in their own folders under folder “webapps”.
2. The script files for start/stop *Tomcat* are in folder “bin”.
3. *Tomcat* configuration files are in folder “conf”. File “conf/server.xml” is the main configuration file with which you can change the default port 8080.
4. *Tomcat* logging files are in folder “logs”. You can read file “logs/catalina.out” to see *Tomcat* startup/stop and error messages. If you print any (debugging) message in your web applications’ Java code, the message will be printed to this file. Therefore you can use Java print statements and this file to do simple debugging of your web applications.
5. Your web applications’ JSP files will be converted to Java servlet source code under folder “work/Catalina/localhost” and organized as web application folders.

## 4.8 Installing *MySQL* Database Server

*MySQL* community version is a popular open-source database server from *Oracle*. This guide shows how to install it on our VM.

1. Run command "sudo apt-get install mysql-server" to download and install the current MySQL server prepared for Linux. At the time of writing it installs MySQL 5.1.
2. When asked for password for root (MySQL account root; it is different from Linux's super user root), enter 123456 (make sure you use this password because some of our example web applications use it; you can change the password later after you are proficient in Linux).
3. MySQL will be installed as a Linux service, and it will start automatically at system boot up time.
4. Many of our example web applications will use a database named "test". For security reason our installer doesn't create this "test" database for us. We now create it with MySQL's administrator console.
    a) Run command "mysql -u root -p123456" to login *MySQL* admin console with *root* as user name and 123456 as password.
    b) After the *MySQL* command prompt "mysql>",
        i. run command "create database test;" to create database "test"
        ii. run command "show databases;" to view a list of available databases
        iii. run command "use test;" to use database "test" as the default database for the following commands
        iv. run command "show tables;" to list the tables in the current database
        v. run command "quit;" to quit the *MySQL* admin console.

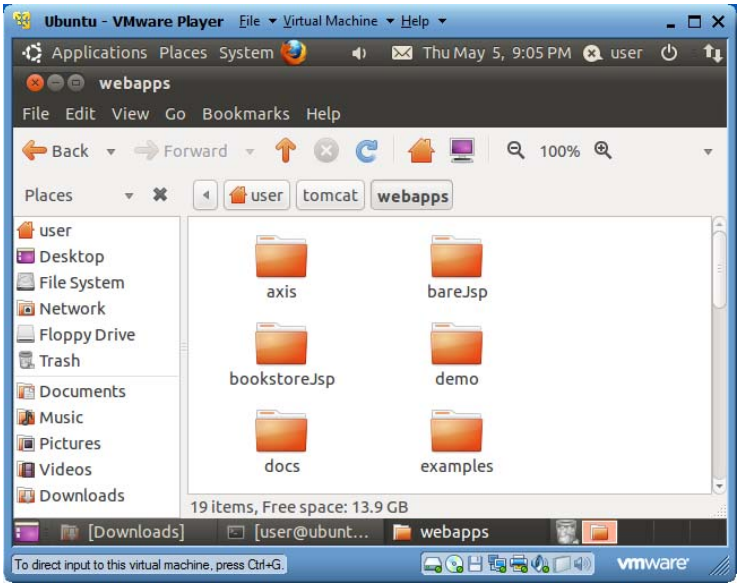
    The following screen capture shows these steps' execution.

```
  user@ubuntu:~$ sudo  apt-get  install  mysql-server
  Reading package lists... Done
  Building dependency tree
  Reading state information... Done
  Reading extended state information
  Initializing package states... Done
  The following NEW packages will be installed:
    libdbd-mysql-perl{a} libdbi-perl{a} libhtml-template-perl{a}
    libnet-daemon-perl{a} libplrpc-perl{a} mysql-client-5.1{a}
    mysql-server mysql-server-5.1{a} mysql-server-core-5.1{a}
  0 packages upgraded, 9 newly installed, 0 to remove and 0 not
upgraded.
  Need to get 20.4MB of archives. After unpacking 48.8MB will be used.
  Do you want to continue? [Y/n/?] y
  Writing extended state information... Done
  ……
  user@ubuntu:~$ mysql  -u root  -p123456
  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 40
  Server version: 5.1.37-1ubuntu5 (Ubuntu)

  Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
  Mysql> create  database  test;
  Query OK, 1 row affected (0.03 sec)
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
3 rows in set (0.00 sec)

mysql> use test;
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> quit;
Bye
user@ubuntu:~$
```

## 4.9 Deploying Sample Tomcat Web Applications

1. Drag and drop WAR (Web Archive) files in folder "C:\VM\resources\Sample Tomcat web applications" of your PC to VM folder "~/tomcat/webapps".
2. Run command "start-tomcat" to start *Tomcat* server.
3. In a file explorer, observe the automatic extraction and deployment of web applications from the WAR files.

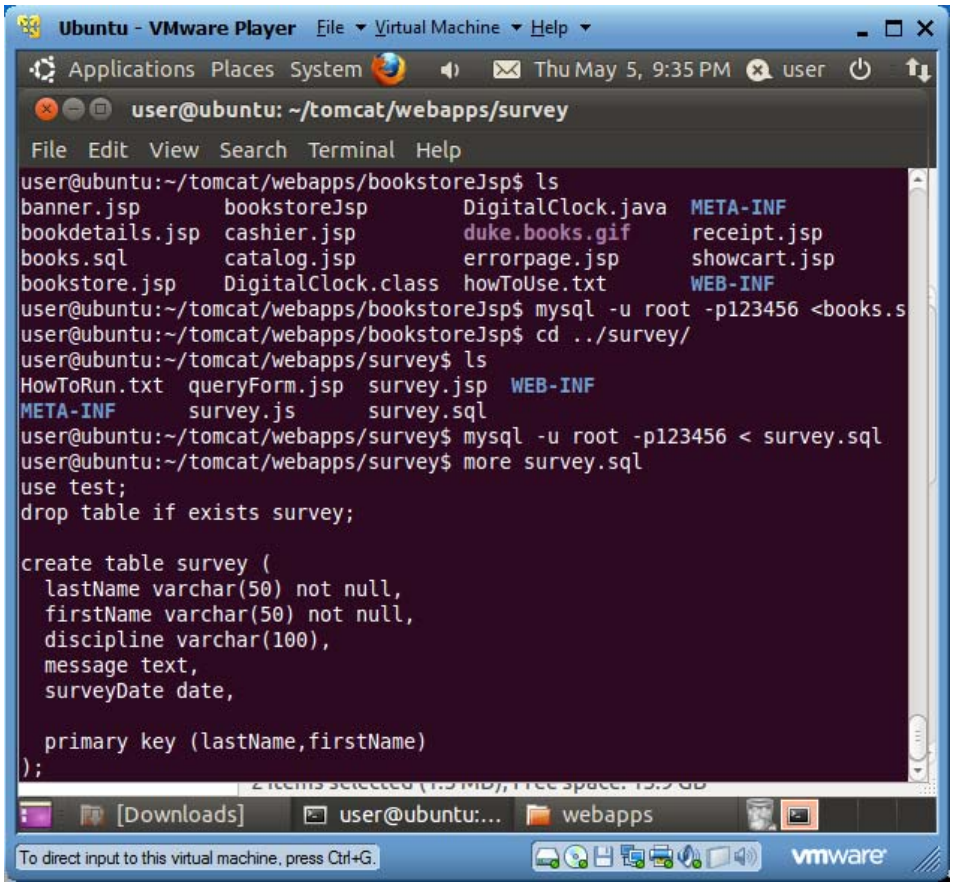



4. In a terminal window, run "cd ~/tomcat/webapps" to change working folder to where the WAR files are deployed.
5. Run "cd bookstoreJsp" to go inside web application folder "bookstoreJsp".
6. Run "ls" to list the files. You will notice a file "books.sql".

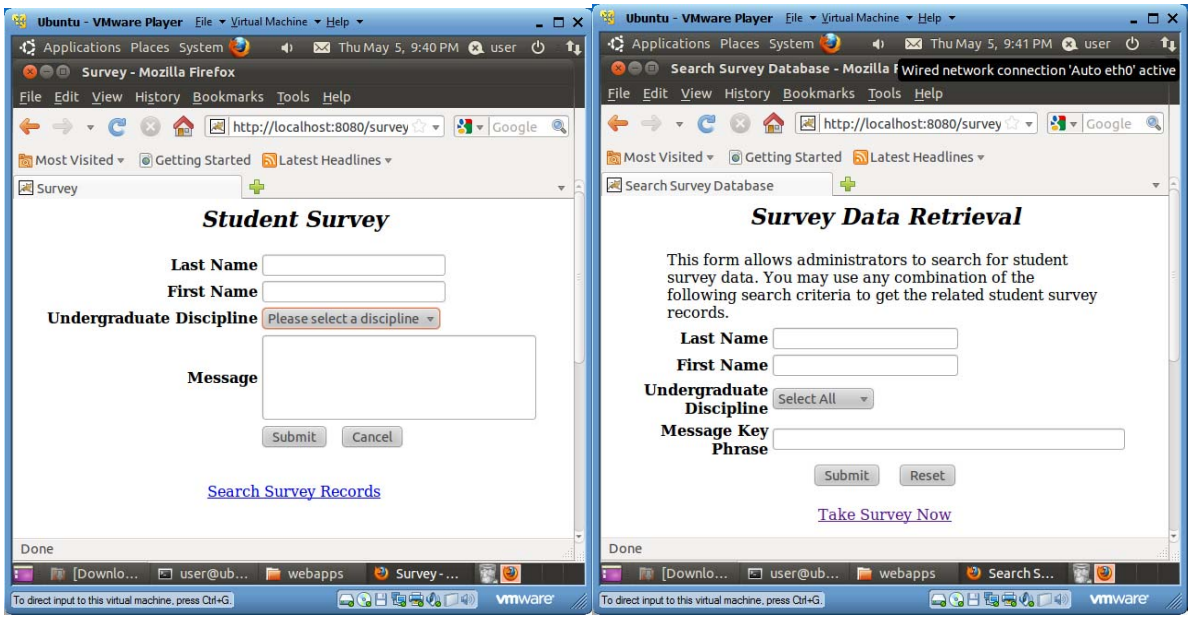
7. Run “more books.sql” to review the top section of SQL code for creating a table “books”:

```
USE test;
DROP TABLE IF EXISTS books;
CREATE TABLE books
(id VARCHAR(8),
surname VARCHAR(24),
first_name VARCHAR(24),
title VARCHAR(96),
price FLOAT,
yr INT,
description VARCHAR(100),
inventory INT,
primary key (id)
);
```

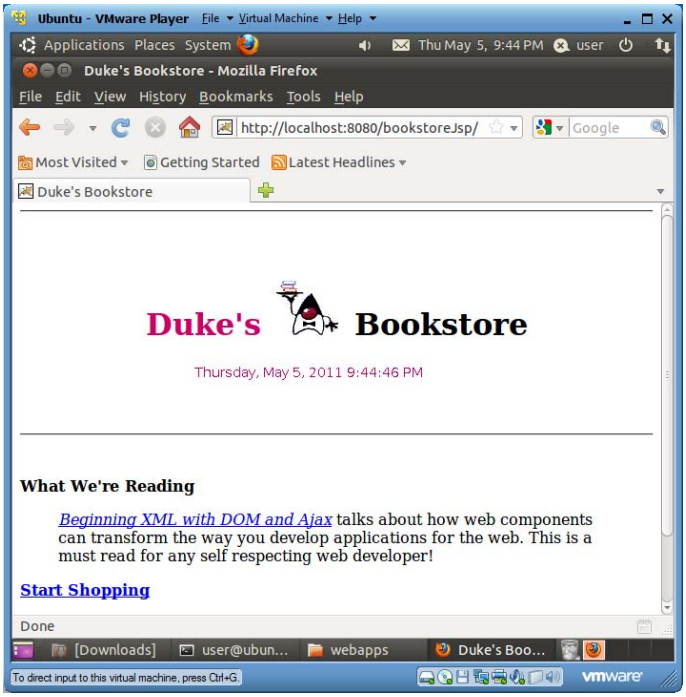
8. Run “mysql -u root -p123456 < books.sql” to execute SQL statements in file “books.sql”. A new table “books” will be created in database “test” and populated with data.
9. Run “cd ../survey” to move working folder to web application folder “survey”.
10. Run “mysql -u root -p123456 < survey.sql” to execute SQL statements in file “survey.sql”. A new table “survey” will be created in database “test”.
11. Run “more survey.sql” to review contents of file “survey.sql”.



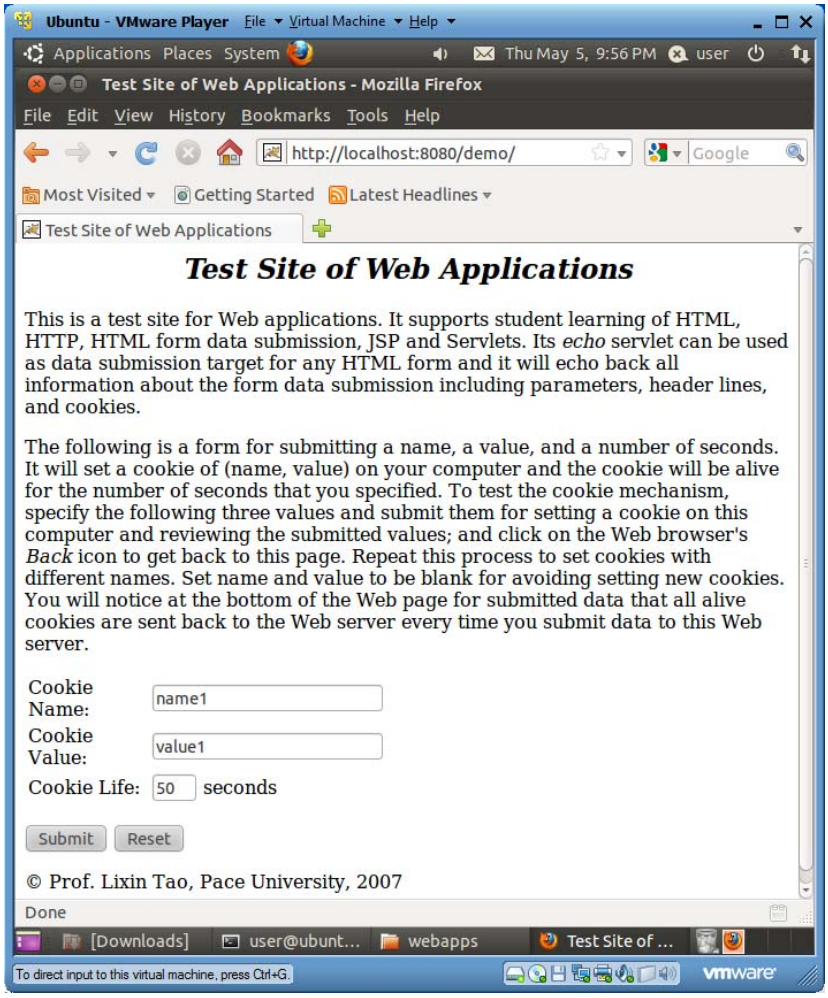
12. Use a web browser to visit http://localhost:8080/survey, enter an entry, and search and review the new entry.




13. Use a web browser to visit http://localhost:8080/bookstoreJsp, and try to buy some books.




14. Use a web browser to visit http://localhost:8080/demo, and try to create some cookies with various life spans.




15. Use your physical PC's web browser to visit http://community.seidenberg.pace.edu:8080/demo/. When you learn HTML form and HTTP protocol, you can send your form data to http://community.seidenberg.pace.edu:8080/demo/echo to echo back all data received by the web server.

## 4.10 Installing *Apache* Web Server and PHP

### *4.10.1 Installing Apache 2*

*Apache* web server is the most popular open-source web server. This section explains how to install it.

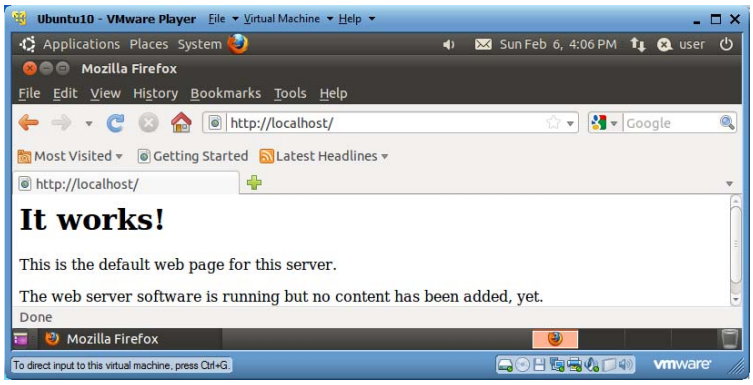
1. Run "sudo apt-get install apache2" to download and install *Apache*. If you see 404 error messages, make sure you update your VM's apt-get information by running "sudo apt-get update" and then try this step again.
2. Run "sudo gedit /etc/apache2/httpd.conf" and insert the following line in it:

```
ServerName    localhost
```

3. Run command "ln -s /var/www  www" to create a symbolic link or shortcut "/home/user/www" to "/var/www", the document root folder for *Apache*. Each web site of yours on *Apache* will be in a folder under "/var/www". The *Apache* web server's installation folder is "/etc/apache2".

```
   user@ubuntu:~$ sudo  apt-get  install  apache2
   Reading package lists... Done
   Building dependency tree
   Reading state information... Done
   Reading extended state information
   Initializing package states... Done
   The following NEW packages will be installed:
     apache2 apache2-mpm-worker{a} apache2-utils{a} apache2.2-bin{a}
     apache2.2-common{a}    libapr1{a}    libaprutil1{a}    libaprutil1-dbd-
sqlite3{a}
     libaprutil1-ldap{a}
   0 packages upgraded, 9 newly installed, 0 to remove and 0 not upgraded.
   Need to get 2,088kB of archives. After unpacking 6,939kB will be used.
   Do you want to continue? [Y/n/?] y
   ……
   user@ubuntu:~$ sudo  gedit   /etc/apache2/httpd.conf
   user@ubuntu:~$ ln -s /var/www  www
   user@ubuntu:~$ ls
   Desktop    examples.desktop  Music             Pictures  Templates  www
   Documents  Hello.java        netbeans-7.0      Public    tomcat
   Downloads  jdk1.6.0_25       NetBeansProjects  sici      Videos
   user@ubuntu:~$ cd www
   user@ubuntu:~/www$ ls
   index.html
   user@ubuntu:~/www$ more index.html
   <html><body><h1>It works!</h1>
   <p>This is the default web page for this server.</p>
   <p>The web server software is running but no content has been added,
yet.</p>
   </body></html>
   user@ubuntu:~/www$ cd /etc/apache2
   user@ubuntu:/etc/apache2$ ls
   apache2.conf  httpd.conf   mods-available  sites-available
   conf.d        httpd.conf~  mods-enabled    sites-enabled
   envvars       magic        ports.conf
   user@ubuntu:/etc/apache2$
```

4. Launch *Firefox* web browser and visit http://localhost. You will see a screen similar to the following one. *Firefox* is rendering HTML file "/var/www/index.html". *Apache* has been installed as a Linux service and it will automatically start at Linux boot-up time.




16. At any time, you can restart Apache web server by running "sudo apache2ctl restart".

### *4.10.2 Installing PHP*

*PHP* is a CGI (Common Gateway Interface) technique for dynamically generating HTML files based on clients' HTTP requests. It is more powerful than *Perl*. To install *PHP* v5 on *Apache*, run the following commands:

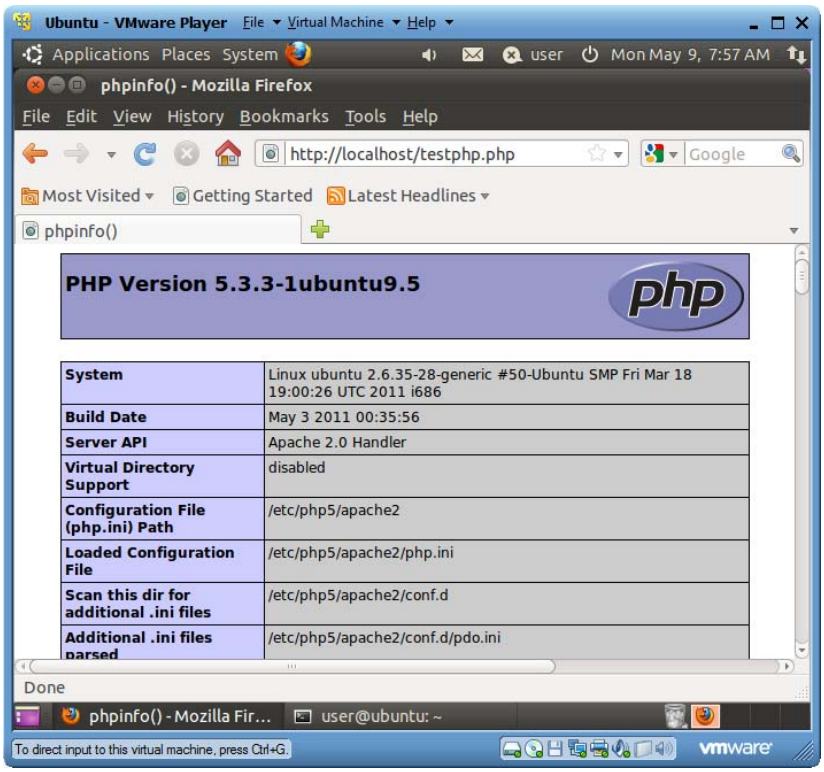
1. Run "sudo apt-get install php5 libapache2-mod-php5" to install *PHP5* on your Linux system.

```
user@ubuntu:~$ sudo  aptitude  install  php5  libapache2-mod-php5
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
The following NEW packages will be installed:
  apache2-mpm-prefork{a} libapache2-mod-php5 php5 php5-common{a}
The following packages will be REMOVED:
  apache2-mpm-worker{a}
0 packages upgraded, 4 newly installed, 1 to remove and 0 not upgraded.
Need to get 2,929kB of archives. After unpacking 6,304kB will be used.
Do you want to continue? [Y/n/?] y
……
user@ubuntu:~$ sudo  apache2ctl  restart
```

2. Run command "sudo apache2ctl restart" to restart *Apache* web server.
3. Run "sudo chown -R user /var/www" to give "user" the right to work in Apache folders as their owners.
4. To test your PHP5 installation, run command "gedit ~/www/testphp.php", and insert the following line into the file:

```
<?php  phpinfo(); ?>
```

5. Save the file, and use *Firefox* to visit http://localhost/testphp.php. If you see a screen similar to the following one, you have succeeded in installing *PHP5* in *Apache*.




## 4.11 Installing *Eclipse*

Before you can install *Eclipse*, you must have installed Java JDK.

*Eclipse* is another IDE (Integrated Development Environment) for developing software projects in Java. Since *Eclipse* is implemented in Java itself, Java JRE or JDK must be installed before this step.

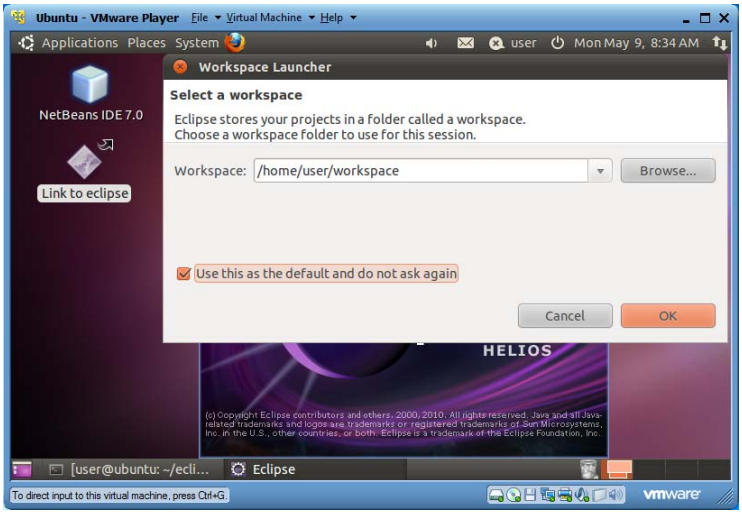
### *4.11.1 Eclipse Setup*

1. Drag and drop the *Eclipse* installer "eclipse-jee-helios-SR1-linux-gtk.tar.gz" from your PC's "C:\VM\resources" to your VM's home folder ~.
2. Launch a terminal window in home folder ~.
3. Run command "tar xvzf eclipse-jee-helios-SR1-linux-gtk.tar.gz -C ." to unzip the installer contents in the current home folder. The *Eclipse* installation folder is "/home/user/eclipse".
4. You could delete the installer by running command "rm eclipse-jee-helios-SR1-linux-gtk.tar.gz".
5. Use file explorer visit folder "~/eclipse". Right-click on file "~/eclipse/eclipse" and choose "Make Link". Drag and drop the new file "Link to eclipse" to the desktop. Now you can double-click this link (shortcut) to launch *Eclipse*.

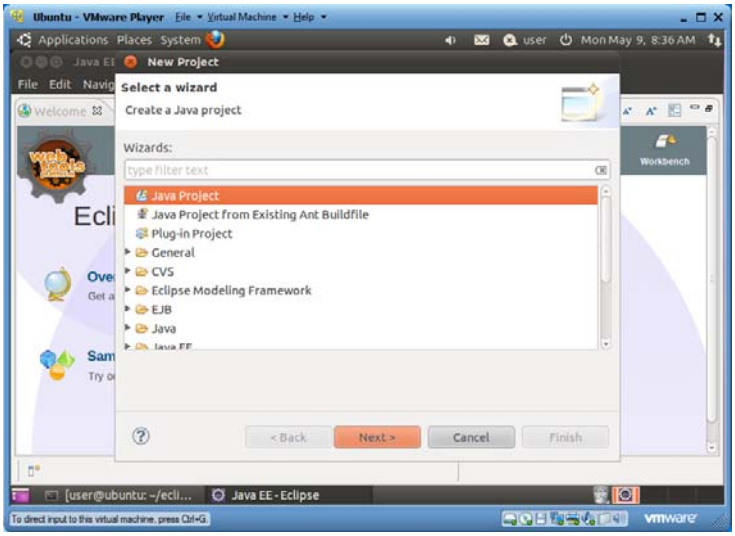
### *4.11.2 Developing a Sample Java Program*

1. Double-click the shortcut "Link to eclipse" to launch *Eclipse* for the first time.

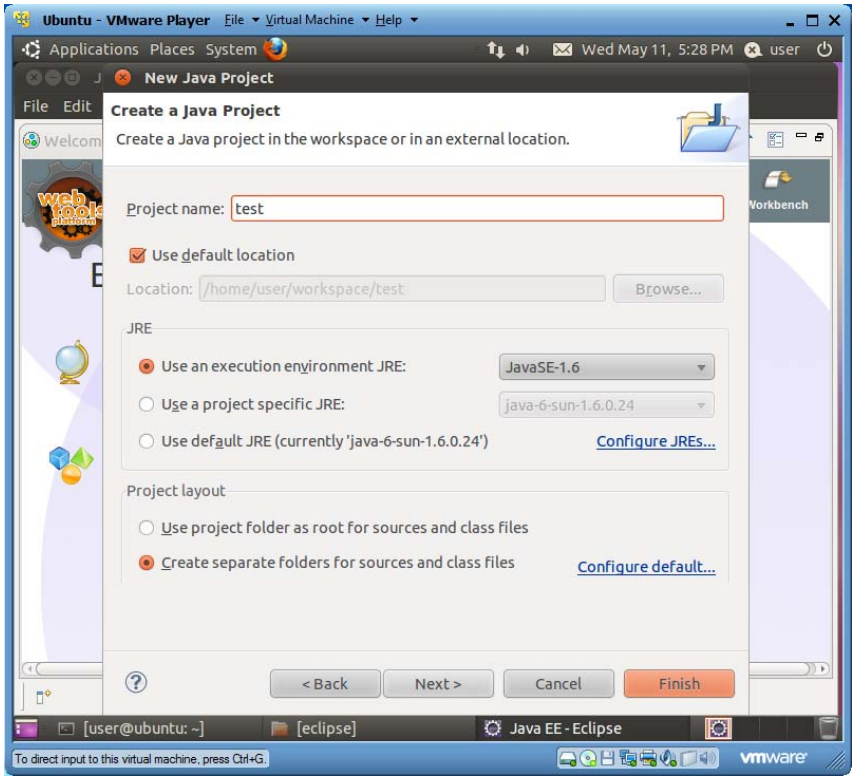
2. Accept the default project workspace “/home/user/workspace”, and check the checkbox for “Use this as the default and do not ask again”.



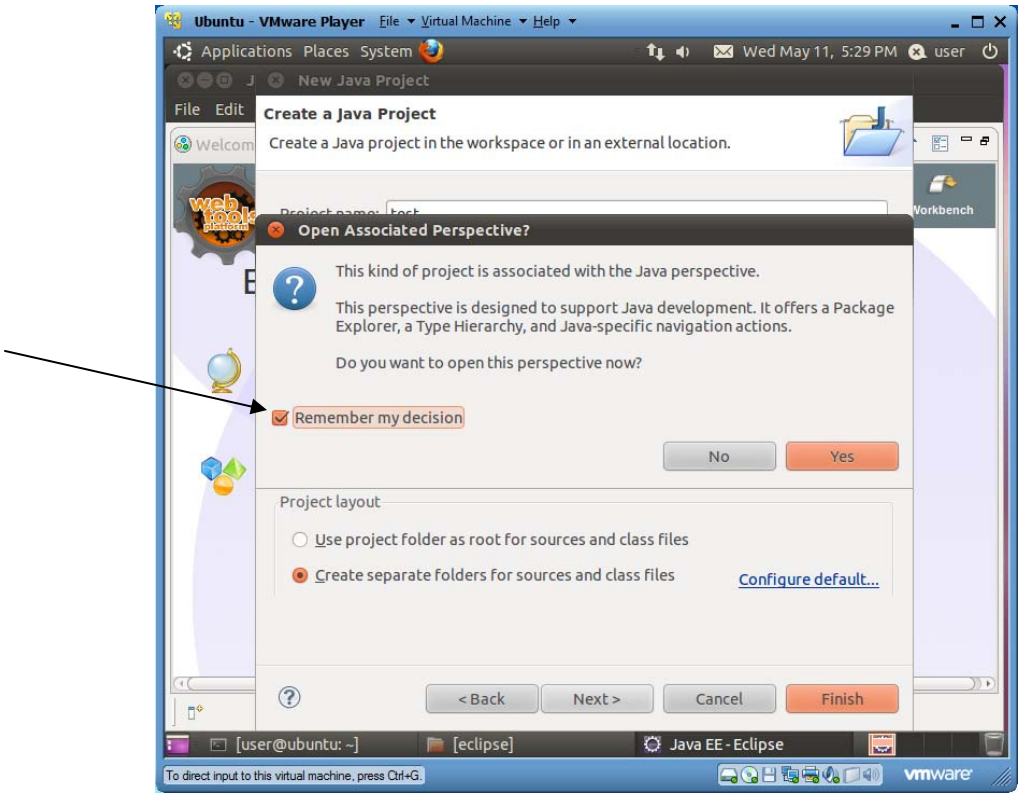

3. Let us create a simple Java project, first choose “File|New|Project…”, then choose “Java Project” to switch to Java Perspective and add “Java Project” to the *New Project* menu.



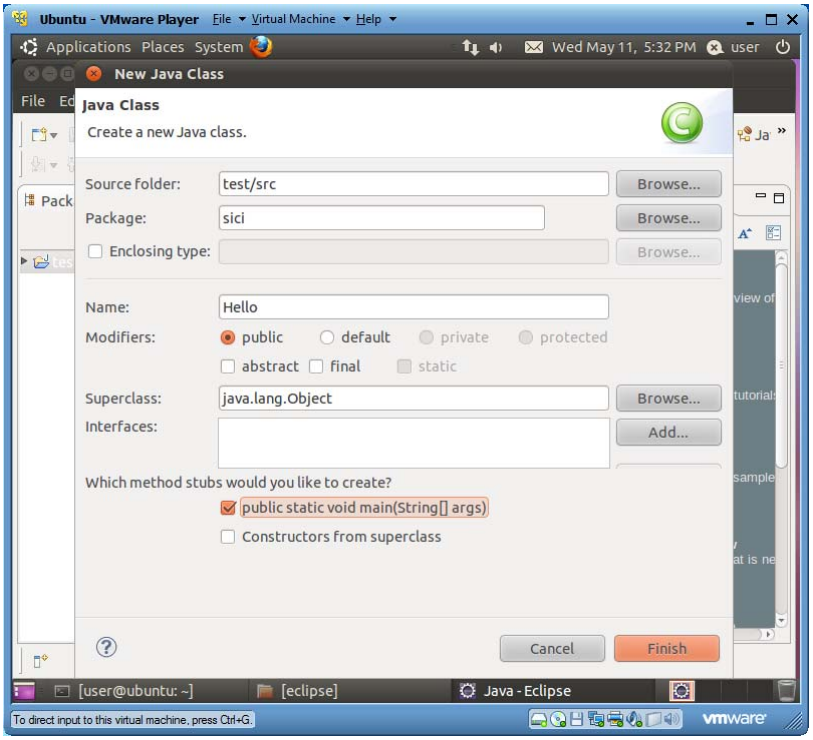

4. Click the *Next* button to get the following view, and enter “test” in “Project Name” text field. Let the “JRE” and “Location” panes keep their default values.



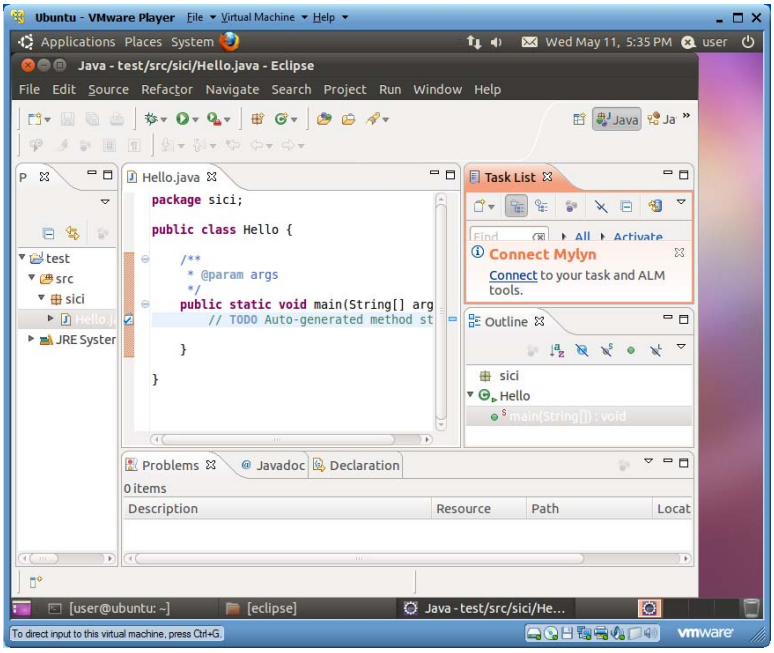

5. Click the *Finish* button to create the "test" Java project. You will then see the following screen:



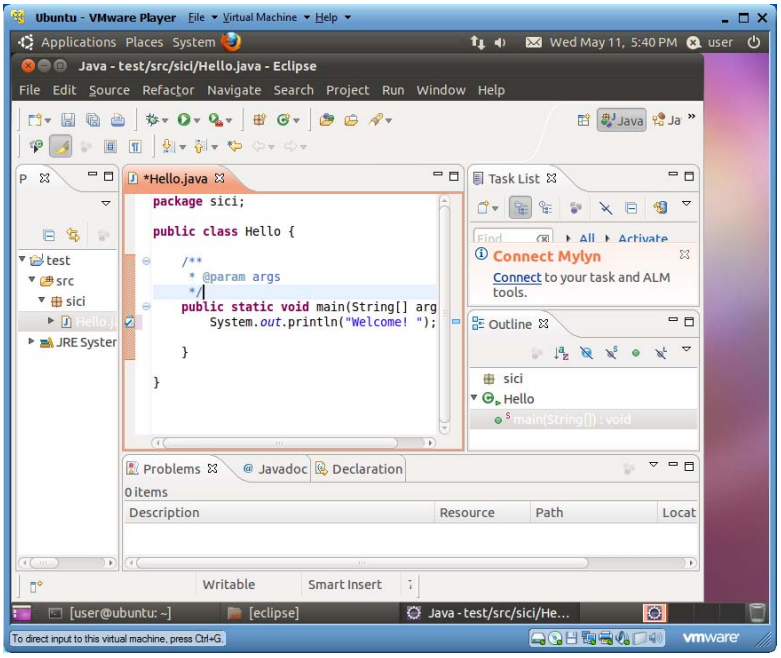
6. Check the “Remember my decision” checkbox, and click the *Yes* button. From now on your Java projects will be displayed with the Java perspective (*Eclipse* GUI features for Java).
7. Click menu item “File|New|Class” to create a new Java class source file. In the following “New Java Class” window, enter “sici” as the package name and “Hello” as the class name. Check the “public static void main” checkbox to create the “main” method in the class. (You may need to enlarge the VM window to see the checkbox.)



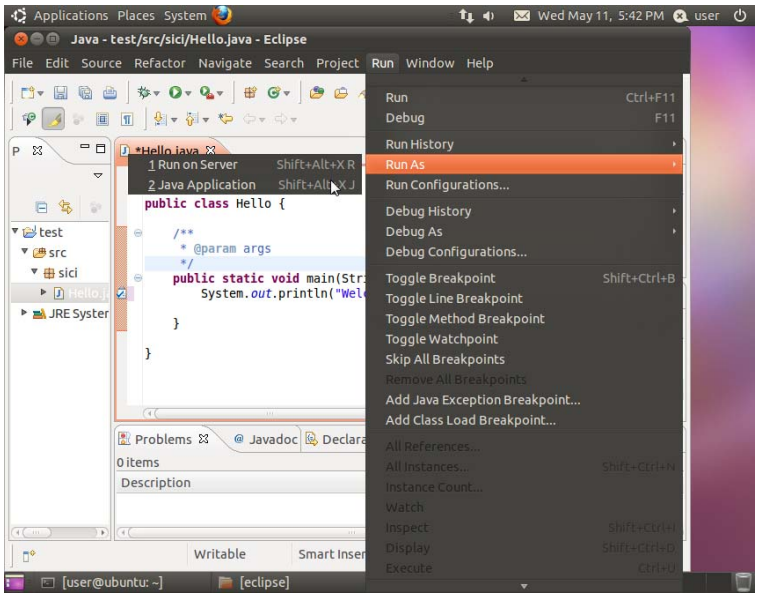

8. Click the *Finish* button and you will have the following view:



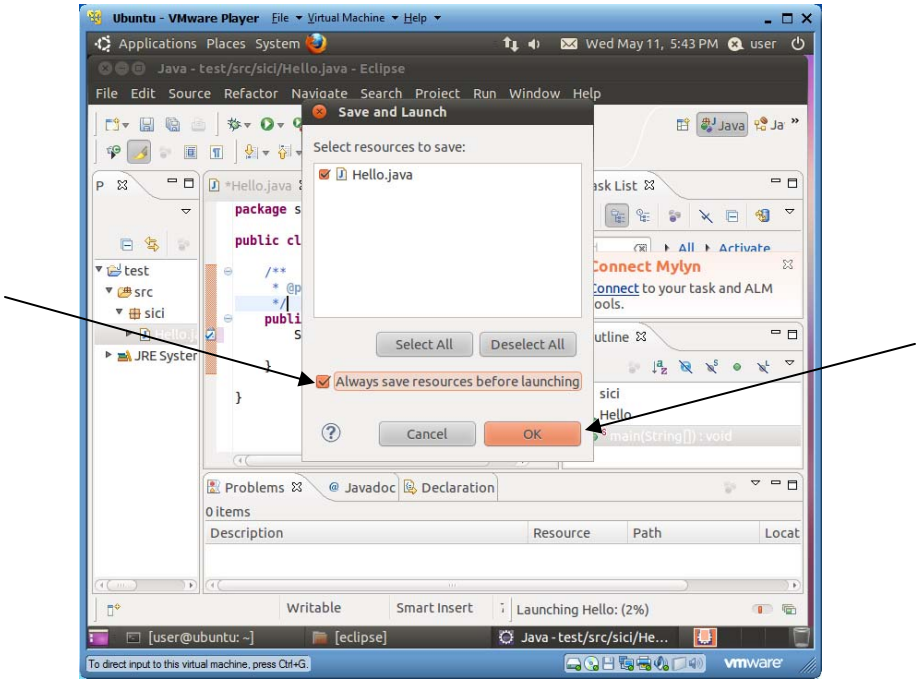
9. In the body of method "main", replace the TODO comment with Java statement "System.out.println("Welcome! ");" as shown below. (If you see a red cross to the left of the print statement it means the line has syntax error. Delete the line and type, instead of paste, it in *Eclipse*.)



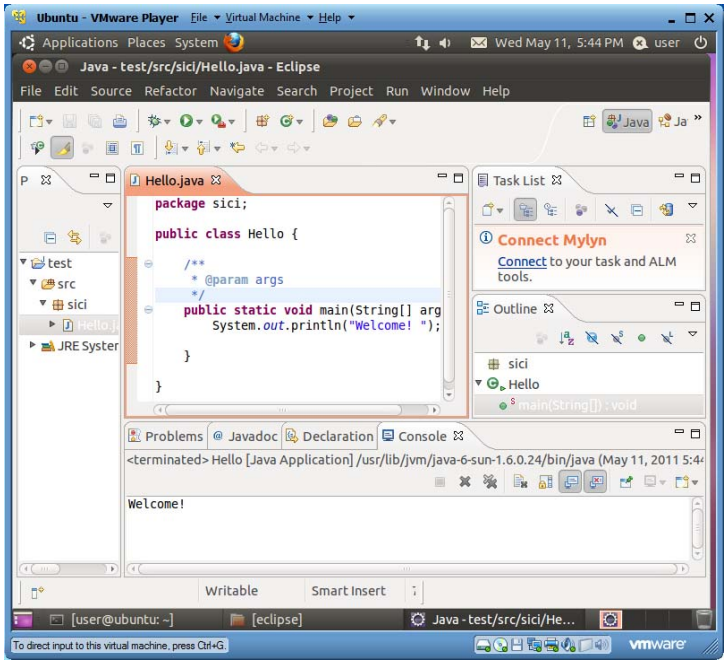

10. Click menu item "Run|Run As|Java Application" to compile and run the project.




11. *Eclipse* now asks you to confirm to save and launch the project. Check both of the two checkboxes as shown below:




12. Click the *OK* button, and *Eclipse* saves the modified source file, compiles it and runs it. The execution output is under the "Console" tab, as shown below:



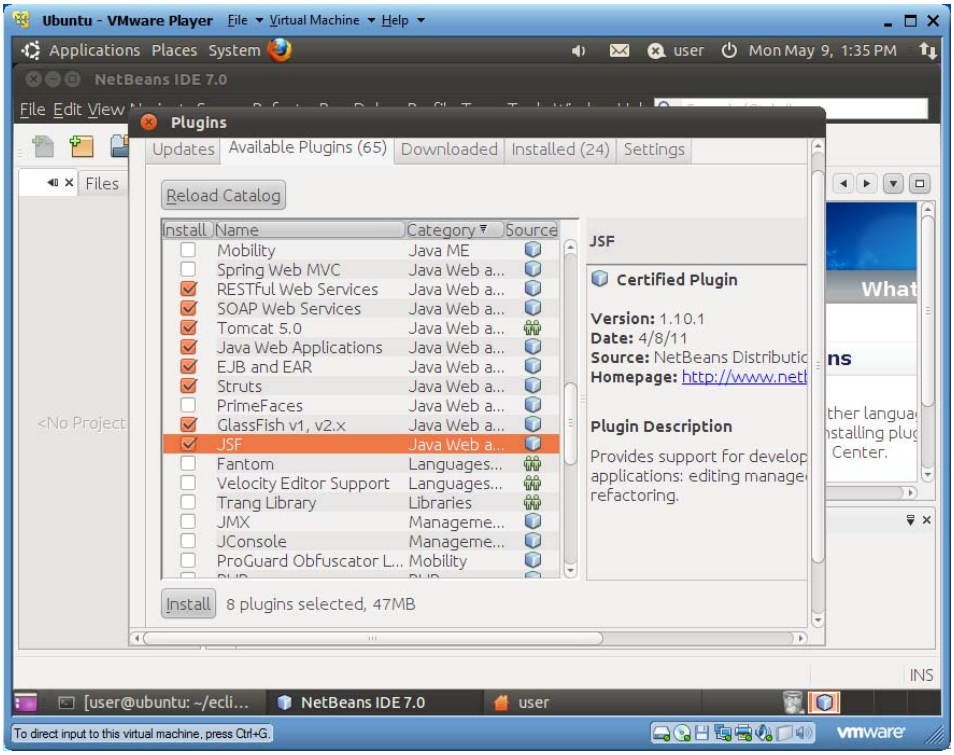

13. You can use "File|Close" to close the current project, and "File|Exit" to shut down the *Eclipse* IDE.

## 4.12 Installing Glassfish Application Server v 3.1 and Oracle's Java EE Tutorials

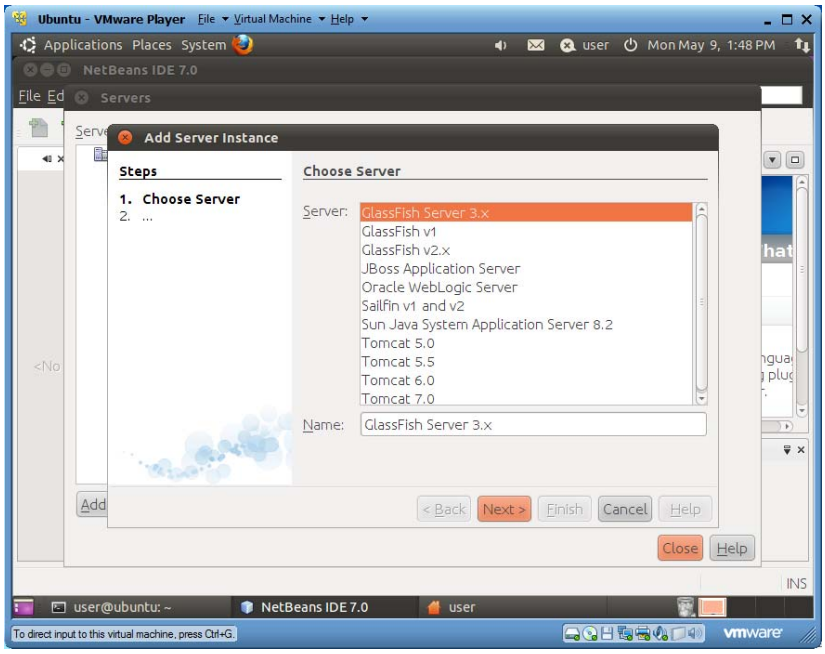
Modern enterprise computing uses application servers to support scalable execution of business logics, and in the Java world GlassFish is an open source implementation of the Java EE application server specification. This section shows how to install the latest GlassFish v3.1 application server and Oracle's Java EE tutorials through *NetBeans* which you just installed.

### *4.12.1 Adding GlassFish and Tomcat Servers to NetBeans*

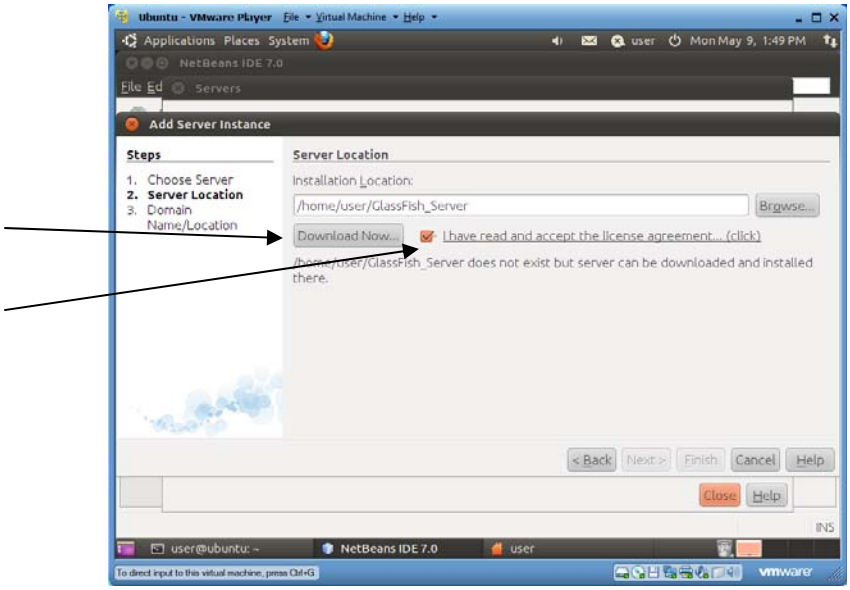
1. Launch *NetBeans*. Use menu item "Tools|Plugins" to display the following plugin window. Select the "Available Plugins" pane. Check the plugins for "RESTful Web Services", "SOAP Web Services", "Tomcat 5.0", "Java Web Applications", "EJB and EAR", "Struts", "GlassFish v1, v2.x", and "JSF", as show below.



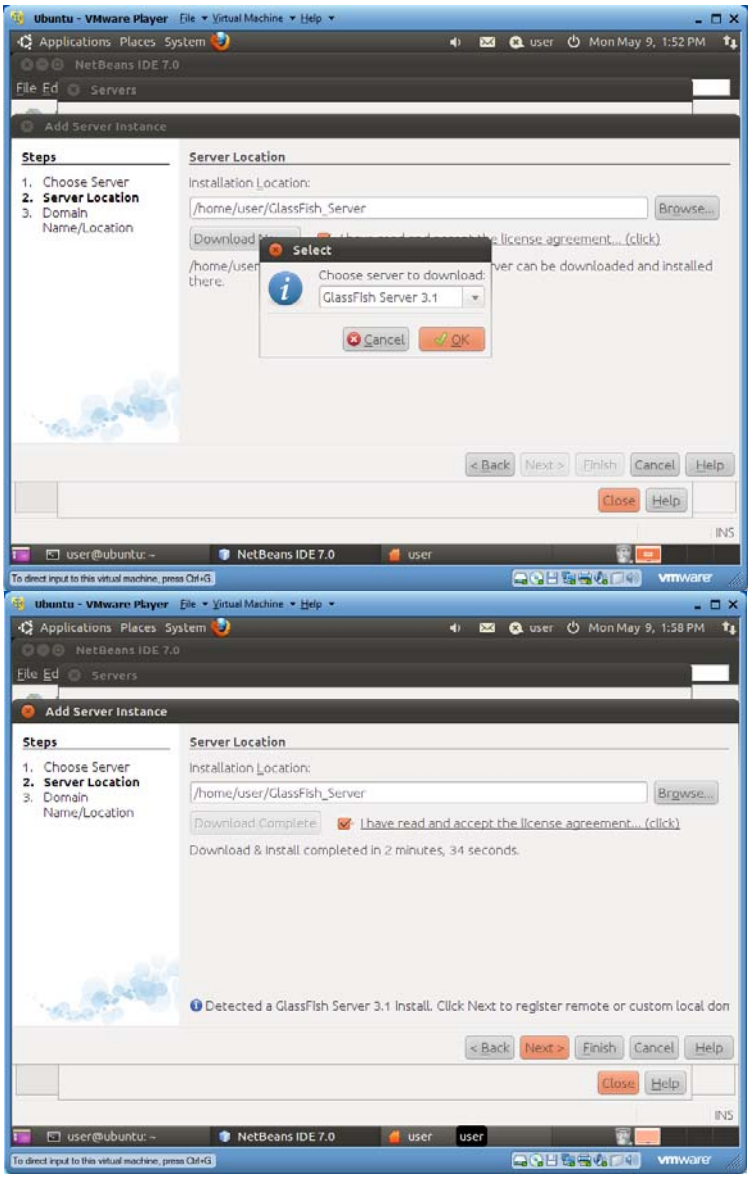

2. Click the *Install* button, then the *Next* button, check "I accept …", *Next* again, and *Finish*.
3. Use menu item "Tools|Servers" to display the *Servers* window. Click button "Add Server…", and select "GlassFish Server 3.x".



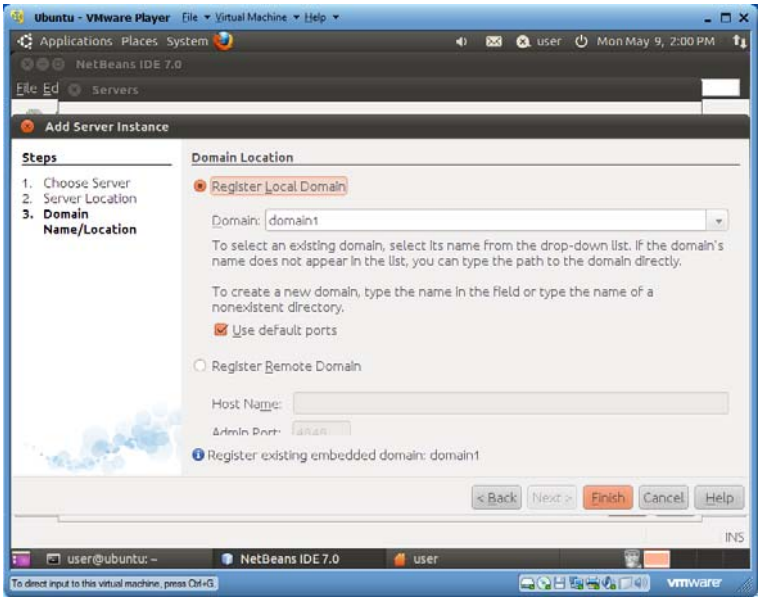

4. Click the *Next* button. Check “I have read and accept …”, and click the “Download Now…” button.



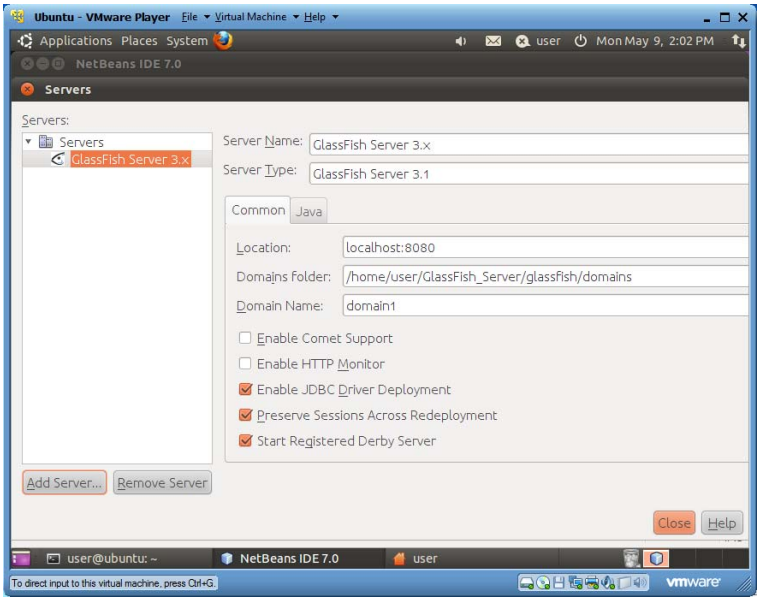

5. Select “GlassFish Server 3.1”, and click the “OK” button.



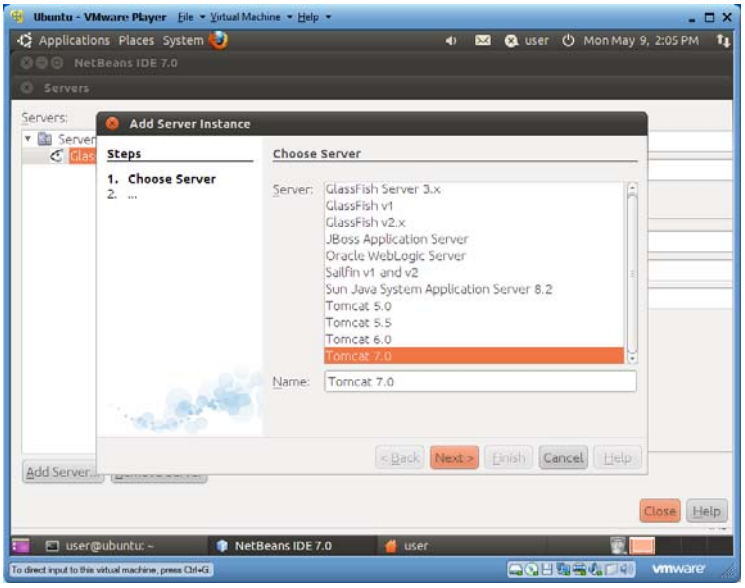

6. After the download is complete, click the *Next* button.



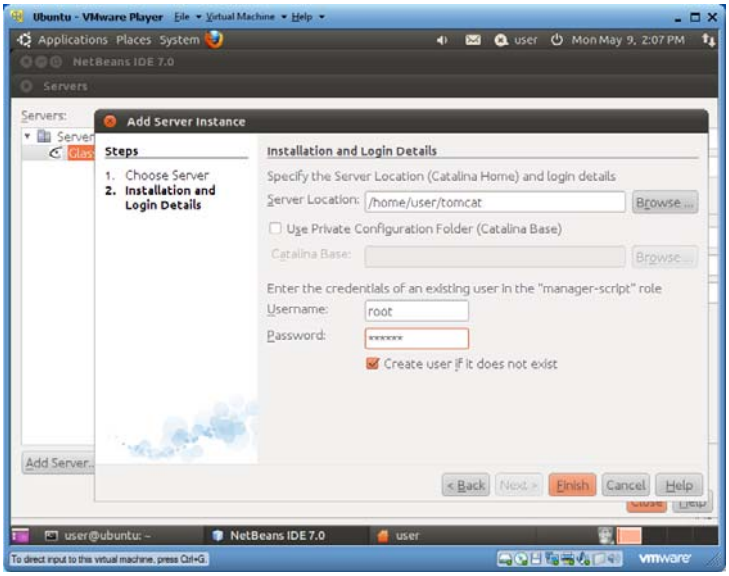

7. Accept the default values, and click the *Finish* button.



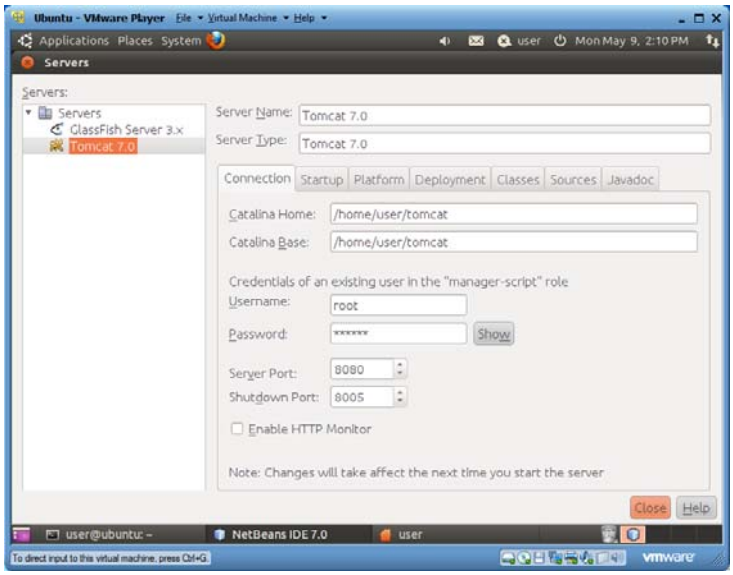

8. You have completed adding “GlassFish v3,1” to *NetBeans* as a project deployment platform. Now you will also add *Tomcat* to *NetBeans* so you can use *NetBeans* to develop web applications on *Tomcat*. Click the “Add Server…” button.




9. Select “Tomcat 7.0” and click the *Next* button.



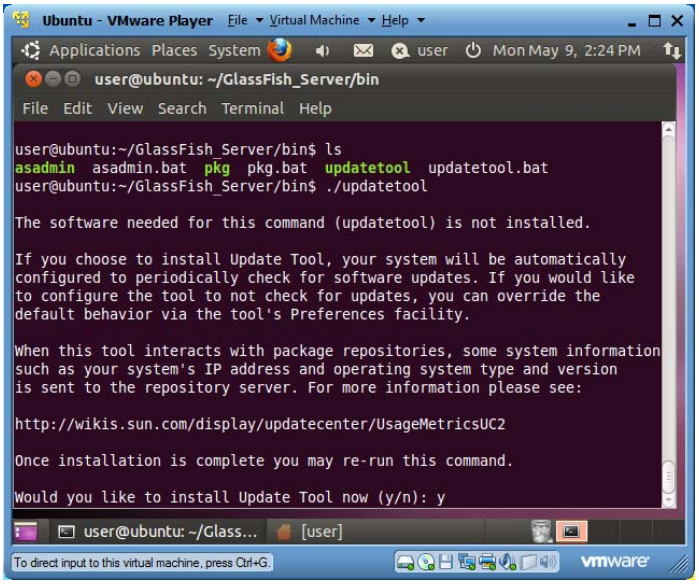

10. Enter or browse for “/home/user/tomcat” for server location, enter “root” and 123456 for user name and password, and then click the *Finish* button.




11. Click the *Close* button to complete adding *GlassFish* and *Tomcat* servers to *NetBeans*. From now on, when you create a web application, you can choose whether to develop the project on *GlassFish* v3.1 or on *Tomcat*.
12. Shut down *NetBeans*.
13. In file explorer you will find that a new folder, "GlassFish_Server", has been created to host the *GlassFish* application server.

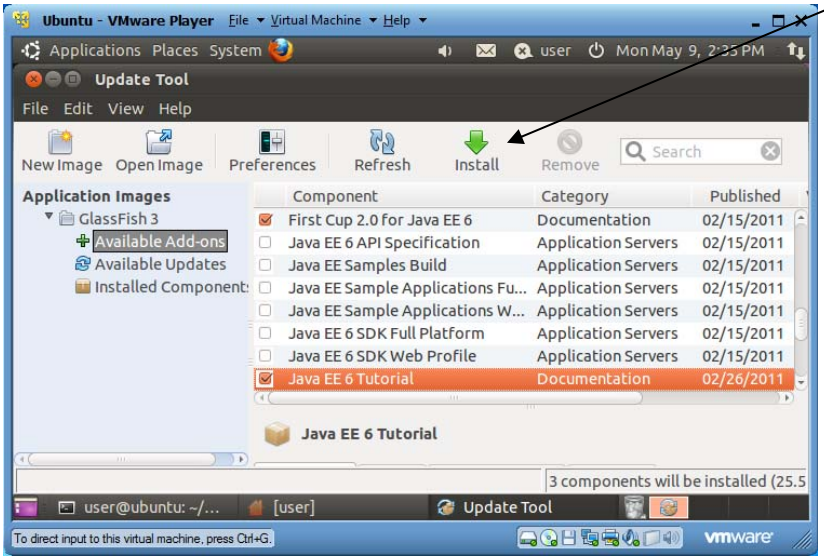
### *4.12.2 Installing Java EE Tutorials through Java Update Tool*

1. Launch a terminal window in "~/GlassFish_Server/bin".
2. Run "./updatetool"




3. Enter "y" to install Java update tool.

4. Run "./updatetool" again. In the new *Update Tool* window, select "Available Add-ons".

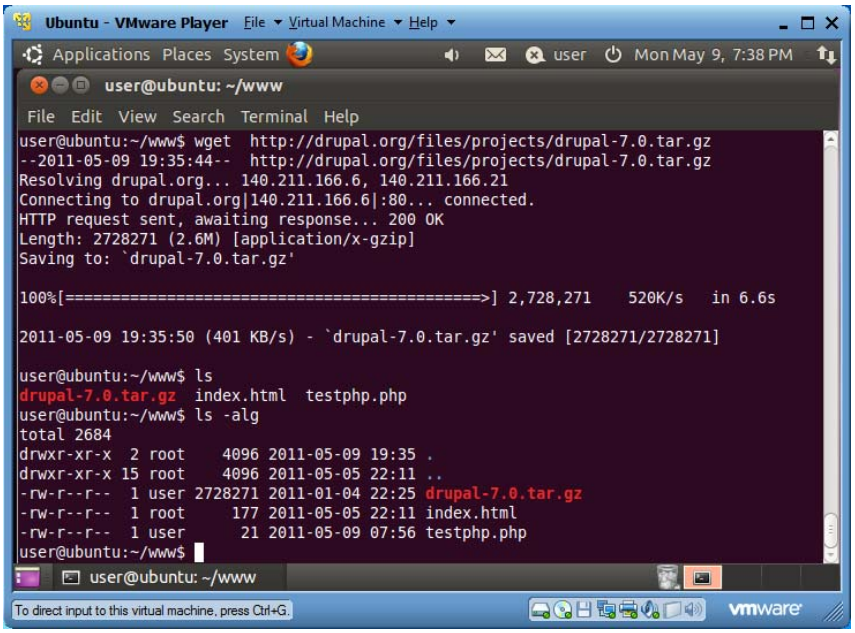



6. Select "GlassFish Ant tasks", "Fisrt Cup 2.0 for Java EE 6", and "Java EE 6 Tutorial", and then click the top "Install" icon. Accept the terms to complete the installation.
7. Shut down the *Update Tool* window.
8. The Oracle Java EE overview tutorial, "Your Fist Cup: An Introduction to the Java EE Platform", is installed in folder "~/GlassFish_Server/glassfish/docs/Firstcup-2.0".
9. The Oracle Java EE v6 tutorial is installed in folder "~/GlassFish_Server/glassfish/docs/javaee-tutorial".

## 4.13 Setting Up a Drupal Website on Apache

Drupal is a powerful open source contents management system supporting a rich repertoire of social networking tools. This section guides you to set up a Drupal site from scratch.

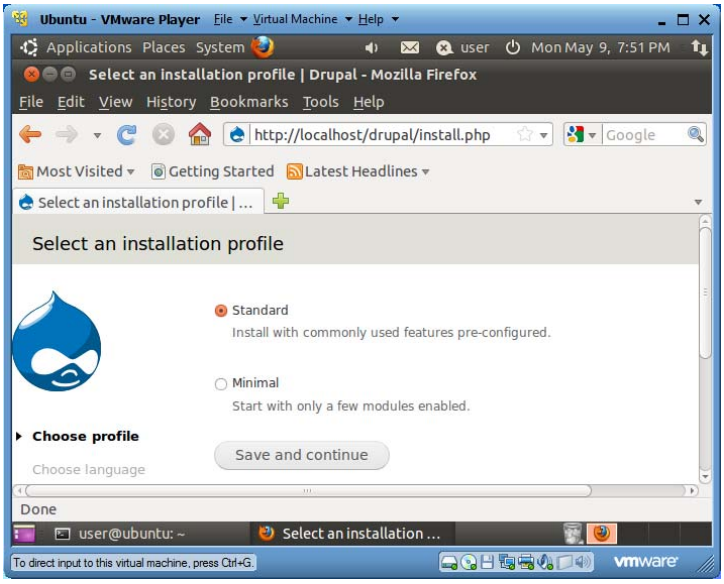
1. Run command "sudo apt-get install php5-mysql" to install PHP5 support for MySQL.
2. Run "sudo apt-get install php5-gd" to install PHP5 module supporting *GD* image library.
3. Reboot the VM or run "sudo apache2ctl restart" to restart Apache.
4. Run "cd ~/www" to change the terminal working folder to "~/www".
5. Run "wget http://drupal.org/files/projects/drupal-7.0.tar.gz" to download Drupal v7.0 installer "drupal-7.0.tar.gz". The installer is also in your PC's folder "C:\VM\resources".



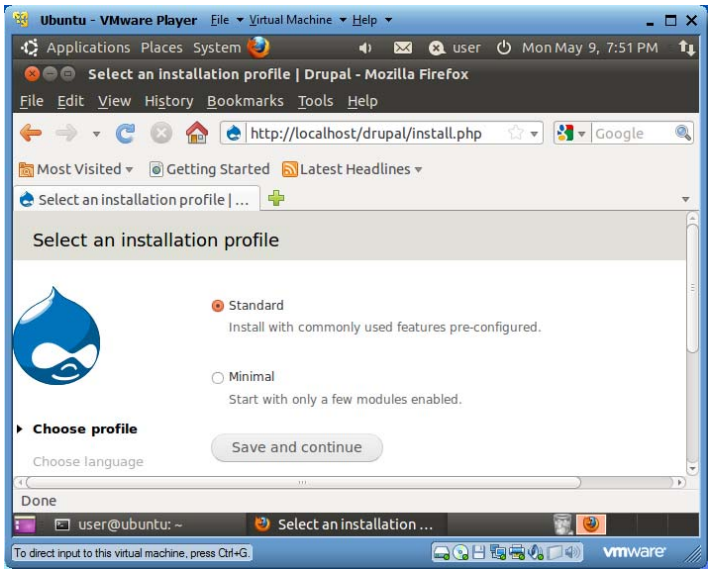

5. Run “tar -zxvf drupal-7.0.tar.gz” to unzip the installer file into folder “drupal-7.0”.
6. Run “mv drupal-7.0 drupal” to rename the folder as “drupal”. You can now run “rm drupal-7.0.tar.gz” to delete the Drupal installer.
7. Run “cd drupal/sites/default” to change working folder.
8. Run “cp default.settings.php settings.php” to make a copy of file “default.settings.php” and name it “settings.php”.
9. Run “chmod a+w settings.php” to give the web server write privileges to the configuration file.
10. Run “cd ..” to move working folder up one level. Then run “chmod a+w default” to give the web server write privileges to the default folder.
11. Run “mysql -u root -p123456” to launch the MySQL admin console.
12. Run “create database drupal;” in the console to create database “drupal”, and then run “quit;” to quit the MySQL admin console.

```
user@ubuntu:~$ mysql -u root -p123456
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 37
Server version: 5.1.49-1ubuntu8.1 (Ubuntu)

mysql> create database drupal;
Query OK, 1 row affected (0.00 sec)

mysql> quit;
Bye
user@ubuntu:~$
```

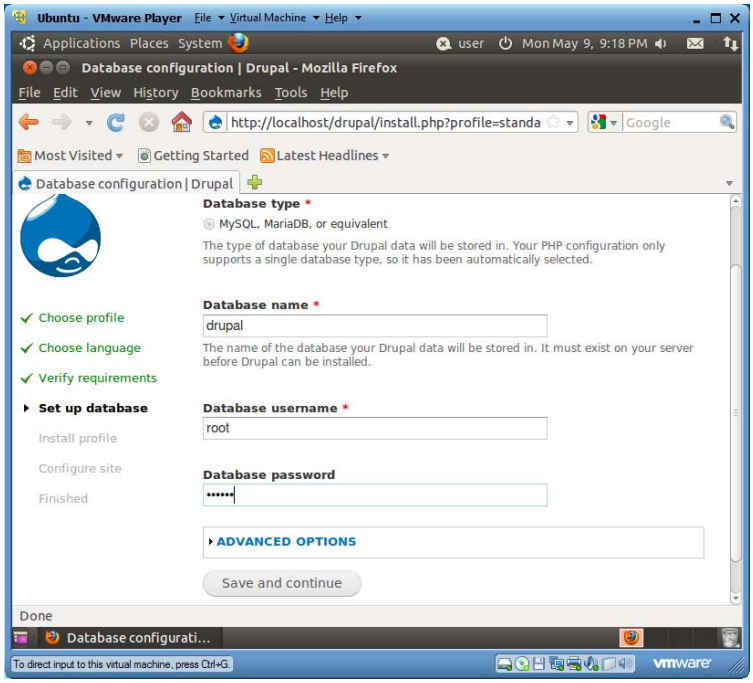
13. Launch a web browser to visit http://localhost/drupal/



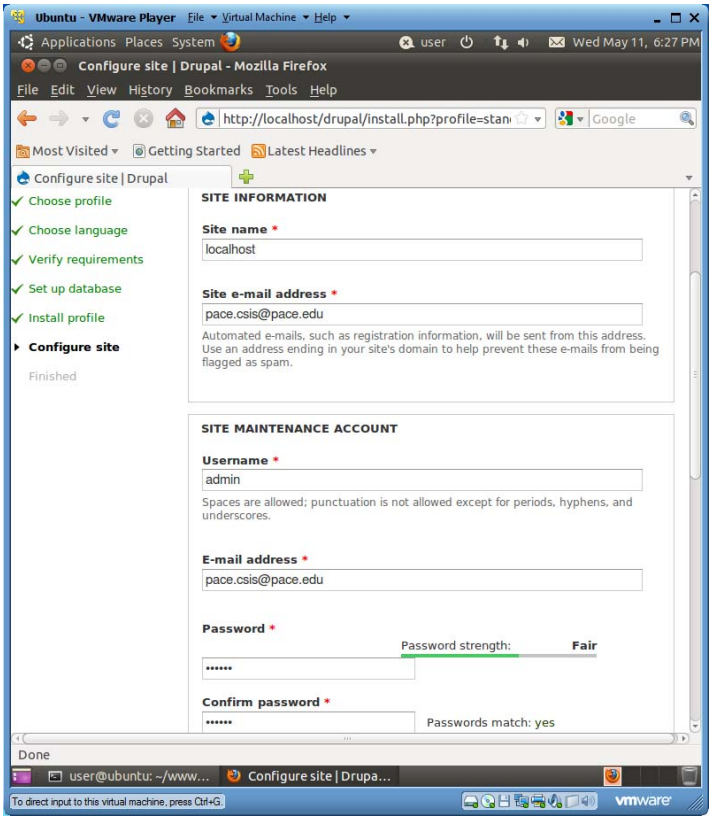

14. Make sure “Standard” is selected. Click “Save and continue”.



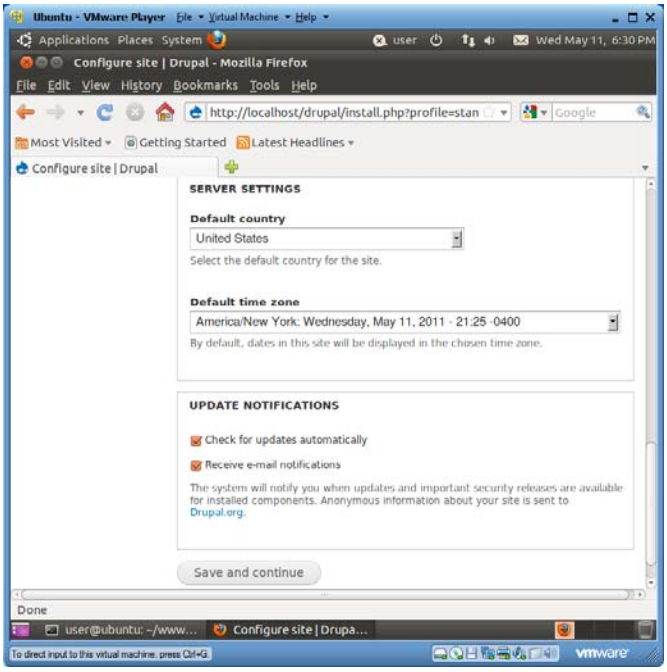

15. Click “Save and continue” to accept “English” as the default language.
16. Review the requirements verification page. Resolve any problems reported on this page. Then click “Save and continue”.



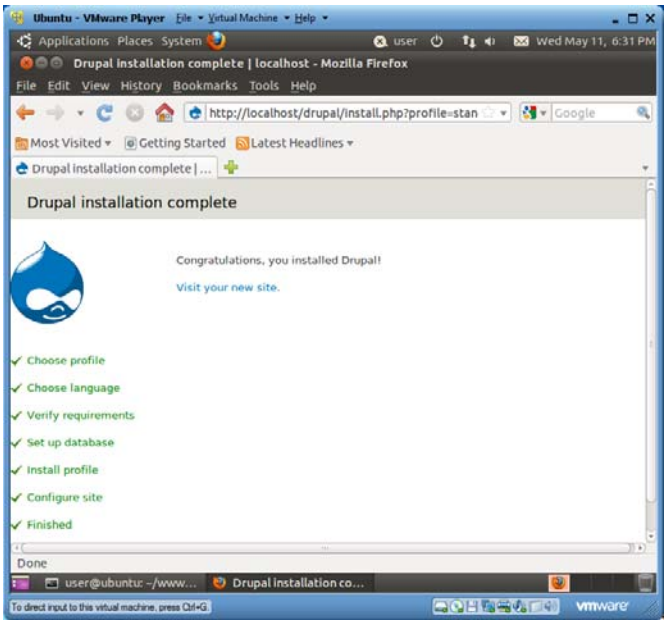

17. Enter “drupal” as database name, “root” as database user, and 123456 as database password. Then click “Save and continue”.



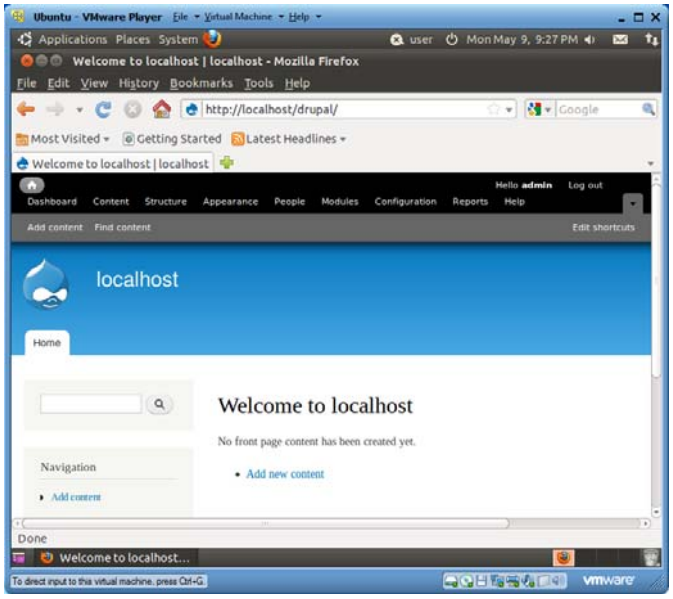
18. In the *Site Information* section of the "Configure site" pane, keep "localhost" as site name, your valid email address as site email address. In the *Site maintenance Account* section, enter "admin" as user name, and 123456 as password.



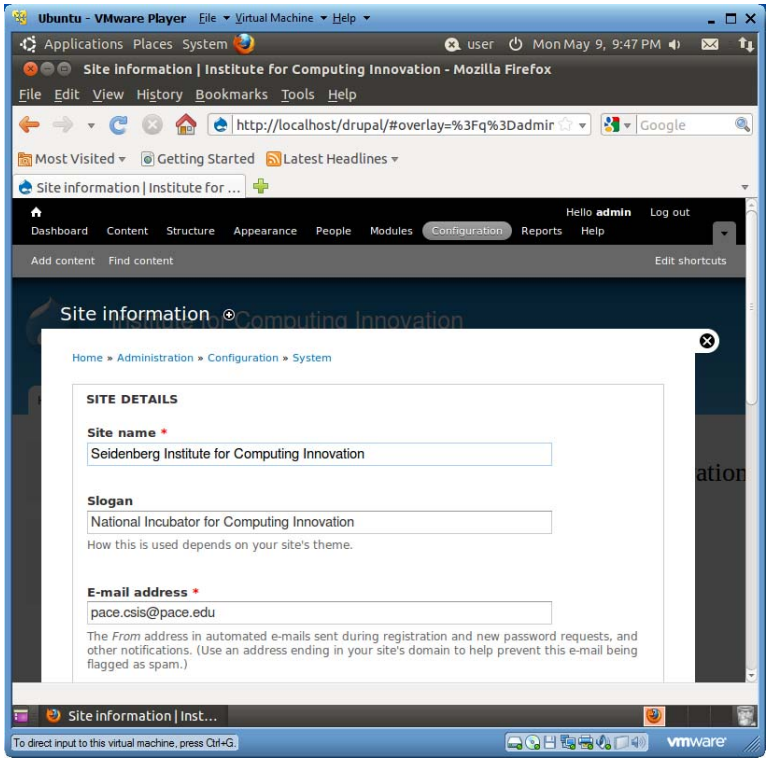

19. In the *Server Settings* section, fill in appropriate information, and then click "Save and continue".



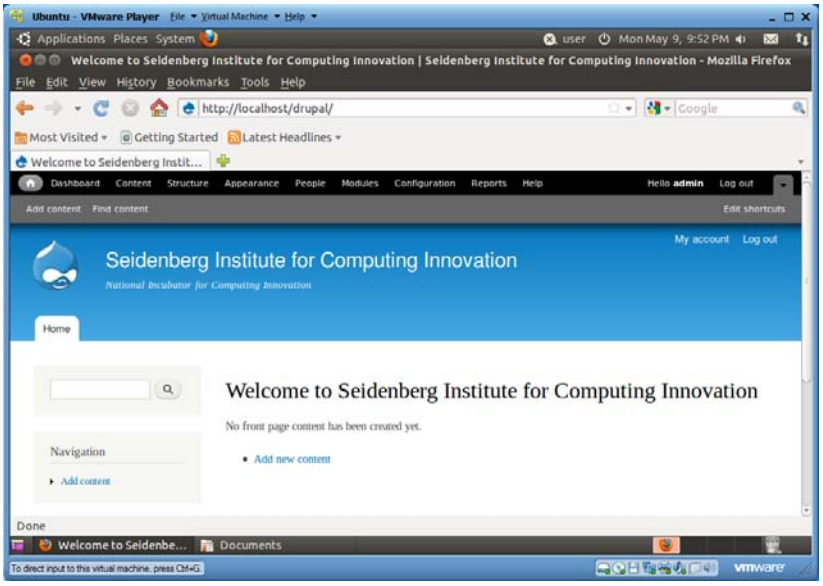

20. Click the "Visit your new site" link. Your Drupal installing is now complete.



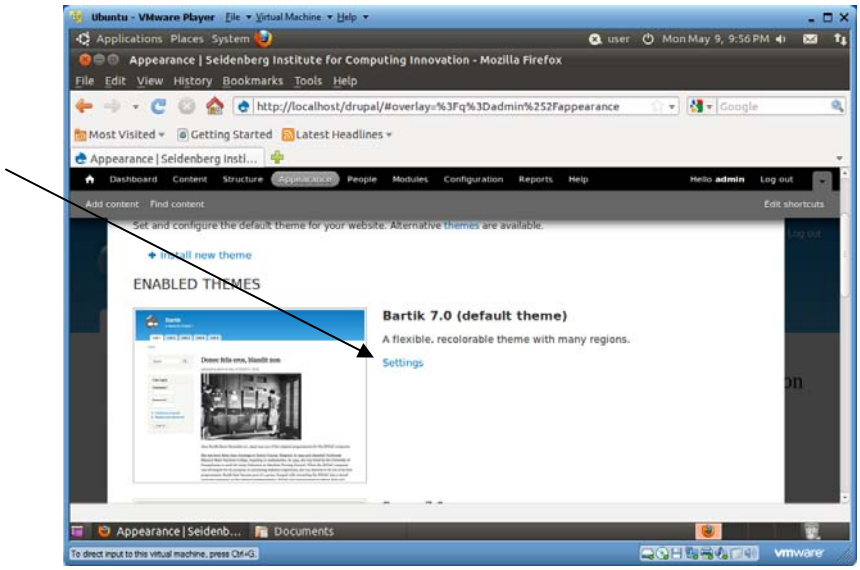

21. Run “chmod a-w ~/www/drupal/sites/default/settings.php” to make file “settings.php” not writable. Otherwise you will see security warning in Drupal.
22. Run “chmod a-w ~/www/drupal/sites/default” to make folder “default” not writable. Otherwise you will see security warning in Drupal.
23. Click the “Configuration” tab.
    a. Click “Site information”, and enter “Seidenberg Institute for Computing Innovation” for site name, and “National Incubator for Computing Innovation” as slogan.



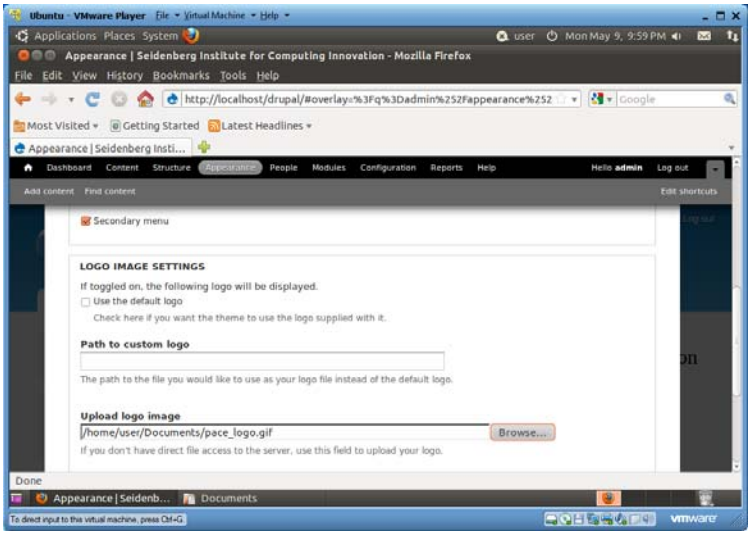

    b. Scroll down to the bottom of the page, and click button “Save configuration”.



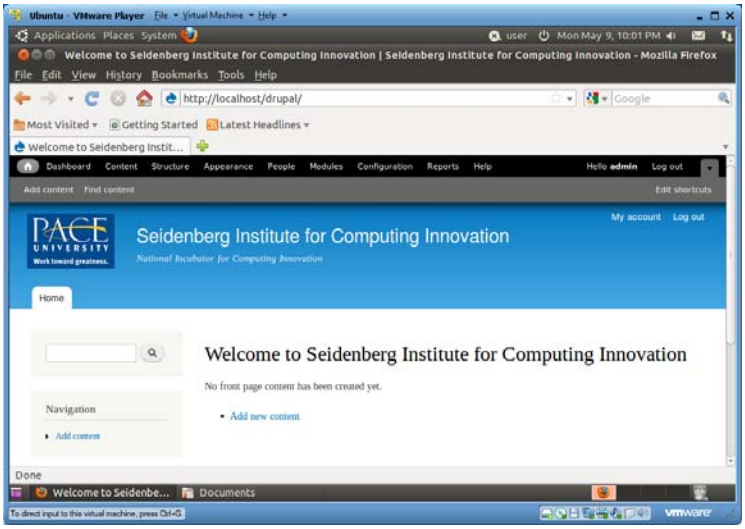

24. Drag and drop file “pace_logo.gif” from your PC’s folder “C:\VM\resources” to your VM’s folder “~/Documents”.
25. Click the “Appearance” tab.



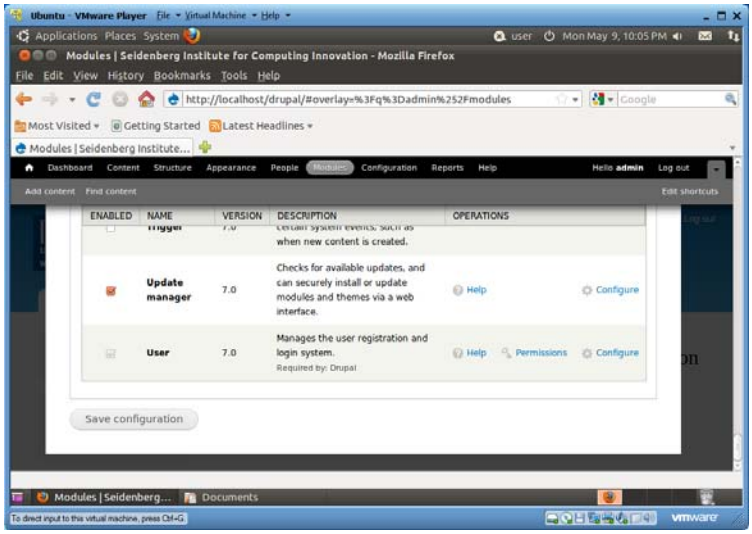

26. Click the “Settings” link.
27. In the “Appearance” page, select the “Global settings” tab in the upper-right corner, uncheck “Use the default logo”, then browse for file “~/Documents/pace_logo.gif” for uploading.



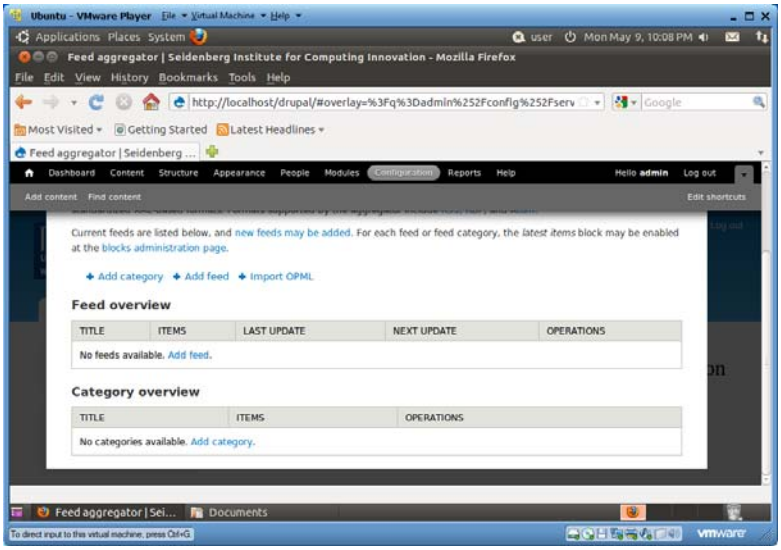

28. Scroll to the page bottom, and click “Save configuration”. Now the Drupal home page looks as below.



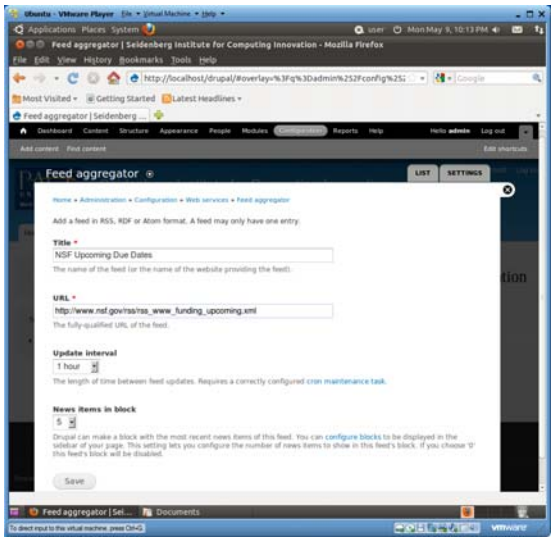

29. Click the “Modules” tab. Check “Aggregator”, “Blog”, “Forum”, “Poll”, and click the bottom button “Save configuration”.



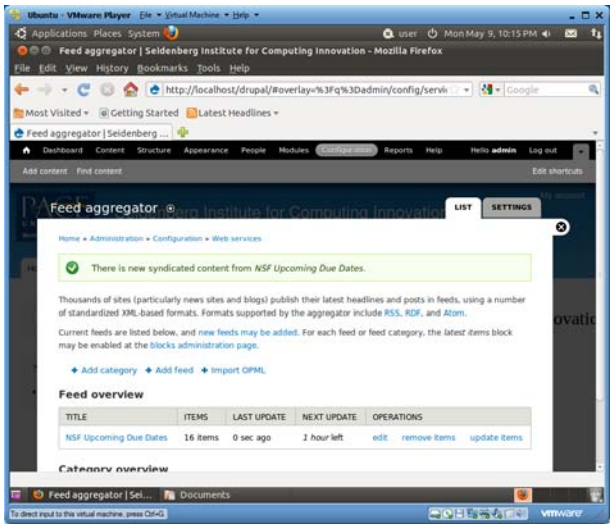

30. Click the “Modules” tab. Click “Feed aggregator” at the bottom right of the page.



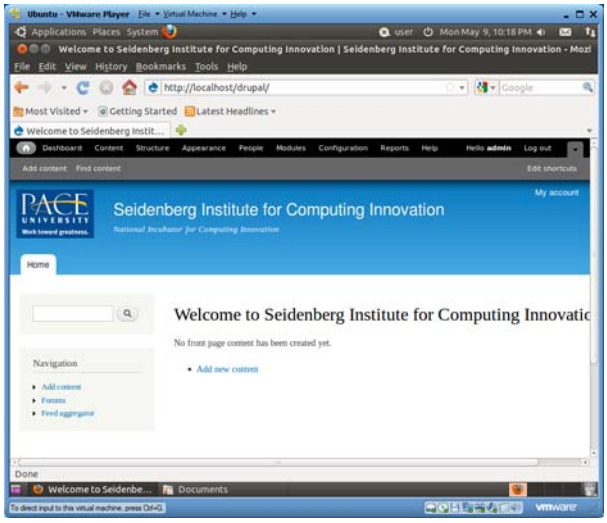

31. Now you will add news feeds for NSF upcoming due dates. Click the “Add feed” link.



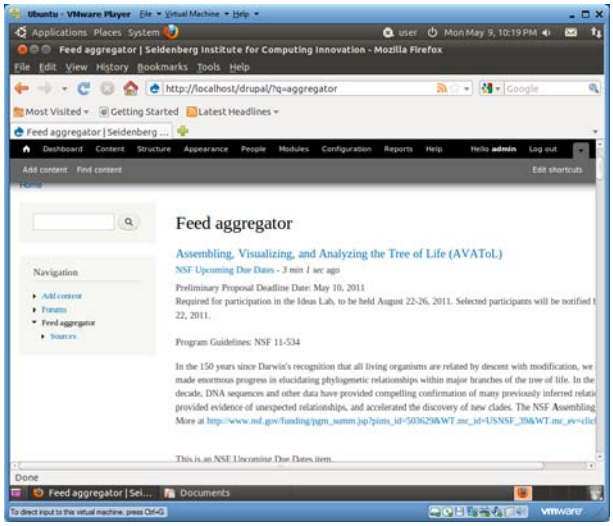

32. Enter "NSF Upcoming Due Dates" for *Titles*, and http://www.nsf.gov/rss/rss_www_funding_upcoming.xml for *URL*. Click "Save" at the bottom.
33. Select the "LIST" tab in the upper-right corner, and click the "update items" link to download the latest feeds for NSF upcoming due dates.



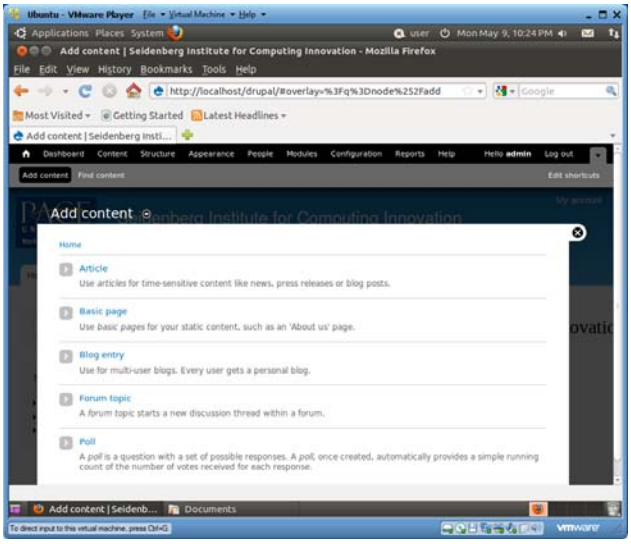

34. Click the house icon to go back to the Drupal site home page.



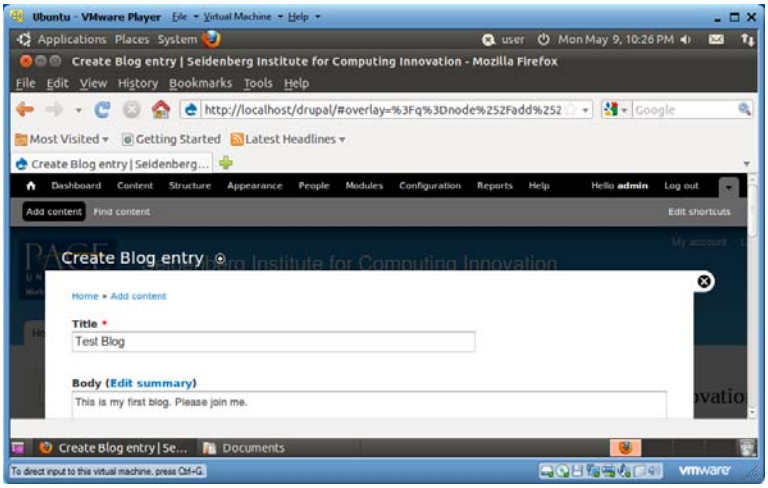

35. Click the “Feed aggregator” link in the *Navigation* menu to read the latest news feeds.



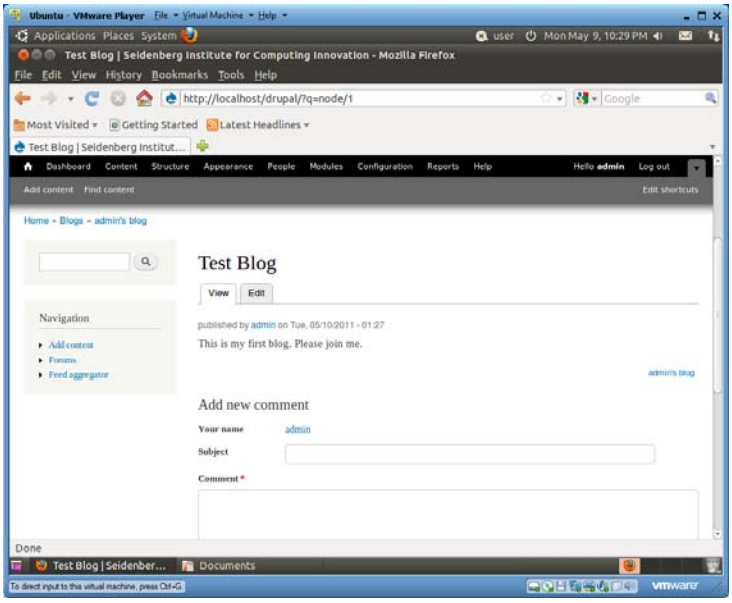

36. Click the “Add content” link in the *Navigation* menu, and choose “Blog entry” in the “Add content” page.



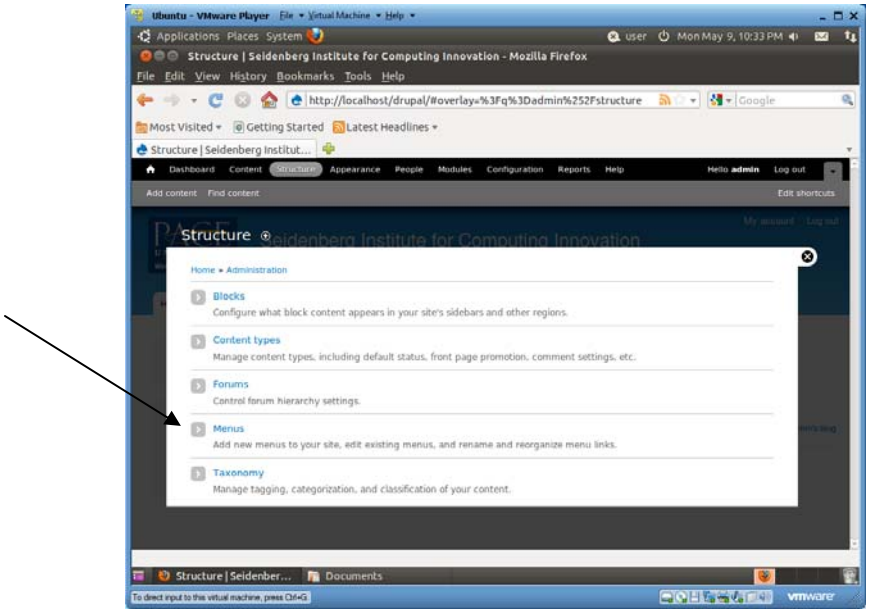

37. Type any contents as the blog's title and body.



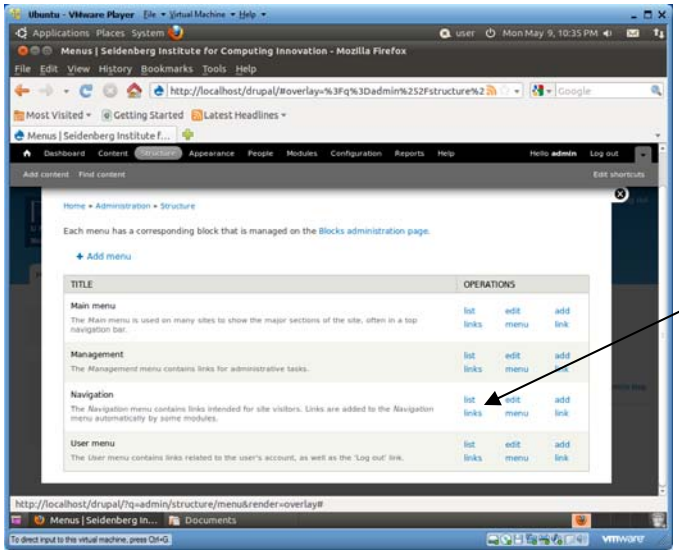

38. Scroll down to the bottom of the page, and click "Save".



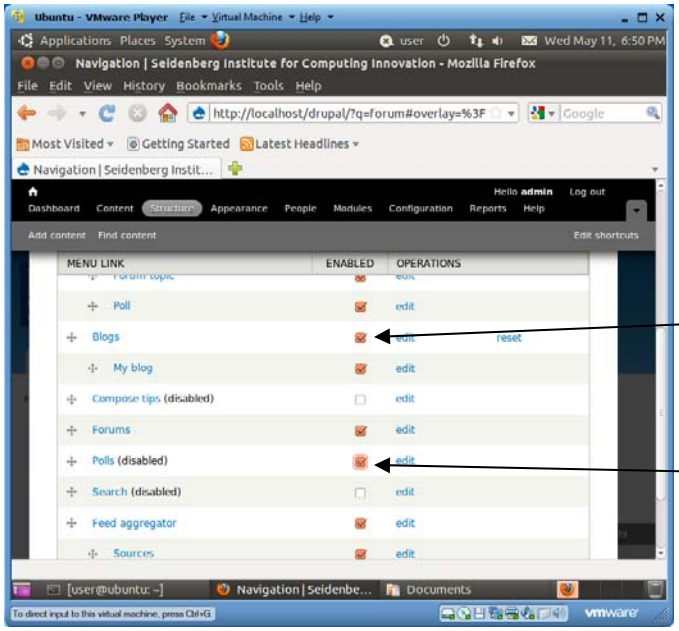

39. Now your first blog has been published and is ready for people to post comments.
40. To add links to blogs and polls in the *Navigation* menu, click the "Structure" tab, and then click the "Menus" link.



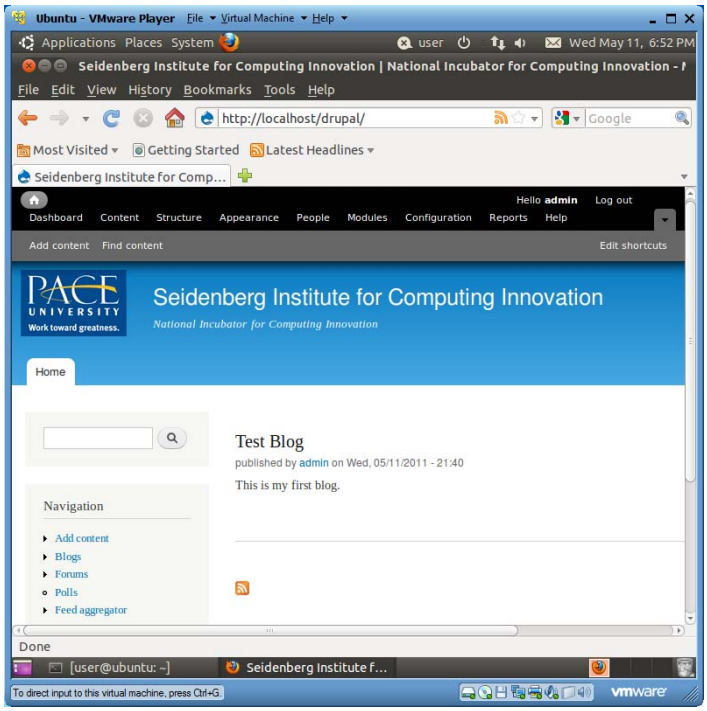

41. Click the "list links" link for *Navigation*.




42. Check “Blogs” and “Polls", and then scroll to the bottom of the page to click “Save configuration”.




43. Click the house icon to go back to Drupal’s home page. You can see that the *Blogs* link has been added to the *Navigation* menu.

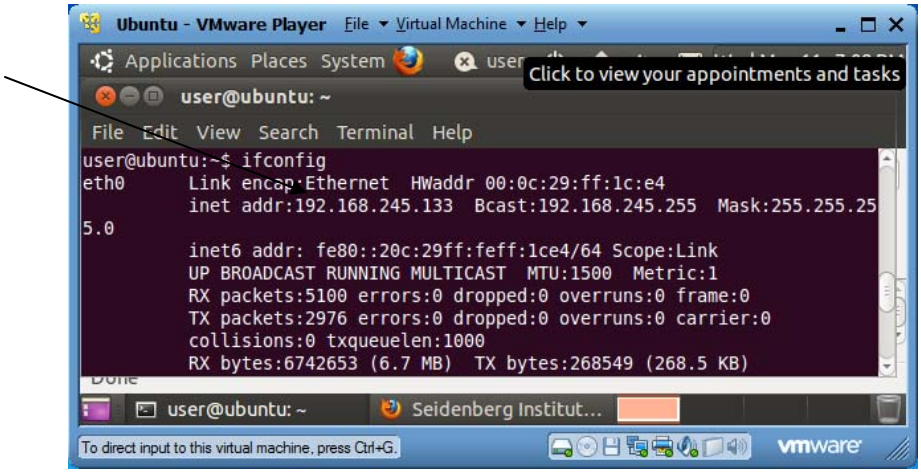



44. Now you can click “Add content”, and choose to add a “Forum topic”. You can now create your first discussion forum post.
45. Now you can click “Add content”, and choose to add a “Poll”. You can now create your first poll.

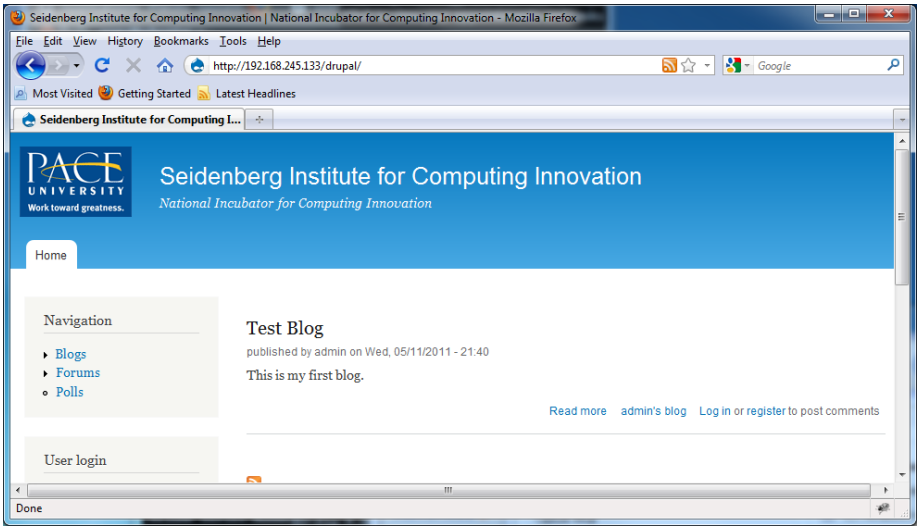
## 4.14 Networking between Host and VM

Here you will learn how to find the IP address of your VM and how to use the host PC’s web browser to visit your new Drupal site. You can generalize what you learn here and let a set of multiple VMs and the host communicate with each other as in typical enterprise IT systems.

1. Run “ifconfig” to find your VM’s IP address.



2. Your VM's IP address is specified as the "inet addr" value. In the example above, the VM has 192.168.245.133 as its IP address.
3. In your host PC, use a web browser to visit http://192.168.245.133/drupal/ (adjust the IP address to the IP address of your VM found in the previous step) to see a web page similar to the following.

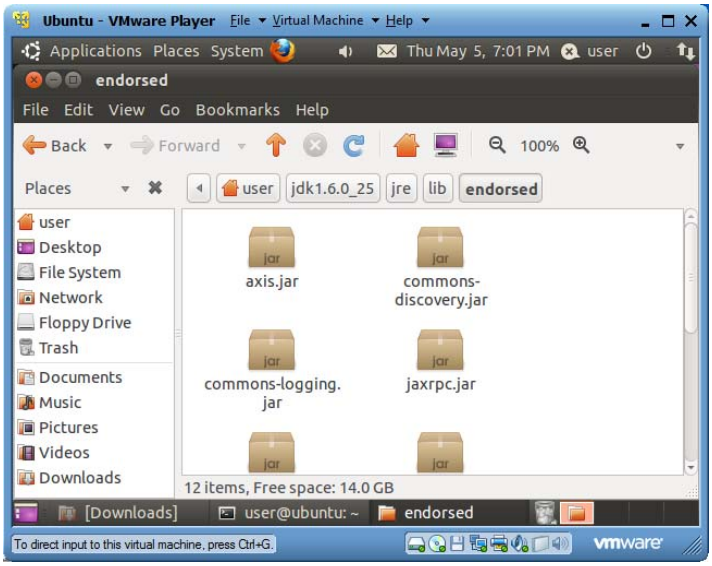


# 5 Getting Ready for the Other SICI 2011 Labs

## 5.1 Setting Up for the "Introduction to Web Services" Lab

To work on the "Introduction to Web Services" lab, you need to set up the lab's working folder as well as the necessary Java libraries.

1. Drag and drop folder "webServiceLab" from your PC's folder "C:\VM\resources" to your VM's home folder ~.
2. Drag and drop folder "endorsed" in "C:\VM\resources" of your PC to your VM's folder "~/user/jdk1.6.0_25/jre/lib". Java classes in the jar (Java Archive/zip) files in folder *endorsed* replace any other implementations of these classes and are available to all users. [Security sensitive folder]




## 5.2 Setting Up for the "Introduction to Cryptography" Lab

To work on the "Introduction to Cryptography" lab, you need to install the *GnuPG-Agent* package.

1. Start a terminal window with menu item "Applications|accessories|Terminal".
2. In the terminal window, run "sudo apt-get install gnupg-agent". When asked for password for user, enter 123456. The GnuPG-Agent installation will complete in a few seconds.
3. Run "gpg" to observe that a few *gpg* configuration files/folders are created, and type Ctr-D to terminate the *gpg* execution.

## 5.3 Setting Up for the "Introduction to Web Technologies" Lab

To work for the "Introduction to web Technologies" lab, you need to set up the *paros* folder.

1. Drag and drop folder "paros" from your PC's folder "C:\VM\resources" to your VM's home folder ~.

## 5.4 Setting Up for the "Introduction to Java Security" Lab

To work for the "Introduction to Java Security" lab, you need to set up the *JavaSecurityLab* folder.

1. Drag and drop folder "JavaSecurityLab" from your PC's folder "C:\VM\resources" to your VM's home folder ~.

The VM built by following the instructions so far can be downloaded from http://community.seidenberg.pace.edu/files/sici2011/ubuntu2.exe.

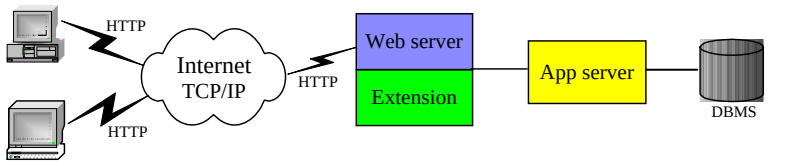
# 6 Introduction to Web Technologies

## 6.1 Concepts

Web servers and web browsers are communicating client-server computer programs for distributing documents and information, generally called web data, and delivering business services over the Internet. Web data are marked up in the HTML language for presentation and interaction with people in web browsers. Each web server uses an IP address or domain name as well as a port number for its identification. People use web browsers to send data requests to web servers with the HTTP protocol, and the web servers running on server computers either retrieve the requested data from local disks or generate the data on-the-fly, mark up the data in HTML, and send the resulting HTML files back to the web browsers to render. *Apache*, *Tomcat* and *IIS* are popular web server programs, and *IE* and *Firefox* are popular web browsers.

### *6.1.1 Web Architecture*

A typical web application involves four tiers as depicted in the following web architecture figure: web browsers on the client side for rendering data presentation coded in HTML, a web server program that generates data presentation in HTML, an application server program that computes business logic, and a database server program that provides data persistency. The three types of server programs may run on the same or different server machines.




Web browsers can run on most operating systems with limited hardware or software requirement. They are the graphic user interface for the clients to interact with web applications. The basic functions of a web browser include:

- Interpret HTML markup and present documents visually;
- Support hyperlinks in HTML documents so the clicking such a hyperlink can lead to the corresponding HTML file being downloaded from the same or another web server and presented;
- Use HTML form and the HTTP protocol to send requests and data to web applications and download HTML documents;
- Maintain cookies (name value pairs, explained later) deposited on client computers by a web application and send all cookies back to a web site if they are deposited by web applications at that web site (cookies will be further discussed later in this chapter);
- Use plug-in applications to support extra functions like playing audio-video files and running Java applets;

- Implement a *web browser sandbox* security policy: any software component (applets, JavaScripts, ActiveX, …) running inside a web browser normally cannot access local clients' resources like files or keyboards, and can only communicate directly with applications on the web server from where it is downloaded.

The web server is mainly for receiving document requests and data submission from web browsers through the HTTP protocol on top of the Internet's TCP/IP layer. The main function of the web server is to feed HTML files to the web browsers. If the client is requesting a static existing file, it will be retrieved on a server hard disk and sent back to the web browser right away. If the client needs customized HTML pages like the client's bank statement, a software component, like a JSP page or a servlet class (the "Extension" box in the web architecture figure), needs to retrieve the client's data from the database and compose a response HTML file on-the-fly.

The application server is responsible for computing the business logics of the web application, like carrying out a bank account fund transfer and computing the shortest route to drive from one city to another. If the business logic is simple or the web application is only used by a small group of clients, the application server is usually missing and business logics are computed in the web server extensions (PHP, JSP, servlet, and ASP). But for a popular web application that generates significant computation load for serving each client, the application server will take advantage of a separate hardware server machine to run business logics more efficiently. This is a good application of the divide-and-conquer problem-solving methodology.

**Question 1:** What are the four tiers of the web architecture?

**Question 2:** List programs for each of the four web tiers that our *ubuntu* VM has installed.

**Question 3:** What is the difference between a web server and an application server?

**Question 4:** What is the main function of HTML?

**Question 5:** What is the main function of HTTP?

**Question 6:** Why many small companies only use web servers and don't use application servers?

**Question 7:** Is HTTP a transportation layer protocol or an application layer protocol?

**Question 8:** Does JavaScript run in web browsers or on web servers?

**Question 9:** Can JavaScript access the web browser user's file system?

**Question 10:** Can JavaScript communicate with the web site where it is downloaded?

**Question 11:** Can JavaScript communicate with a web site if the JavaScript is not downloaded from the web site?

**Question 12:** What is the web browser sandbox?

**Question 13:** Web browser or web server, who generates cookies?

**Question 14:** When and how are cookies sent to a web server?

**Question 15:** Can a cookie downloaded from web site A be sent to web site B by a web browser?

### *6.1.2 Uniform Resource Locators (URL)*

A web server program runs multiple web applications (sites) hosted in different folders under the web server program's document root folder. A server computer may run multiple server programs including web servers. Each server program on a server computer uses a port number, between 0 and 65535 and unique on the server machine, as its local identification (by default a web server uses port 80). Each server computer has an IP address, like 198.105.44.27, as its unique identifier on the Internet. Domain names, like www.pace.edu, are used as user-friendly identifications of server computers, and they are mapped to IP addresses by a Domain Name Server (DNS). A *Uniform Resource Locator* (URL) is an address for uniquely identifying a web resource (like a web page or a Java object) on the Internet, and it has the following general format:

http://domain-name:port/application/resource?query-string

where *http* is the protocol for accessing the resource (*https* and *ftp* are popular alternative protocols standing for *secure HTTP* and *File Transfer Protocol*); *application* is a server-side folder containing all resources related to a web application; *resource* could be the name (alias or nickname) of an HTML or script/program file residing on a server hard disk; and the optional query string passes user data to the web server. An example URL is http://www.amazon.com/computer/sale?model=dell610.

There is a special domain name "`localhost`" that is normally defined as an alias of local IP address `127.0.0.1`. Domain name "`localhost`" and IP address `127.0.0.1` are for addressing a local computer, very useful for testing web applications when the web browser and the web server are running on the same computer.

Most computers are on the Internet as well as on a local area network (LAN), like home wireless network, and they have an external IP address and a local IP address. To find out what is your computer's external IP address on the Internet, use a web browser to visit http://whatismyip.com. To find out what is your local (home) IP address, on Windows, run "ipconfig" in a DOS window; and on Linux, run "sudo ifconfig" in a terminal window.

**Question 16:** What is the general structure of an URL?

**Question 17:** Why do we need port numbers for networking?

**Question 18:** What is the default port number of a web server?

**Question 19:** How is a domain name mapped to an IP address?

**Question 20:** Could a server computer have multiple IP addresses?

**Question 21:** What is the function of domain name *localhost*?

**Question 22:** Domain name *localhost* is mapped to which IP address?

**Question 23:** How to find your computer's IP address on the Internet?

**Question 24:** How to find your Linux computer's IP address in your home wireless network?

**Question 25:** How to find your Windows computer's IP address in your home wireless network?

### *6.1.3 HTML Basics*

HTML is a markup language. An HTML document is basically a text document marked up with HTML tags to specify document's logical structure and presentation. The following is the contents of file "~/tomcat/webapps/demo/echoPost.html" in the *Ubuntu* VM.

```
<html>
<head>
<body>
  <form method="post" action="http://localhost:8080/demo/echo">
    Enter your name: <input type="text" name="user"/> <br/><br/>
    <input  type="submit"  value="Submit"/>
    <input  type="reset"   value="Reset"/>
  </form>
</body>
</html>
```

An HTML *tag name* is a predefined keyword, like `html`, `body`, `head`, `title`, `p`, and `b`, all in lower-case. A tag name is used in the form of a *start tag* or an *end tag*. A start tag is a tag name enclosed in angle brackets < and >, like `<html>` and `<p>`. An end tag is the same as the corresponding start tag except it has a forward slash / immediately before the tag name, like `</html>` and `</p>`.

An *element* consists of a start tag and a matching end tag based on the same tag name, with optional text or other elements, called *element value*, in between them. The following are some element examples:

```
<p>This is free text</p>
```

```
<p>This element has a nested <b>element</b></p>
```

While the elements can be nested, they cannot be partially nested: the end tag of an element must come after the end tags of all of its nested elements (*first starting last ending*). The following example is not a valid element because it violates the above rule:

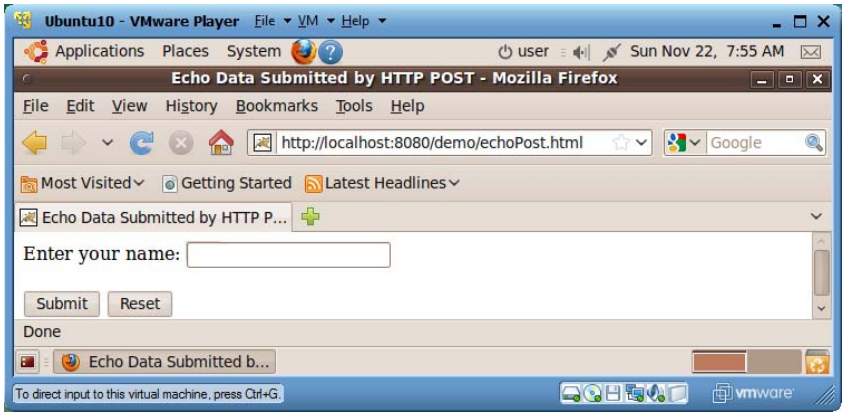
```
<p>This is not a valid <bold>element<p><bold>
```

The *newline* character, the *tab* character and the *space* character are collectively called the *white-space characters*. A sequence of white-space characters acts like a single space for web browser's data presentation. Therefore, in normal situations, HTML document's formatting is not important (it will not change its presentation in web browsers) as long as you don't remove all white-space characters between successive words.

If an element contains no value, the start tag and the end tag can be combined into a single one as `<tagName/>`. As an example, we use `<br/>` to insert a line break in HTML documents.

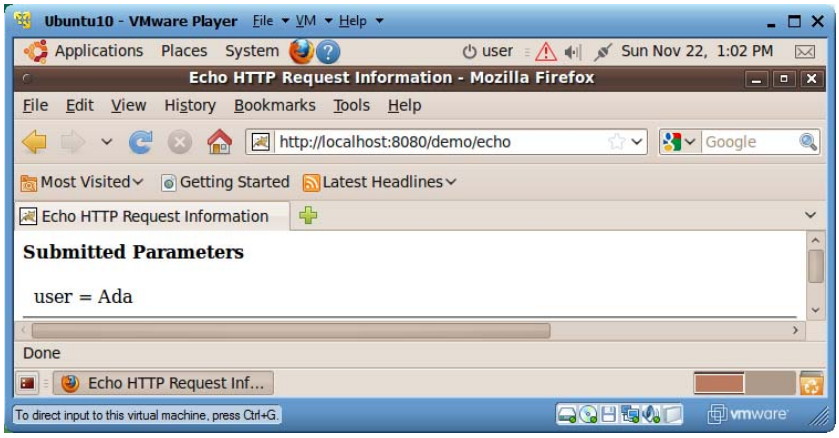
The start tag of an element may contain one or more *attributes*, each in the form "`attributeName="attributeValue"`". The above form element has two attributes: `method` and `action`.

An HTML document must contain exactly one top-level `html` element, which in turn contains exactly one `body` element. Most of the other contents are nested in the `body` element. If you load the above file "echoPost.html" in a web browser you will see the following:

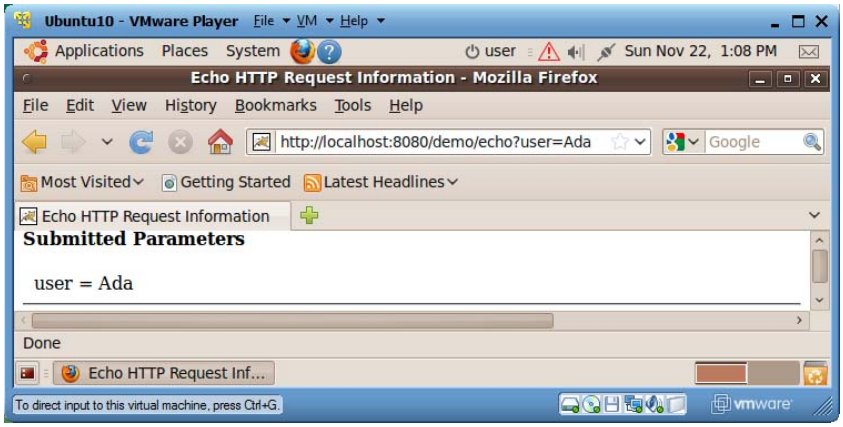



The `form` element is the most important mechanism for interaction between people and web applications. A form typically contains a few `input` elements and at least one submit button. A `form` element usually has two attributes: the `method` attribute for specifying HTTP method for submitting the form data to the web application (only values normally used are "get" and "post"); and the `action` attribute for specifying the form data submission destination, which is the URL of a web application. In this example, when people click the *submit* button, the form data will be sent to resource "echo" of the same web application "demo" deployed on your Ubuntu VM's *Tomcat* web server, which will echo back all information the web browser sent to the web server. If the `action` value doesn't specify the domain name/IP address of the web application, then the web application from where this HTML file came from will receive the form data. If the *action* attribute is missing, the form data will be submitted to the same web component that generated the current web form. The first `input` element of type "text" has been rendered as a text field, the second `input` element of type "submit" has been rendered as a submit button, and the third `input` element of type "reset" has been rendered as a reset button. The `value` attribute of the `input` elements determines what string will be displayed on the element's image. The `name` attribute of the `input` element specifies the variable name with which web server programs can access what people type/enter in the element. When the submit button is clicked, the form data will be packaged as an HTTP request and sent to the web resource specified by the `action` attribute with the method specified by the `method` attribute.

If you type "Ada" in the name field and click the submit button, you will receive the HTTP response partially displayed below.




If you load file "echoGet.html" from the same web application folder "demo", the HTML file contents is basically the same except the method attribute for the form is changed from "post" to "get". If you enter "Ada" in the name field and click the submit button again, you will notice that the query string "`?user=Ada`" has been appended to the end of the URL. This is a major difference from HTTP POST method, and you will learn more about HTTP GET/POST soon.




An HTML file can contain hyperlinks to other web pages so users can click them to visit different web pages. A hyperlink has the general structure of `<a href="url">Hyperlink Text</a>`. The following is an example hyperlink. Since its `href` value is not a web page, the *welcome page* of the Google web site, which is the default page sent back if a browser visits the web site without specifying a specific page, will be sent back to the web browser.

```
<a href="http://www.google.com">Google</a>
```

When you click a hyperlink, an HTTP GET request will be sent to the web server with all values to be submitted in the form of query strings.

**Question 26:** What is the difference between HTML tag name and element?

**Question 27:** HTML attributes are specified in start tag or end tag?

**Question 28:** What are the two main HTML mechanisms for supporting interactions between a web browser and a web server?

**Question 29:** Can a web server initiate a communication with a web browser?

**Question 30:** What are the two main attributes of HTML's form element and what are their functions?

**Question 31:** How to create a text field for users to enter a value?

**Question 32:** How to create a submit button for users to submit the values in an HTML form?

**Question 33:** What is the meaning of the *name* attribute of HTML input elements?

**Question 34:** How to create an HTML hyperlink?

**Question 35:** When you click a hyperlink, do you generate an HTTP POST or HTTP GET request?

### *6.1.4 HTTP Protocol*

Web browsers interact with web servers with a simple application-level protocol called HTTP (HyperText Transfer Protocol), which runs on top of TCP/IP network connections. When people click on the submit button of an HTML form or a hyperlink in a web browser, a TCP/IP virtual communication channel is created from the browser to the web server specified in the URL; an HTTP GET or POST request is sent through this channel to the destination web application, which retrieves data submitted by the browser user and composes an HTML file; the HTML file is sent back to the web browser as part of an HTTP response through the same TCP/IP channel; and then the TCP/IP channel is shut down.

The following is the HTTP POST request sent when you type "Ada" in the text field and click the submit button of the previous file "echoPost.html".

```
POST   /demo/echo   HTTP/1.1
Accept:  text/html
Accept:  audio/x
User-agent:  Mozilla/5.0
Referer:  http://localhost:8080/demo/echoPost.html
Content-length:  8

user=Ada
```

The first line, the request line, of a HTTP request is used to specify the submission type, GET or POST; the specific web resource on the web server for receiving and processing the submitted data; and the latest HTTP version that the web browser supports. As of 2011, version 1.1 is the latest HTTP specification. The following lines, up to before the blank line, are HTTP *header lines* for declaring web browser capabilities and extra information for this submission, each of form "name: value". The first two `Accept` headers declare that the web browser can process HTML files and any standard audio file formats from the web server. The `User-agent` header declares the software architecture of the web browser. The `Referer` (yes this misspelled word is used by the HTTP standard) header specifies the URL of a web page from which this HTTP request is generated (this is how online companies like Amazon and Yahoo collect money for advertisements on their web pages from their sponsors). Any text after the blank line below the header lines is called the *entity body* of the HTTP request, which contains user data submitted through HTTP POST. The `Content-length` header specifies the exact number of bytes that the entity body contains. If the data is submitted through HTTP GET, the entity body will be empty and the data go to the query string of the submitting URL, as you saw earlier.

In response to this HTTP POST request, the web server will forward the submitted data to resource `echo` of web application `demo`, and the resource `echo` (a Java servlet) will dynamically generate an HTML page for most data it can get from the submission and let the web server send the HTML page back to the web browser as the entity body of the following HTTP response.

```
HTTP/1.1   200   OK
Server: NCSA/1.3
Mime_version: 1.0
Content_type: text/html
Content_length: 2000

<HTML>
……
</HTML>
```

The first line, the response line, of an HTTP response specifies the latest HTTP version that the web server supports. The first line also provides a web server processing status code, the popular values of which include 200 for OK, 400 if the server doesn't understand the request, 404 if the server cannot find the requested page, and 500 for server internal error. The third entry on the first line is a brief message explaining the status code. The first two header lines declare the web server capabilities and meta-data for the returned data. In this example, the web server is based on a software architecture named "NCSA/1.3", and it supports *Multipurpose Internet Mail Extension* (MIME) specification v1.0 for web browsers to submit text or binary data with multi-parts. The last two header lines declare that the entity body contains HTML data with exactly 2000 bytes. The web browser will parse this HTTP response and present the response data.

The HTTP protocol doesn't have memory: the successive HTTP requests from the same web browser don't share data.

HTTP GET was initially designed for downloading static web pages from web servers, and it mainly used short query strings to specify the web page search criteria. HTTP POST was initially designed for submitting data to web servers, so it used the request entity body to send data to the web servers as a data stream, and its response normally depended on the submitted data and the submission status. While both HTTP GET and HTTP POST can send user requests to web servers and retrieve HTML pages from web servers for a web browser to present, they have the following subtle but important differences:

- HTTP GET sends data as query strings so people can read the submitted data over submitter's shoulders.
- Web servers have non-standard limited buffer size, typically 512 bytes, for accommodating query string data. If a user submits more data than that limit, either the data would be truncated, or the web server would crash, or the submitted data could potentially overwrite some computer code on the server and the server was led to run some hideous code hidden as part of the query string data. The last case is the so-called *buffer overflow*, a common way for hackers to take over the control of a server and spread virus or worms.
- By default web browsers keep (cache) a copy of the web page returned by an HTTP GET request so the future requests to the same URL can avoid downloading the web page and the cached copy could be easily reused. While this can definitely improve the performance if the requested web page doesn't change, it could be disastrous if the web page or data change with time.

**Question 36:** When you click a *submit* button of an HTML form, you find your form data appears in the web browser's URL address field. Is your form using method Get or POST?

**Question 37:** You use an HTML form to check the value of a specific stock. You found the stock price not changing for extended period of time even though you saw on TV that the stock's price had changed. What could be the problem?

**Question 38:** What are the main differences between HTTP GET and HTTP POST?

**Question 39:** HTTP GET and HTTP POST, which is more secure?

**Question 40:** Which HTTP request method can be used to launch buffer overflow attack to a web server?

**Question 41:** What is an HTTP request's *entity body* for?

**Question 42:** What are an HTTP request's header lines for?

**Question 43:** How can a web browser or a web server know its communication partners capabilities regarding HTTP version and data type support?

**Question 44:** What is the function of HTTP response header line for "referee"?

### *6.1.5 Session Data Management*

Most web applications need a user to interact with it multiple times to complete a business transaction. For example, when you shop at *Amazon*, you choose one book at a time by clicking on some HTML form's submission buttons/hyperlinks in a web browser, and *Amazon* will process your submitted data and send you another HTML form for further shopping. A sequence of related HTTP requests between a web browser and a web application for accomplishing a single business transaction is called a *session*. All data specified by the user is called the *session data*. Session data are private so they must be protected from other users. A session normally starts when you first visit a web site in a particular day, and terminates when you pay off your purchase or shut down your web browser. Since the HTTP protocol has no memory, web applications have to use some special mechanisms to securely maintain the user session data.

#### 6.1.5.1 Cookies

A *cookie* is a pair of name and value, as in (name, value). A web application can generate multiple cookies, set their life spans in terms of how many milliseconds each of them should be alive, and send them back to a web browser as part of an HTTP response. If cookies are allowed, a web browser will save all cookies on its hosting computer, along with their originating URLs and life spans. When an HTTP request is sent from a web browser of the same type on the same computer to a web site, all live cookies originated from that web site will be sent to the web site as part of the HTTP request. Therefore session data can be stored in cookies. This is the simplest approach to maintain session data. Since the web server doesn't need to commit any resources for the session data, this is the most scalable approach to support session data of large number of users. But it is not secure or efficient for cookies to go between a web browser and a web site for every HTTP request, and hackers could eavesdrop for the session data along the Internet path.

#### 6.1.5.2 Hidden Fields

Some web users have great concern of the cookie's security implications and they disable cookie support on their web browsers. A web application can check the header fields of HTTP requests to detect whether cookies are supported by the requesting web browser. If the cookies are disabled, the web application will normally use form hidden fields to store session data. Upon receiving submitted data through an HTTP request, the web application will generate a new HTML form for the user to continue the business transaction, and it will populate all useful session data in the new HTML form as hidden fields (`input` elements of type "hidden"). When the user submits the form again, all the data that the user just entered the form, as well as all data saved in the form of hidden fields, will be sent back to the web application again. Therefore this hidden fields approach for maintaining session data shares most of the advantages and disadvantages of the cookie approach.

#### 6.1.5.3 Query Strings

Sometimes query strings can also be used to maintain small amount of session data. This is particular true for maintaining the short session IDs that will be introduced below. But since most business transactions are implemented with HTML forms, this approach is less useful.

#### 6.1.5.4 Server-Side Session Objects

For improving the security of session data and avoiding the wasted network bandwidth for session data to move back and forth between a web browser and a web server, you can also save much of the session data on the web server as server-side *session objects*. A session object has a unique session ID for identifying a specific user. A session object is normally implemented as a hash table (lookup table) consisting of (name, value) pairs. A single cookie, hidden field of a form, or query string of a hyperlink can be used to maintain the session ID. Since session ID is a fixed size small piece of data, it will not cause much network overhead for going between a web browser and a web server for each HTTP request. For securing the session data, you need to make sure that the session ID is unique and properly protected on the client site. Since this approach stores all session data on the web server, it takes the most server resources and is relatively harder to serve large number of clients concurrently.

**Question 45:** What is the meaning of a web session?

**Question 46:** What is web session data?

**Question 47:** When you shop at *Amazon*, how does *Amazon* maintain your session data (products that you have added to your virtual shopping cart)?

**Question 48:** What are the main mechanisms for supporting session data management?

**Question 49:** What is web session ID and why its security is important?

**Question 50:** Which session data management mechanism is relatively more secure?

**Question 51:** Which session management mechanism is relatively more scalable (supporting huge number of clients' session data without committing proportional amount of resources on the web server)?

**Question 52:** The life span of a cookie is determined by a web server, a web application, or a web browser?

**Question 53:** What are the pros and cons for the long life span of a cookie?

## 6.2 Lab Objectives

In this lab you will

1. Compare HTTP GET and HTTP POST requests;
2. Observe HTTP communications with proxy server *Paros*;
3. Experiment with cookies through web applications;
4. Compare web browser and web server interactions with HTML forms and with hyperlinks;
5. Learn how to use JavaScript to validate form data in the web browsers;
6. Learn how to create a static web site;
7. Learn how to create your first JSP web application on *Tomcat*;
8. Learn how to create your first servlet web application on *Tomcat*.

## 6.3 Lab Guide

### *6.3.1 Comparing HTTP GET and HTTP POST Requests*

1. If VM *Ubuntu* is not on, launch it with username "user" and password 123456.
2. If *Tomcat* is not running, run "tomcat-start" to launch it.
3. Start web browser and visit "http://localhost:8080/demo/echoPost.html".
4. Use the *gedit* (menu item "Applications|Accessories|gedit Text Editor") text editor to open and review the contents of file "~/tomcat/webapps/demo/echoPost.html".
5. Enter your name in the name text field, and click the *submit* button. Observe that the URL doesn't include your name. Read the returned web page for information that was submitted from your web browser to the *Tomcat* web server.
6. Redo steps 2-4 but visit http://localhost:8080/demo/echoGet.html ("~/tomcat/webapps/demo/echoGet.html").

**Question 54:** List all differences between HTTP Get and HTTP POST requests that you observe through this exercise.

**Question 55:** If you use HTTP GET to submit form data that contains a space, such as "John Jay", how the space is represented in query strings?

**Question 56:** If you change the `action` value of the form in "demo/echoGet.html" or "demo/echoPost.html" from "http://localhost:8080/demo/echo" to "/demo/echo", do you see any differences in the behavior of form data submission?

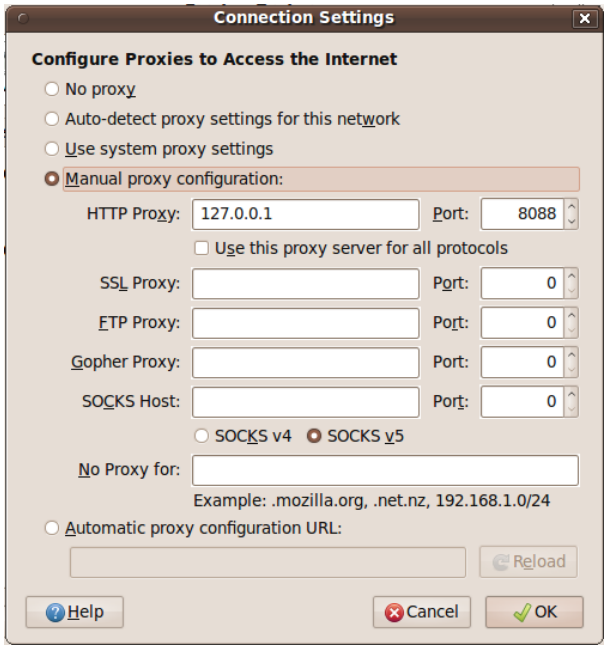
**Question 57:** If you change the `action` value of the form in "demo/echoGet.html" or "demo/echoPost.html" from "http://localhost:8080/demo/echo" to "/echo", do you see any differences in the behavior of form data submission?

**Question 58:** If you change the `action` value of the form in "demo/echoGet.html" or "demo/echoPost.html" from "http://localhost:8080/demo/echo" to "echo", do you see any differences in the behavior of form data submission?

### *6.3.2 Observing HTTP Communications with Paros*

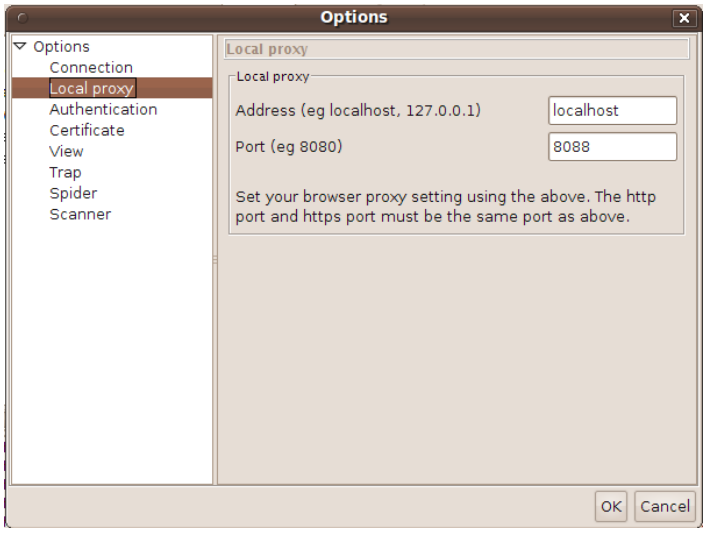
In this exercise, you will use the *Paros* proxy server on your *Ubuntu* VM to observer HTTP communications. The *Paros* proxy server will run at port 8088. You will set up the *Firefox* web browser so that when you use the browser to visit any web site, the HTTP request will be first forwarded to the *Paros* proxy server running at port 8088, which displays the HTTP request information in its graphic user interface, lets the user to have a chance to review and modify the request, sends the request to its destination server, and forwards the HTTP response from the web server back to the *Firefox* web browser. In this exercise you mainly use *Paros* to intercept HTTP GET/POST requests.

1. First you need to edit your browser's proxy settings. Assume you are using *Firefox* V3.6.10 (VM ubuntu). Launch your *Firefox* web browser, and follow its menu item path "Edit|Preferences|Advanced|Network Tab|Settings button" to reach the "Connection Settings" window. Check the "Manual proxy configuration" checkbox. In the HTTP Proxy" text field, enter **127.0.0.1**. In its "Port" text field, enter **8088**. Delete values in the "No proxy for" text field. Now your "Connection Settings" window looks like the following one

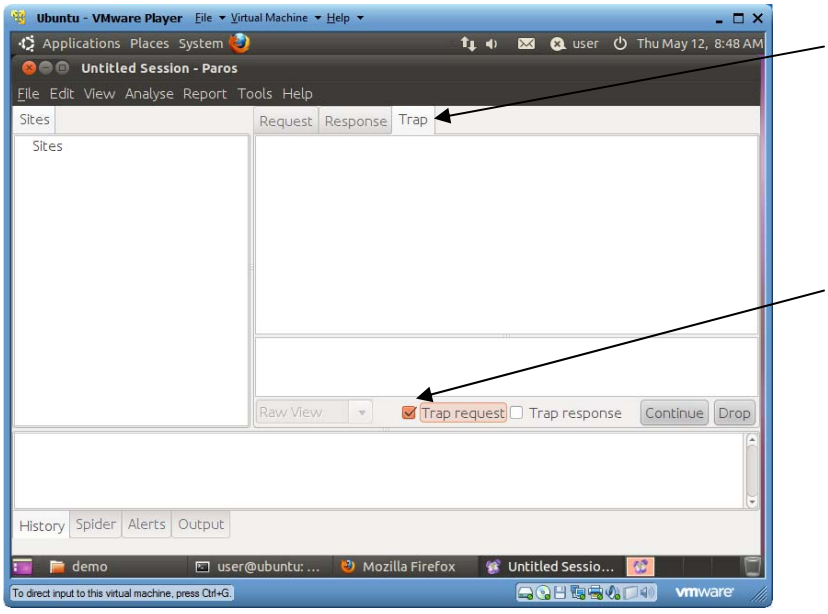



2. Click the *OK* button to close the "Connection Settings" window. Click the *Close* button to close the "Firefox Preferences" window.
3. Use menu item "Applications|Accessories|Terminal" to start a terminal window. Run command "cd ~/paros" to change folder to "~/paros". Run command "java -jar paros.jar" (or "sh

startserver.sh"), accept the license agreement, and *Paros* will start to run with a graphic user interface. In the terminal window you should see error messages because by default *Paros* uses port 8080 which is already in use by the *Tomcat* web server. In the GUI of Paros, click menu item "Tools|Options…" to pop up the "Options" window, select "Local proxy" in the left pane, and enter 8088 in the right *Port* text field, as shown below.




4. Now click on the *OK* button to shut down the *Options* window. Shut down *Paros*, and then restart it. Now *Paros* remembers its new port number and runs at port 8088. Please don't shut down the terminal window that you used to launch *Paros*. Otherwise *Paros* will shut down too.
5. Click the *Trap* pane, and then check "Trap request".




6. Test the *Paros* proxy server by visiting http://localhost with your *Firefox* web browser. Once the browser contacts your web server, *Paros* starts to display HTTP request information in its *Trap* pane. You can modify the HTTP request text. If you click the *Continue* button, the request will be

sent to the HTTP request target, and an HTTP response will be returned and displayed in *Paros*' *Response* pane.

7. You have just enabled all HTTP traffic generated by *Firefox* to be sent to the running *Paros* proxy server which can analyze HTTP traffic before it is sent off to its final destination.
8. Click *Sites* to reveal "http://localhost", and select it.
9. Click a HTTP request for http://localhost in the button pane. You can read its request contents in the *Request* pane. You can read its HTTP response in the *Response* pane.
10. If you uncheck "Trap request" in the *Trap* pane, HTTP requests will not be trapped and you will see HTTP responses right away.
11. Now redo exercise in 6.3.1 with the configured web browser, and use *Paros* to observe the HTTP communications.

**Important Note**: After finishing this exercise, you should reset *Firefox* proxy setting so it stops using the proxy server. Otherwise you would not be able to visit web sites without running the proxy server at port 8088.

### *6.3.3 Working with Cookies*

1. If VM *Ubuntu* is not on, launch it with username "user" and password 123456.
2. If *Tomcat* is not running, run "tomcat-start" to launch it.
3. Start web browser and visit "http://localhost:8080/demo".
4. Use a *gedit* (menu item "Applications|Accessories|gedit Text Editor") text editor to open and review the contents of file "~/tomcat/webapps/demo/index.html".
5. In the web browser, enter "name1" and "value1" in the *name* text field and *value* text field, and click the *submit* button. Read the returned web page for information that was submitted from your web browser to the *Tomcat* web server. You will see that the web server has not received cookies yet (If you have run this demo application before, your will see a session ID cookie).
6. Use the browser *back* button to go back to the web page for "http://localhost:8080/demo". Within 50 seconds. Enter "name2" and "value2" in the name text field and value text field, and click the *submit* button. Read the returned web page for information that was submitted from your web browser to the *Tomcat* web server. You will see that the web browser has received cookie (name1, value1).
7. Use the browser *back* button to go back to the web page for "http://localhost:8080/demo". Wait for one minute or longer. Enter "name3" and "value3" in the *name* text field and *value* text field, and click the *submit* button. Read the returned web page for information that was submitted from your web browser to the *Tomcat* web server. You will see that the web server has received no cookies because the two cookies that you created have expired.
8. Use web browser to visit "http://localhost:8080/testCookie".
9. Use the *gedit* text editor to open and review the contents of file "~/tomcat/webapps/testCookie/index.html".
10. Enter your name in the *name* text field, and click the *submit* button.
11. Within 10 seconds you click on the "Say Hello" button, and you will see a greeting to you.
12. Use the browser *back* button to go back to the previous web page. Wait for 15 seconds or longer for your name cookie to expire. Then click on the "Say Hello" button, and you will not be able to see the greeting to you.

**Question 59:** Can the web developer determine how long a cookie should be alive on the end users' computer?

**Question 60:** Is it a good idea to let your web applications' cookies live forever on end users' computers?

**Question 61:** Can you figure out how to delete all cookies live in your web browser?

**Question 62:** Can you figure out how to disable cookies for your web browser?

**Question 63:** Can you figure out where are your cookies saved by your web browser?

**Question 64:** If your computer has multiple types of web browsers installed, do they share cookies?

### *6.3.4 Submitting Data with HTML Form and Hyperlink*

1. If VM *Ubuntu* is not on, launch it with username "user" and password 123456.
2. If *Tomcat* is not running, run "tomcat-start" to launch it.
3. Start web browser and visit "http://localhost:8080/tripler".
4. Use a gedit (menu item "Applications|Accessories|gedit Text Editor") text editor to open and review the contents of file "~/tomcat/webapps/tripler/index.html".
5. Enter "1" in the first text field, and click the "Triple this integer" button to its left. Notice the query string "?number=1" appended to the end of the URL because an HTTP GET request was used.
6. Use the browser *back* button to go back to the web page for "http://localhost:8080/tripler". Enter "2" in the second text field, and click the "Triple this integer" button to its left. Notice that no query string is used in the URL because it is an HTTP POST request.
7. Use the browser back button to go back to the web page for "http://localhost:8080/tripler". Click the "Triple 4" hyperlink in the last line of the web page. Notice the query string "?number=4" appended to the end of the URL because HTTP GET request is always used when a hyperlink is clicked.

**Question 65:** What type of HTTP request will be used when a user clicks a hyperlink?

### *6.3.5 Validating Form Data with JavaScript*

JavaScript is a simple scripting language that runs inside web browser security sandbox. It can be used to manipulate HTML file and validate form data before they are submitted to web applications. In this exercise you will learn how to use JavaScript to validate form data.

1. If VM *Ubuntu* is not on, launch it with username "user" and password 123456.
2. If *Tomcat* is not running, run "tomcat-start" to launch it.
3. In a file explorer, open "/home/user/tomcat/webapps". Right-click any blank space in the explorer right pane and choose "Create Folder" to create new folder "test".
4. In the file explorer, open folder "~/tomcat/webapps/test". Right-click any blank space in the explorer and choose "Create Document|Empty File" and create a new file with name "index.html".
5. Right-click file "index.html" and choose menu item "Open With Text Editor" to open file "index.html" in the *gedit* editor.
6. Type the following text into the file, and save the file. You can leave the editor open for later file modification.

```
<html>
<head><title>Echo Submitted Data</title></head>
<body>
  <form  method="get"  action="/demo/echo">
    Enter your name: <input   type="text"  name="user"/><br/><br/>
    <input  type="submit"  value="Submit"/>
    <input  type="reset"  value="Reset"/>
  </form>
</body>
</html>
```

7. Start web browser and visit “http://localhost:8080/test”.
8. Type your name in the name text field and click the *submit* button, and you will see your form data echoed back to you from the web server.
9. Use the browser’s left arrow to go back to the web page of “http://localhost:8080/test”.
10. Make sure the *name* text field is empty, and click the *submit* button. You will see that the incomplete form data is still submitted to the web server and no warning was given.
11. Now you are ready to add JavaScript code in the HTML file to make sure that the form would not submit unless the user has typed something in the *name* field. In the *gedit* editor, modify the file contents so it reads as below. Save the file. The JavaScript code is declared inside the *script* element which is nested in the *head* element. You don’t need to pay attention to the *trim()* function which is used to remove the leading and trailing white space characters of a string (Java has this method, but JavaScript doesn’t have it). First notice the newly added 3rd attribute of the *form* element: onsubmit="return dataCheck(this)". With this new attribute, when a submit button is clicked, the form (*this*) is first validated by JavaScript function *dataCheck()*. If the function returns true, the form data is valid and then actually submitted to the *action* target as an HTTP request. If the function returns false, the validation fails, the form is not submitted, and the user has the chance to revise the form data. In the body of function *dataCheck(form)*, parameter *form* represents the HTML form, and the data that the user has typed in the name text field is in variable “form.user” (“user” is the text field’s name). The text field’s value is first assigned into a new variable *widget*. If the value is not empty, then the validation succeeds and the *dataCheck()* function returns true. If the value is an empty string, then the form has not been completed yet and thus is not valid. In this case the *alert()* function is used to pop up a warning window, the *focus()* method is called to position the cursor in the text field, the *select()* function is called to highlight the text field so the user can be ready to type value in the text field, and then the *dataCheck()* function returns false to declare validation failure.

```
<html>
<head><title>Echo Submitted Data</title>
<script language="Javascript" type="text/javascript">
  String.prototype.trim = function() {
    // remove string's leading spaces, then trailing spaces
    return this.replace(/^\s*/, "").replace(/\s*$/, "");
  }
  function dataCheck(form) {
    widget = form.user;
    if (widget.value.trim() == "") {
      alert("You must enter your name");
      widget.focus();  // position the cursor in the field
      widget.select(); // highlight the field
```

```
        return false;
      }
      return true;
    }
  </script>
  </head>
  <body>
    <form method="post" action="/demo/echo"
          onsubmit="return dataCheck(this)">
      Enter your name: <input   type="text"  name="user"/><br/><br/>
      <input  type="submit"  value="Submit"/>
      <input  type="reset"  value="Reset"/>
    </form>
  </body>
  </html>
```

12. Use the web browser to visit http://localhost:8080/test again. Reload the page. Leave the name text field empty and click the *submit* button. This time you will see the warning and the form is not submitted and you are ready to revise it.

Be warned that validating form data with JavaScript is not secure because people could save a copy of the HTML file on the hard disk, modify the file to remove the validation invocation, and then reload the HTML file. For sensitive form data the web applications must validate them on the web server.

**Question 66:** Which attribute of HTML form is used to implement JavaScript form data validation?

**Question 67:** Why JavaScript form data validation is not secure?

### *6.3.6 Creating Your First JavaServer Page Web Application*

JavaServer Page (JSP) is a Java technology for generating HTML files on-the-fly based on data submitted by web browser users. In this exercise you will create an HTML form that submits data to a JSP page for presentation. A JSP page is basically an HTML file with small pieces of Java code pieces enclosed in <% and %> delimiters to compute dynamic values.

1. If VM *Ubuntu* is not on, launch it with username "user" and password 123456.
2. If *Tomcat* is not running, run "tomcat-start" to launch it.
3. In a file explorer, open "/home/user/tomcat/webapps". Right-click on any blank space in the explorer right pane and choose "Create Folder" to create new folder "welcomeJSP".
4. In the file explorer, open folder "~/tomcat/webapps/welcomeJSP". Right-click any blank space in the explorer and choose "Create Document|Empty File" and create a new file with name "index.html".
5. Right-click file "index.html" and choose menu item "Open With Text Editor" to open file "index.html" in the *gedit* editor.
6. Type the following text into the file, and save the file.

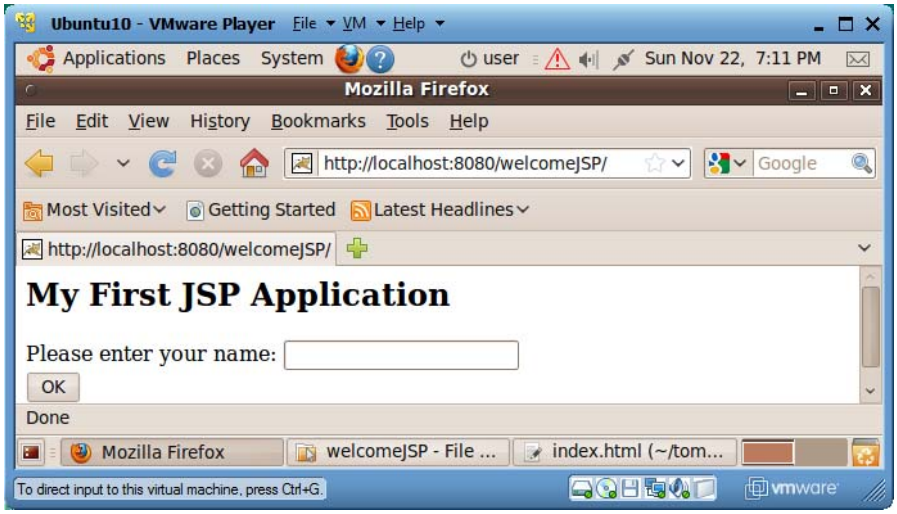
```
<html>
<body>
<h2>My First JSP Application</h2>
<form method="post" action="welcome.jsp">
```

```
Please enter your name: <input type="text" name="name"/>
<br/>
<input type="submit" value="OK"/>
</form>
</body>
</html>
```

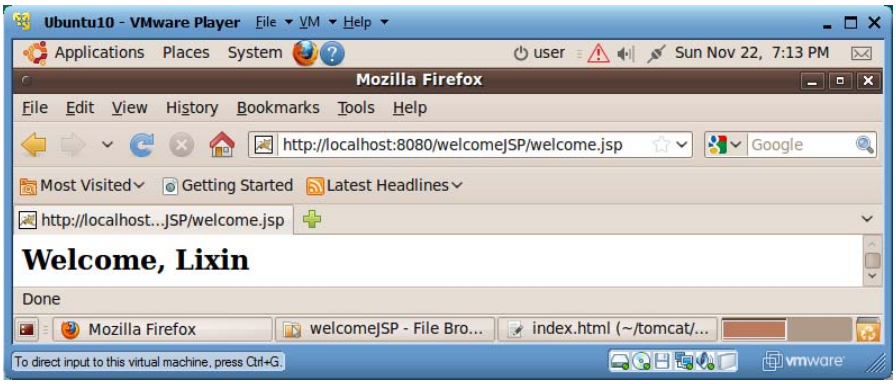
7. Repeat steps 3, 4 and 5 to create a new file “welcome.jsp” with the following contents. JSP *request* object represents all data submitted by a web browser. Method *request.getParameter("name")* retrieves the value that the user has entered the text field with name “name”. JSP expression *<%=name%>* returns the current value of variable *name*.

```
<html>
<body>
<%
  String name = request.getParameter("name");
%>
<h2>Welcome, <%=name%></h2>
</body>
</html>
```

8. Use the web browser to visit http://localhost:8080/welcomeJSP and you will see the following window:




9. Type your name in the text field and click the *OK* button, and you will see a window similar to the following one.




10. Congratulations and you have completed your first JSP web application!
11. Use the file explorer to visit folder "/home/user/tomcat/work/Catalina/localhost/welcomeJSP/org/apache/jsp". You will find two files "welcome_jsp.java" and "welcome_jsp.class". When the JSP page "welcome.jsp" is visited for the first time, the file is converted into a Java servlet source file "welcome_jsp.java" which is then compiled into bytecode file "welcome_jsp.class" for execution. This transformation will only happen for the first visit of the JSP page. The following is part of contents of file "welcome_jsp.java":

```
        out.write("<html>\n");
        out.write("<body>\n");
        String name = request.getParameter("name");
        out.write("\n");
        out.write("<h2>Welcome, ");
        out.print(name);
        out.write("</h2>\n
        out.write("</body>\n");
        out.write("</html>\n");
```

12. Therefore JSP is built on top of Java servlets and JSP just makes Java servlets easier to use by web developers.

**Question 68:** A JSP page is more like an HTML file or a Java source code file?

**Question 69:** In JSP, how to retrieve the value that the web browser user has typed in a text field?

**Question 70:** In JSP how to insert the value of a variable in the HTML file?

**Question 71:** Is JSP a technology independent of the servlet technology?

**Question 72:** When are JSP files converted into Java servlet files?

### *6.3.7 Creating Your First Servlet Web Application*

From the last exercise you know that servlets are the cornerstone of Java web technologies. In this exercise you will develop a servlet web application with the same function as the last *welcomeJSP* web application so you can better compare the two technologies.

1. If VM *Ubuntu* is not on, launch it with username “user” and password 123456.
2. If *Tomcat* is not running, run “tomcat-start” to launch it.
3. In a file explorer, open “/home/user/tomcat/webapps”. Right-click any blank space in the explorer right pane and choose “Create Folder” to create new folder “welcomeServlet”.
4. In the file explorer, open folder “~/tomcat/webapps/welcomeServlet”. Right-click any blank space in the explorer and choose “Create Document|Empty File” and create a new file with name “main.html”.
5. Right-click file “main.html” and choose menu item “Open With Text Editor” to open file “main.html” in the *gedit* editor.
6. Type the following text into the file, and save the file.

```
<html>
<body>
<h2>My First Servlet Application</h2>
<form method="post" action="welcome">
Please enter your name: <input type="text" name="name"/>
<br/>
<input type="submit" value="OK"/>
</form>
</body>
</html>
```

7. Create in folder “~/tomcat/webapps/welcomeServlet” a new folder “WEB-INF”. All servlet based web applications have this folder for holding those files not directly accessible from web browsers.
8. Create in folder “~/tomcat/webapps/welcomeServlet/WEB-INF” a new folder “classes”. All Java classes to be run on the web server must be under this folder.
9. Create in folder “~/tomcat/webapps/welcomeServlet/WEB-INF/classes” a new file “Welcome.java”. Copy the following contents into this file and save the file. If a servlet needs to process HTTP POST requests, it needs to have a *doPost(request, response)* method. If a servlet needs to process HTTP GET requests, it needs to have a *doGET(request, response)* method. We put all logics in the *doPost(request, response)* method and call this method inside the *doGet(request, response)* method to avoid redundant code. Both of the two methods have two parameters: *request* representing all data submitted through the HTTP request, including data in the HTTP request entity body and query string; and *response* representing the data to be sent back to the remote web browser through an HTTP response. Method *request.getParameter("name")* is first called to retrieve the value that the user has typed in the *name* text field. After setting the output data type and retrieving the output object of *response*, the remainder of the code just prints out an HTML file piece by piece. This section of code must remind you of the similar code that you reviewed in file “welcome_jsp.java” produced from file “welcome.jsp” in the last exercise.

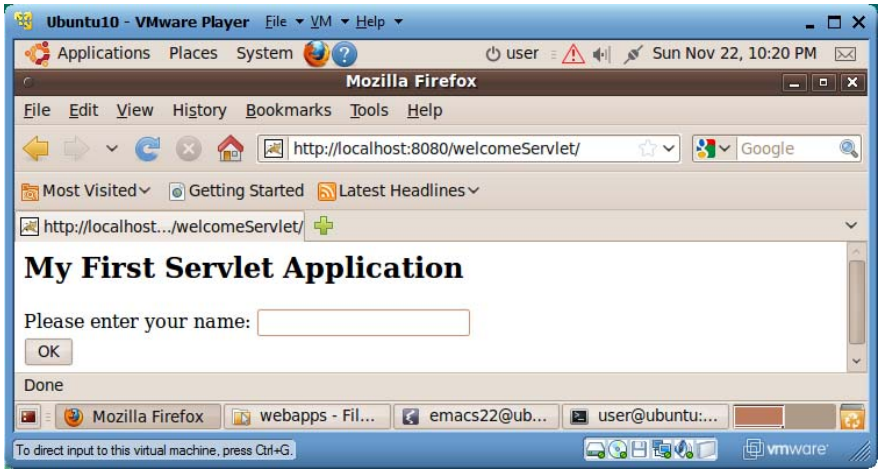
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class Welcome extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws IOException, ServletException {
    String name = request.getParameter("name");
    response.setContentType("text/html");
```

```
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<body>");
      out.println("<h2>Welcome, " + name + "</h2>");
      out.println("</body>");
      out.println("</html>");
    }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
      throws IOException, ServletException {
      doPost(request, response);
    }
  }
```

10. Create in folder "~/tomcat/webapps/welcomeServlet/WEB-INF" a new file "web.xml" and copy the following contents into it. Each servlet web application needs this configuration file. You first declare that file "main.html" is the web application's welcome file: if a web browser visits this web application but not specifying which file to retrieve, the welcome file will be sent back by default. You then assign a name "welcome" to the servlet class "Welcome". In the last "servlet-mapping" element, you declare that if the URL of an HTTP request contains "/welcome", the request will be sent to the servlet named "welcome" for processing, which is "Welcome" in this case.

```
<web-app>
  <servlet>
    <servlet-name>welcome</servlet-name>
    <servlet-class>Welcome</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>welcome</servlet-name>
    <url-pattern>/welcome</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>main.html</welcome-file>
  </welcome-file-list>
</web-app>
```

11. Now you need to compile the servlet into a bytecode file "Welcome.class". Use the file browser to open folder "~/tomcat/webapps/welcomeServlet/Web-INF/classes". Right-click any blank space in the right pane, and choose menu item "Open in Terminal". A new terminal window will start with "~/tomcat/webapps/welcomeServlet/WEB-INF/classes" as its working folder. Run command "javac Welcome.java" to compile Java source file "Welcome.java" into "Welcome.class".
12. You have completed your first servlet web application. Use a web browser to visit http://localhost:8080/welcomeServlet and you will see a web browser view similar to the following one. [Sometimes *Tomcat* could get confused when you are developing web applications. Restart the VM may resolve some of these problems.] Test it.




13. JSP and servlet web applications are normally deployed as Web Archive (WAR) files. To make a WAR file "welcomeServlet.war" for the current web application, start a terminal window in folder "~/tomcat/webapps/welcomeServlet, and run command "jar cvf welcomeServlet.war *". To deploy this web application in a different *Tomcat* web server, you only need to drop file "welcomeServlet.war" in that *Tomcat* installation's "webapps" folder, and this WAR file will be automatically extracted into web application folder "welcomeServlet" and the web application will start to work right away, assuming that *Tomcat* is running.
14. By now you have successfully completed your first servlet web application. Congratulations!

**Question 73:** What kind of files should be put under folder "WEB-INF"?

**Question 74:** Where should Java servlet files be located in a servlet web application?

**Question 75:** Applets are Java classes to be downloaded to web browsers and run in web browser sandboxes. Should applet files be put under folder "WEB-INF"?

**Question 76:** What are the main methods that a Java servlet class should have?

**Question 77:** What are the functions of parameters `request` and `response` of servlet methods `doGet()` and `doPost()`?

**Question 78:** What is the main function of servlet methods `doGet()` and `doPost()`?

**Question 79:** What is the main function of configuration file "WEB-INF/web.xml"?

**Question 80:** What is a welcome file of a web application?

**Question 81:** What is URL pattern of a servlet?

**Question 82:** What is the relationship between a JSP page and a servlet class?

## 6.4 Review Questions

**Question 83:** In your *Ubuntu* VM, visit web application http://localhost:8080/bareJsp and study its source files. Explain the function and design of this web application.

**Question 84:** Develop a JSP web application that displays in a web browser an integer and a submit button. The integer is initially 0. Each time the user clicks the button, the integer increases by 1. [Hint: To convert string "12" to integer 12, you can use Java code

```
int v = 0;
try  {  v = Integer.parseInt("12"); }
catch (Exception  e) { v = 0; }
```

]

**Question 85:** Develop a servlet web application that displays in a web browser an integer and a submit button. The integer is initially 0. Each time the user clicks the button, the integer increases by 1.
[Hint: If you want a servlet to create the default welcome web page, use the servlet's "servlet-name" element value as the value of the "welcome-file" element in file "web.xml". To convert string "12" to integer 12, you can use Java code

```
int v = 0;
try  {  v = Integer.parseInt("12"); }
catch (Exception  e) { v = 0; }
```

]

# 7 Introduction to Web Services

## 7.1 Concepts

Since you have read, hopefully worked out, the previous section "Introduction to Web Technologies", you should now have basic understanding of the four-tier web architecture, HTML form and the HTTP protocol for communications between web servers and web browsers. If you are not sure, please review that module before continue.

This module addresses the following fundamental challenges to today's computing industry: (a) how to let heterogeneous information systems communicate with each other, (b) how to let the consumers get computing services without the need of managing their own hardware/software systems, (c) how to mechanically convert a net-blind legacy application into a net-aware application and deliver services across the Internet. The solution to all of these challenges depends on the technology called *web services*.

### *7.1.1 Challenges that Web Services Address*

#### 7.1.1.1 Integration of Heterogeneous Information Systems

The business processes of a company are normally supported by many types of information systems, including transaction servers, inventory systems, accounting systems, and payroll systems. These systems may be from different vendors and built with different technologies. For smooth and efficient operation of the company, these internal information systems need be integrated to work together.

For many companies, their operating and management units are distributed across multiple buildings, cities, or even across the world. For efficient global decision making and business coordination, the information systems of these distributed units must be integrated. Depending on the size, location, inception time, and function of these units, their information systems may differ in terms of operating platforms and technologies.

In today's business environment, very few companies can be really self-sufficient. Each company needs to consume products or services from other companies, and produce new products or services to serve its own clients. As a result, the Business-to-Business (B2B) computing model is very important, and the information systems of different companies need to be integrated to some extent so that they can conduct business transactions automatically without extensive human interventions. Information systems of different companies are usually based on different platforms and technologies.

The web service technology introduced in this module can solve this enterprise information system integration problem.

**Question 1:** What are the main advantages of hosting computing services on data centers and deliver them to web browsers through the Internet?

**Question 2:** What is the main difference of the web services from the computing services that we discussed in question 1?

### 7.1.1.2 Service-Oriented Architecture (SOA)

*Service-oriented architecture (SOA)* is a style of software architecture for loose-coupling of software agents (programs or information systems of various granularities) in a distributed environment. A *service* could be as simple as a remote method invocation, or it could be a complete business transaction like buying a book from *Amazon* online. The services could also be classified into *horizontal* ones and *vertical* ones, the former providing basic common services to many other services' implementations, and the latter providing services in a particular application domain. The *service provider* and *service consumer* are two roles that could be played by different or the same software agent. The important services should have standardized *application programming interfaces (API)* so that a service consumer can choose from many service implementations conforming to the same standardized API.

The success of SOA depends on a few things. First, a standard interfacing technology must be adopted by all participating software agents, and service consumers should not need to install special tools to use each special service implementation. Second, the service providers should have a standardized mechanism to publicize their services as to their availability, functions, APIs, cost, and quality assurance levels; and the service consumers could use the same mechanism to search for desired service implementations based on the industry category of the services in demand. A service consumer should be able to easily switch from one service implementation to another more cost-effective one. The standardization of the interfacing technology and business service API are critical to the success of SOA.

As an architectural style, SOA involves more than IT technologies. SOA could have impacts on business models or processes too. In this module we focus on the foundation problem of SOA: how to use web services to support communications between heterogeneous computing systems (software agents).

**Question 3:** Why *web service* API standardization is important to the success of the service-oriented architecture methodology?

**Question 4:** Why *service* API standardization is important to the success of the service-oriented architecture methodology?

### 7.1.1.3 Fast Transformation of Legacy Applications for E-Commerce

Many enterprise applications take decades to improve and stabilize. They have been the cornerstones of the business operation. Most of these applications are designed for interacting with the company staff only, not with the end users whom they serve. We say such legacy applications are net-blind since they cannot communicate with the remote end users over the Internet.

With today's e-commerce and the ever-increasing communication models including social networking, the companies have the urge to provide similar services over the Internet to the national or global end users. In this environment the applications need to communicate directly with the end users and deliver many services with minimal staff involvement. It is just too expensive or slow to redesign and re-implement those mature applications, and even small changes to those applications could introduce subtle bugs. The solution is to use web service as a wrapper technology to transform the legacy applications and allow them communicate directly with the remote users without modifying the business logic core of the applications. This is one of the major strengths of web services.

**Question 5:** Why should you try to avoid rewriting programs for core business logics?

### *7.1.2 Fundamental Concepts of Web Services*

Now we use a simple scenario to explain the fundamental ideas of web services. Assume computer A runs a Java object *Business* (business logic implementation) on the Linux, and the object supports a method with signature "String hello(String n)" (the invoker needs to send method "hello" a string, and the method then returns another string); and computer B runs a C++ object *Client* on the Windows; both computers are connected by some computer network including the Internet or a home LAN; now the object *Client* on computer B needs to invoke the method "hello" of object *Business* on computer A over the network. We need to address the following challenges:

1. How for computer A to accept method invocation requests over the network? The solution is to run a web server on computer A, and deploy a Java servlet or ASP .NET web page on the web server as the reception point web object for method invocations.
2. How for computer A to tell computer B the signature of method "hello" in a programming language neutral way (computer B may have no clue about the programming language in which method "hello" on computer A is implemented), and how for computer A to receive method invocation requests over the network? The solution is to define a simple XML dialect called *Web Service Description Language (WSDL)*, which is just expressive enough to specify the method signatures, like "String hello(String n)", and the URL for the reception point web object for method invocation (a Java servlet or ASP .NET web page on computer A's web server). A web service utility on computer A could read the source code of object *Business*, generate a WSDL file for describing method "hello", generate the reception point web object for method "hello", and deploy the reception point web object on the local web server. The WSDL file for method "hello" could be sent to computer B in any possible way including web downloading or email attachment. The following is the simplified WSDL file for method "hello".

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions>
  <message name="helloResponse">
    <part  name="helloReturn"  type="xsd:string" />
  </message>
  <message name="helloRequest">
    <part name="n" type="xsd:string" />
  </message>
  <portType name="HelloServer">
    <operation  name="hello"  parameterOrder="n">
      <input message="helloRequest"  name="helloRequest" />
      <output message="helloResponse"  name="helloResponse" />
    </operation>
  </portType>
  <service name="HelloServerService">
    <port binding="HelloServerSoapBinding" name="HelloServer">
      <address location="http://localhost:8080/axis/HelloServer.jws" />
    </port>
  </service>
</definitions>
```

3. How could computer B send method invocation arguments to computer A? The solution is to define a separate XML dialect *Simple Object Access Protocol (SOAP)* for specifying all information about a method invocation: name of the target method, and value for each of the parameters; or the return value of such a method invocation, in a programming language independent way (in XML). A simplified sample SOAP message may read like

```
<?xml version="1.0"?>
<Envelope>
  <Body xmlns:m="localhost:8080/axis/HelloServer.jws">
    <m:HelloServer>
       <m:helloRequest>John</m:helloRequest>
    </m:HelloServer>
  </Body>
</Envelope>
```

4. How could C++ object *Client* on computer B invoke the remote Java method "hello"? A web service utility, maybe called *wsdl2cpp*, on computer B would download the WSDL file for "hello", read it and generate source code for a new proxy object in a programming language specified by the developer to the utility. In our case a set of C++ source code would be generated to implement the proxy class. The proxy class supports a method "hello" with exactly the same signature (method name and parameter number and data types) as the "hello" method on computer A but in C++ syntax. At run time the *Client* object would create a local proxy object and invoke its "hello" method. The proxy's method "hello" would convert the argument string into the neutral XML format, use this XML format argument and method name "hello" to generate a SOAP message, and initiate an HTTP POST request to the reception point web object on computer A (the URL is part of the WSDL file) with the SOAP message in the HTTP request's entity body. On computer A the reception point web object would pick up the HTTP request entity body, the SOAP message, unwrap it, convert argument value from XML into Java format, invoke the local Java *Business* object method "hello" with the converted argument from computer B, convert the return value into XML, wrap the converted return value in a new SOAP message, and return the SOAP message back to computer B as part of its HTTP response. Upon receiving this new SOAP message from computer A, the proxy object's method "hello" would unwrap the incoming SOAP message, convert the return value from XML format to C++ format, and return the resulting value as its own return value.

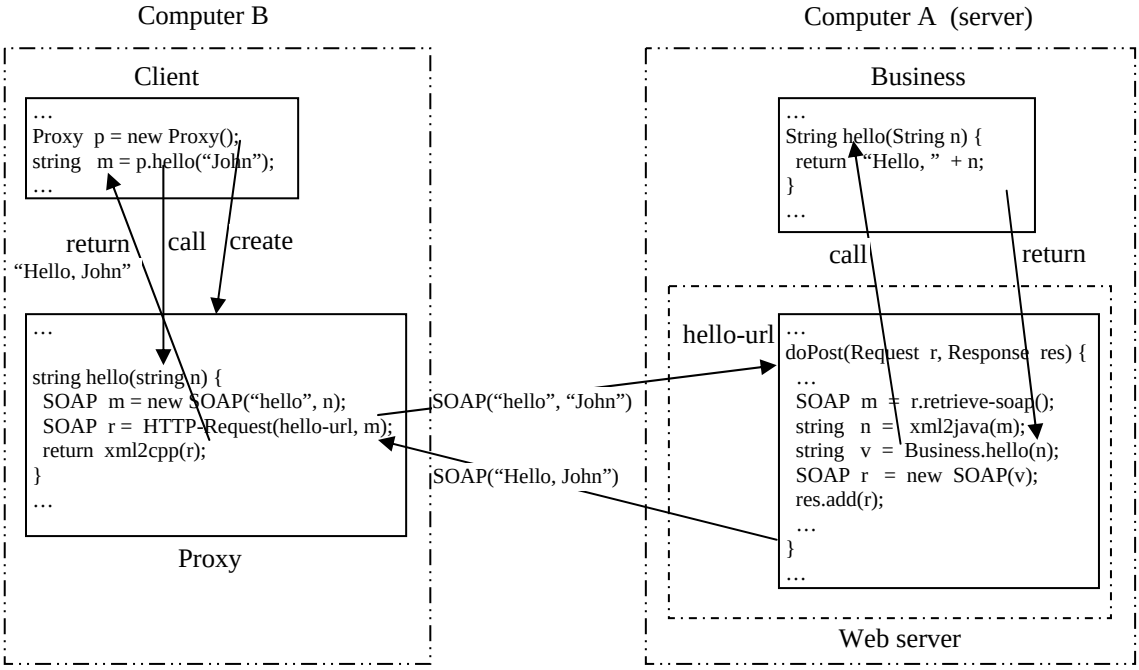



**Figure 1 Web service method invocation**

The above simple scenario explained the basic mechanism of web services. As with all remote procedure or method invocation technologies, web services is based on the popular *proxy design pattern* (http://sourcemaking.com/design_patterns/proxy).

One observation can be made that, since WSDL files and SOAP files are generated and consumed by tool utilities or machine-generated code, it is not mandatory for developers to understand completely the syntaxes of the WSDL and SOAP structures, even though some basic understanding of WSDL and SOAP is helpful. WSDL and SOAP are not intended for human beings to write or read. That is why even though they are important in the implementation of web services, they are not covered beyond logical descriptions in most textbooks on web services.

**Question 6:** What is the function of a WSDL file?

**Question 7:** What is the function of a SOAP message?

**Question 8:** What is the function of XML in web services?

**Question 9:** Do all servers running web services have a web server installed?

**Question 10:** What is the proxy design pattern?

### *7.1.3 Major Components of Web Services*

Web services are based on the following major technologies:

- *Universal Description, Discovery and Integration (UDDI)*, a standard for web service registries to which web service providers can register and advertise their services, and potential clients can use the registries as yellow pages to search for potential service providers and download the WSDL service description files for the chosen services. UDDI allows programs to access the registries through their open APIs.

  At this point there are no commercial UDDI registries available. Microsoft, IBM and SAP used to run their public free UDDI registries to validate the technology, but they have shut down this service in 2006 after they declared that the UDDI validation was a success. The following web pages contain some public web services that you can explore and test.

  http://webservices.seekda.com/
  http://www.xmethods.com/ve2/index.po
  http://www.service-repository.com/

- *Web Service Description Language (WSDL)*, a dialect of XML for specifying in a language-independent way the exposed web service methods' signatures, the data types that they depend on, the exceptions that the methods may throw at execution time, and the reception point on a web server for clients to send SOAP messages through HTTP POST/GET requests representing method invocations. Because WSDL files are generated and consumed by tools and machine-generated code, a web service developer does not need to write them or read them.

- *Simple Object Access Protocol (SOAP)*, a dialect of XML for specifying in language-independent way all information about a method invocation, including the name of the method, the arguments

of the method invocation, return value, and return exceptions. Because SOAP messages are generated and consumed by tools and machine-generated code, a web service developer does not need to write them or read them.

- *Extensible Markup Language (XML)*, the base language for WSDL and SOAP. XML is not needed explicitly in the development of web services.

- *HyperText Transfer Protocol (HTTP)*, a common carrier for sending SOAP messages between web service client systems and service implementations. You need to have a basic understanding of how a web server processes the HTTP POST/GET requests.

### *7.1.4 Web Service's Role in System Integration*

If you are sure that your application and its consuming applications, local or remote, are all based on the Windows platform, then Microsoft .NET remoting (http://msdn.microsoft.com/en-us/library/ms973857.aspx), part of Microsoft COM+, is the best technology for system integration because it is customized for the Windows platform. On the other hand, if you are sure that your application and its consuming applications, local or remote, are all based on the Java platform, then Java Remote Method Invocation over IIOP, RMI-IIOP (http://en.wikipedia.org/wiki/RMI-IIOP), is the best technology for system integration because it is customized for the Java platform. Since 1990 the industry consortium *Object Management Group* (OMG, http://omg.org), including over 200 major companies worldwide, developed the sophisticated distributed object model/technology CORBA and API standards for heterogeneous system integration in various application domains. While CORBA supports more advanced services than web services, it is too aggressive and heavy and thus not successful in business. Since early 2000s the major IT companies, including Microsoft, IBM and Sun (now part of Oracle), acknowledged that no single platform/technology could dominate in all domains, and started to define the common open-specification for a simpler system integration technology based on existing technologies HTTP and XML. Web service's main objectives are simplicity and wide acceptance, not performance. Fundamentally, web services support remote method invocation where the two communicating computers may be separated by the Internet, based on different hardware and software platforms (computer architecture and operating system respectively), and implemented with different programming languages. The direct consumers of web services are programs, not people.

The main strength of web services include

1. It supports method invocation across heterogeneous systems.
2. It is not a proprietary technology and therefore inexpensive to adopt.
3. It requires minimal programming in its adoption.
4. Systems don't need to install special software for consuming different web services. The client system only needs to have the generic web service toolkit and XML library for its specific platform.
5. Applications access the remote web services through port 80 which is reserved for the web and normally not blocked by enterprise firewalls, so web service deployment is very simple.

The major drawbacks of web services include

1. It is a wrapper technology running on top of multiple existing software layers (web, HTTP, XML) therefore it is not as efficient as those technologies customized for specific platforms.
2. Its specification, especially for XML representation of the advanced data structures, is still not completely standardized. As a result in real projects you may find the need to work at lower levels than we describe in this module.

**Question 11:** Are web services a cutting-edge technology?

**Question 12:** Should you always use web services to integrate components of an information system?

**Question 13:** How could you use web services to support fast transformation of a legacy application to provide services over the Internet?

**Question 14:** Why do we say that web services are easy to deploy?

### *7.1.5 Web Service Security*

While many web services are open to the public, there are many proprietary web services that are intended for targeted clients. Also, it is very important to avoid malicious intruders from compromising the business data behind the web services or the availability of the web services.

There are three major approaches to secure a web service.

1. Use *Secure Socket Layer (SSL)* protocol to provide a secure communication channel between the web service and its client systems. As a result, intruders cannot easily eavesdrop data between the web service and its client systems.
2. Encrypt SOAP data by the client-side's proxy classes, and decrypt SOAP data at the reception end web object. As a result, even if the intruders intercepted the SOAP data, they could not easily interpret the data.
3. Implement authentication and authorization mechanisms at OS level or application level.

## 7.2 Lab Objectives

In this lab you will

1. Develop a Java web service for squaring an integer;
2. Develop a Java client application for consuming the local Java web service;
3. Run web service client and server on a LAN;
4. Develop a Java client application to consume a public remote .NET web service.

## 7.3 Lab Guide

### *7.3.1 Development of a Java Web Service for Squaring an Integer*

In this lab, you are going to create a new Java web service that receives one integer and returns its squared value. For example, if input is 2, it will return 4.

The implementation is through a Java class source file. For your convenience, the file has been created in "~/webServiceLab/java/server/SquareIntegerServer.java". Its contents are listed below:

```
public  class  SquareIntegerServer {
   public int square(int x) throws Exception {
      int  result = x*x;
      return  result;
   }
}
```

**Program 1  SquareIntegerServer.java**

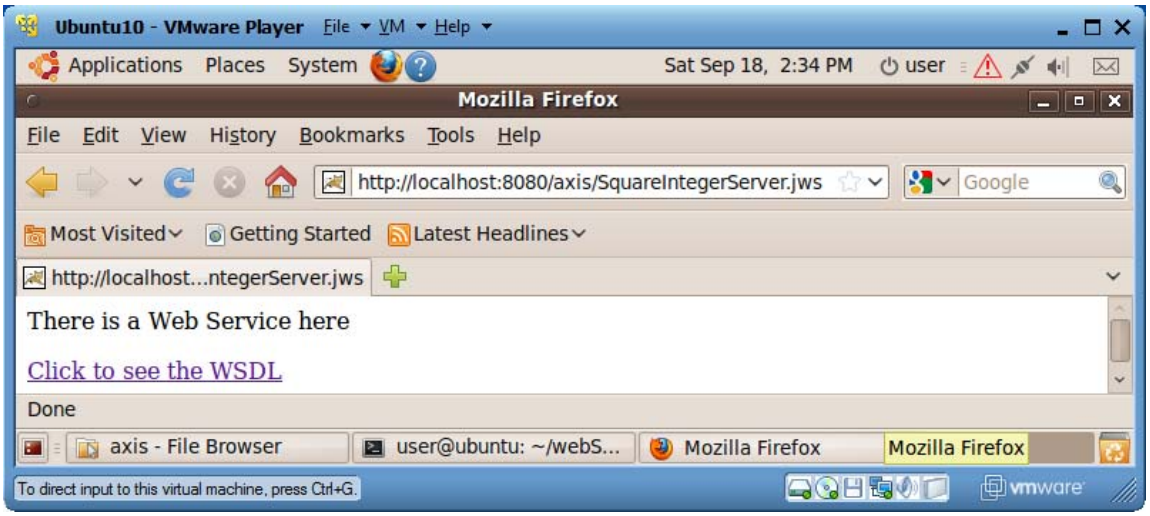
Class "SquareIntegerServer" has only one method named "square", which accepts one integer-typed parameter "x", and returns an integer. The business logic for this method is to evaluate the square of the value in variable "x". The result is then returned to the method invoker. The second line of the source code contains clause "throws Exception" because we intend to use this class to implement a web service, and there are many reasons for the network interactions between the method implementation and its remote client to go wrong. You need to explicitly declare that networking errors may happen.

After reviewing the source code, you are ready to deploy it as a web service on our Tomcat web server.

1. Start a terminal window with menu item "Applications|Accessories|Terminal".
2. Run "cd ~/webServiceLab/java/server" to change terminal working folder to "~/webServiceLab/java/server".
3. Review the contents of file "SquareIntegerServer.java" with command "more SquareIntegerServer.java".
4. Verify that the source code is a valid Java class by typing the following command in the terminal window to compile the source file:

   javac SquareIntegerServer.java [Enter]

5. In this case you will notice that a new file "SquareIntegerServer.class" is successfully created. In general, if you see error messages during compilation, you need to revise your source code to make it error-free.
6. In a file browser, rename file "SquareIntegerServer.java" to "SquareIntegerServer.jws". Here "jws" stands for "Java web service".
7. In a file browser, copy file "SquareIntegerServer.jws" to folder "~/tomcat/webapps/axis", which is the folder for you to deploy any of your Java web services. If this file is already available in folder "axis", replace it with your copy.
8. Since we just deployed a new web component, it is safer to restart *Tomcat* if it is already running. Run in a terminal window "tomcat-stop" followed by "tomcat-start". Wait until the *Tomcat* startup is complete. You can now minimize the terminal window, but don't close it.
9. Open a web browser and visit URL http://localhost:8080/axis/SquareIntegerServer.jws. You are going to see a window like the following one:




10. Your new web service is working! Click the hyperlink "Click to see the WSDL" to review the WSDL file for this web service. You will notice that the URL address box now contains value "http://localhost:8080/axis/SquareIntegerServer.jws?wsdl". As a matter of fact, given any web service URL, you can always add "?wsdl" to the end of the URL for retrieving the WSDL file for the web service. When you started the *Tomcat*, the *Axis* toolkit embedded in the *Tomcat* compiled our source file "SquareIntegerServer.jws" into a Java servlet with URL http://localhost:8080/axis/SquareIntegerServer.jws, and also generated the web service definition language (WSDL) file for describing the method exposed by this web service. Figure 2 shows the contents of the WSDL file after slight formatting. The lines in bold face specify the format of SOAP response message (for method call return value), SOAP request message (for method invocation information), and the URL for accepting the SOAP requests.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <wsdl:definitions
  targetNamespace="http://localhost:8080/axis/SquareIntegerServer.jws"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://localhost:8080/axis/SquareIntegerServer.jws"
  xmlns:intf="http://localhost:8080/axis/SquareIntegerServer.jws"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 >

 <wsdl:message name="squareResponse">
  <wsdl:part name="squareReturn" type="xsd:int" />
 </wsdl:message>

 <wsdl:message name="squareRequest">
  <wsdl:part name="x" type="xsd:int" />
 </wsdl:message>

 <wsdl:portType name="SquareIntegerServer">
  <wsdl:operation name="square" parameterOrder="x">
   <wsdl:input message="impl:squareRequest" name="squareRequest" />
   <wsdl:output message="impl:squareResponse" name="squareResponse" />
  </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="SquareIntegerServerSoapBinding"
                type="impl:SquareIntegerServer">
  <wsdlsoap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="square">
   <wsdlsoap:operation soapAction="" />
   <wsdl:input name="squareRequest">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://DefaultNamespace" use="encoded" />
   </wsdl:input>
   <wsdl:output name="squareResponse">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://localhost:8080/axis/SquareIntegerServer.jws"
                use="encoded" />
   </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="SquareIntegerServerService">
   <wsdl:port  binding="impl:SquareIntegerServerSoapBinding"
                name="SquareIntegerServer">
    <wsdlsoap:address location="http://localhost:8080/axis/SquareIntegerServer.jws" />
   </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

**Figure 2 WSDL file for the SquareIntegerServer web service**

**Question 15:** Do you write a complete program to implement a web service?

**Question 16:** Is creating a web service a highly technical task?

**Question 17:** What is the relationship between the URL for accepting web service method invocation requests and the URL for displaying the web service WSDL file?

### *7.3.2 Development of a Java Client Application for Consuming the Local Java Web Service*

Now it is time for you to develop a Java program to function as the client of the new web service.

1. In the terminal window, run "cd ~/webServiceLab/java/client" to change working folder to "~/webServiceLab/java/client" .
2. Make sure that your *Tomcat* is still running (using a web browser to visit http://localhost:8080 and make sure you could see the *Tomcat* home page). Run "./makeProxy.bat", which is equivalent to running the following command on a single line (no line break), in the terminal window to generate source files for your proxy class for your new web service:

```
java  org.apache.axis.wsdl.WSDL2Java
   http://localhost:8080/axis/SquareIntegerServer.jws?wsdl
```

This command will generate four Java source files supporting the web service proxy class. These four Java files all belong to Java package "localhost.axis.SquareIntegerServer_jws", therefore they are located in folder, relative to the current folder "~/webServiceLab/java/client", "localhost/axis/SquareIntegerServer_jws". The name and function of these four Java files are as follows:

a) **SquareIntegerServer.java**
Interface for the web service proxy class to implement

b) **SquareIntegerServerService.java**
Interface for the factory class of the web service proxy objects (proxy objects are not generated by operator **new**, but through method calls to a factory object)

c) **SquareIntegerServerSoapBindingStub.java**
Proxy class source, which implements interface **SquareIntegerServer**

d) **SquareIntegerServerServiceLocator.java**
Factory class of proxy objects; it implements interface **SquareIntegerServerService**

The names of these files and the subfolders depend on the URL and contents of the WSDL file. When you create proxy classes for a different web service, you need to change the argument to class WSDL2Java to the URL of the WSDL file of that web service. The resulting proxy class files may have different names and package path, but they should follow the same pattern as our example here.

3. Now in folder "~/webServiceLab/java/client", use a text editor, like *gedit*, to create a Java source file named "SquareIntegerClient.java" with its contents specified below (for your convenience, the file has been created for you; please review its contents):

```
import localhost.axis.SquareIntegerServer_jws.*;

public class SquareIntegerClient {
  public static void main(String[] args) throws Exception {
    int value = 0; // value to be squared
    // The program expects to receive an integer on command-line
    // Program quits if there is no such integer
    if (args.length == 1)  // there is one command-line argument
      value = Integer.parseInt(args[0]);  // parse the string form of integer to an int
    else {
      System.out.println("Usage: java SquareIntegerClient  [integer]");
      System.exit(-1);  // terminate the program
    }
    // Get the proxy factory
    SquareIntegerServerServiceLocator factory =
        new SquareIntegerServerServiceLocator();
    // Generate the web service proxy object
    SquareIntegerServer proxy = factory.getSquareIntegerServer();
    // Access the web service
    int result = proxy.square(value); // invoke server method to square value
    System.out.println("Square of " + value + " is " + result);
  }
}
```

**Program 2 SquareIntegerClient.java**

The file first imports all the Java classes generated by class "WSDL2Java" for supporting the web service proxy object. Then it checks whether there is a command-line argument. If there is no command-line argument, the program prints a usage message and then terminates. Otherwise it parses the string form of the command-line integer into an *int* number, and saves the number in variable *value*. A factory object for generating a proxy object is then created with "new SquareIntegerServerServiceLocator()", and a new proxy object is obtained by calling the factory's method "getSquareIntegerServer()". The actual web service invocation happens on the line with "proxy.square(value)". Even though this method call is to the local proxy object, the proxy's body for method "square" makes a TCP IP connection to the *Tomcat* servlet specified in the WSDL file, issues an HTTP request carrying a SOAP request message in its *entity body*, receives a response SOAP message from the *Tomcat* servlet (from the *entity body* of the HTTP response), parses the SOAP response message into a Java *int* value, and returns the value as its own method return value.

4. Now it's time to compile the client program. In the terminal window, type

   **javac  SquareIntegerClient.java  -source 1.4**

The *javac* command-line switch "-source 1.4" is for informing the Java compiler that this class should be compiled with Java SDK 1.4's API. This is because in Java SDK 1.5 or later, "enum" is defined as a Java key word, but they were not in Java SDK 1.4 or earlier. The *Axis* toolkit uses "enum" extensively as a package name, which is not allowed in Java SDK 1.5 or later. You may see two warnings related to this fact, which are harmless. After executing this command, you will notice a new file "SquareIntegerClient.class".

5. Now you can test your web service with your new Java client program. Type the following command, and the second line is the response of your client program. The computation is performed by your web service. You can try to use different integers to replace integer 2.

   **java SquareIntegerClient 2**
   Square of 2 is 4

Congratulations! You have successfully implemented a web service client application in Java. The class name of your client application is not important and you can choose any name meaningful to you.

To consume a web service implemented on another platform, say on Microsoft .NET with C# or Visual Basic .NET, you just follow the same procedure. As long as you provide the right URL for the WSDL file of the web service, the implementation language or platform of the web service is of no relevance to you. This is the real power of web services!

**Question 18:** Is the web service client a person or a program?

**Question 19:** What is the name of the class that you used to create the proxy classes?

**Question 20:** What is the relationship between the folder paths for the proxy classes and the Java package path for the proxy classes?

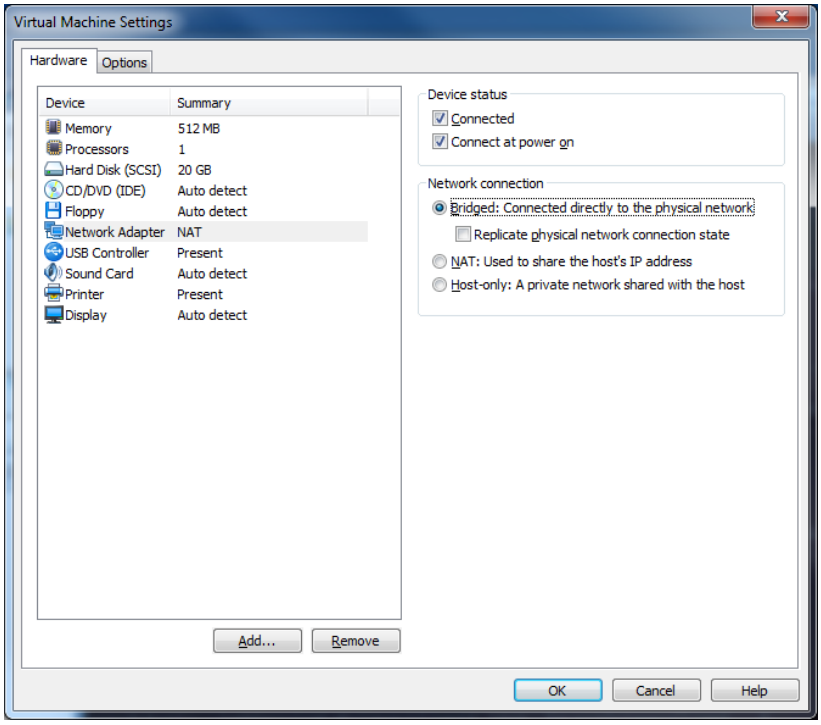
**Question 21:** What is a factory class?

**Question 22:** What are the two most important proxy classes that you use in creating a web service client program?

### *7.3.3 Running Web Service Client and Server on a LAN*

In this lab you learn how to deploy our web service server and client on different Ubuntu VMs running on different PCs in a (home) local area network (LAN). We assume that a (*CableVision* or *Verizon*) router runs a Dynamic Host Configuration Protocol (DHCP) server that assigns dynamic IP addresses for your computers in the LAN; and you plan to run the web service on computer A and run the web service client on computer B, both are part of your home LAN. Find out your router's IP address that is used for using a web browser to login to your router and configure it. This is also your DHCP server's IP address. All of your physical or virtual computers must have the same initial three integers in their IP addresses, typically "192.168.0" or "192.168.1", to be able to communicate with each other. As a special case the two computers could be the same one (in this case let the two VM folders differ in folder names, or be located in different file system location). Both of the two computers should have *VMware Player* installed. Make a copy of VM folder "ubuntu" of both computers. When you launch each of the VM for the first time, make sure you choose "I copied it" to VMware Player so your VMs would have unique network MAC values.

1. On computer A, set up the square integer web service server on a Ubuntu VM as described in Section 7.3.1.
2. Use *VMware Player* top menu "Virtual Machine|Virtual Machine Settings…" to launch the "Virtual Machine Settings" window. Choose the "Hardware" tab. Click on the "network Adapter" row. In the right "Network connection" section, check the "Bridged" check box, as shown below:




3. Click the bottom "OK" button to shut down the "Virtual Machine Settings" window.
4. Restart your VM. This is mandatory.
5. Use menu item "Applications|Accessories|Terminal" to launch a terminal window, and run in it command "sudo ifconfig". If the "eth0" or "eth1" section of the output message contains an inet address whose first three digits are the same as those of your router's IP address, then your VM's network configuration is complete and this VM can be accessed on your LAN as a physical PC.
6. Repeat steps 2-5 on your Ubuntu VM on computer B to make it part of your LAN.
7. Run "tomcat-start" to make sure Tomcat is running on computer A.
8. Complete the lab describe in Section 7.3.2 to develop the web service client on computer B, but adjust the URL for the web service WSDL file and the import line of file "SquareIntegerClient.java". In my case my Ubuntu VM on computer A has IP address 192.168.0.240. I generated the proxy classes on computer B with command

```
java  org.apache.axis.wsdl.WSDL2Java
   http://192.168.0.240:8080/axis/SquareIntegerServer.jws?wsdl
```

Find the path for your new proxy classes, and adjust the import statement of file "SquareIntegerClient.java" accordingly. If the proxy files are in folder "localhost/axis/SquareIntegerServer_jws", then you don't need to change the import statement of file "SquareIntegerClient.java". Sometimes the proxy class package path may match the IP address of the web service server. In my experiment, since my computer A has address 192.168.0.240, my proxy classes are in Java package

"_240._0._168._192.axis.SquareIntegerServer_jws". Therefore I need to use text editor "gedit" to change the first line of file "SquareIntegerClient.java" to

```
import  _240._0._168._192.axis.SquareIntegerServer_jws.*;
```

9. The web service client on computer B should be able to invoke the web service on computer A and provide the same function of squaring integers.

**Question 23:** What is the function of a DHCP server?

**Question 24:** Suppose your home LAN has network mask "255.255.255.0". Can your home computers communicate if their IP addresses differ on any of the first three numbers?

**Question 25:** If two VMs are deployed on the same PC and you want them to communicate with each other and with the host PC. How should you configure the VMs' network adapters?

**Question 26:** If two VMs are deployed on two different PCs on your LAN and you want them communicate with each other and with the two host PCs. How should we configure the VMs' network adapters?

**Question 27:** On a Linux computer which command will tell us the IP address and network mask of the computer?

### *7.3.4 Developing a Java Client Application to Consume a Public Remote ASP .NET Web Service*

In this lab you will develop a Java client application to consume the remote public web service for finding US holidays in a year. The web service was developed with Microsoft ASP .NET and described at http://www.holidaywebservice.com/Holidays/US/Dates/USHolidayDates.asmx. The WSDL file for the web service is at http://www.holidaywebservice.com/Holidays/US/Dates/USHolidayDates.asmx?WSDL. That is all information we have about this web service. This lab provides guidance on how to consume unknown public web services.

1. If not yet, launch the *Ubuntu* VM.
2. Start a terminal window with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/webServiceLab/java/client" to change the working folder of the terminal window to the folder for developing web service Java clients. The only reason that we use this folder is because we have an example web service client program here for our reference. You could conduct this lab in any folder.
6. Run the following command on a single line (no line break) in the terminal window to generate source files for your proxy class for your new web service:

```
java  org.apache.axis.wsdl.WSDL2Java
   http://www.holidaywebservice.com/Holidays/US/Dates/USHolidayDates.asmx?WSDL
```

This command will generate five Java source files supporting the web service proxy class. These files have names "USHolidayDates.java", "USHolidayDatesLocator.java", "USHolidayDatesSoap.java", "USHolidayDatesSoapStub.java", and "USHolidayDatesSoap12Stub.java" respectively. These five Java files all belong to Java package "com._27seconds.www.Holidays.US.Dates", therefore they are located in folder, relative to the current directory "~/webServiceLab/java/client", "com/_27seconds/www/Holidays/US/Dates".

By comparison with the file names for the Square Integer proxy class files, we guess class "USHolidayDatesLocator" is the factory for creating the proxy object, and "USHolidayDates" is a Java interface listing web service methods that we could call. We use *gedit* to open file "USHolidayDatesLocator.java" and did find the factory method "USHolidayDatesSoap getUSHolidayDatesSoap()". Therefore our guess about the proxy factory class is confirmed. The return data type "USHolidayDatesSoap" of the factory method told us that the interface for the web service methods is in file "USHolidayDatesSoap.java". Use *gedit* to review the contents of this file and we see a long list of web service method that we could use including

```
public interface USHolidayDatesSoap extends java.rmi.Remote {

  // Get the date of New Year.
  public java.util.Calendar getNewYear(int getNewYearYear)
    throws java.rmi.RemoteException;

  // Get the date of Martin Luther King Day.
  public java.util.Calendar
    getMartinLutherKingDay(int getMartinLutherKingDayYear)
    throws java.rmi.RemoteException;

  // Get the date of President's Day.
  public java.util.Calendar getPresidentsDay(int getPresidentsDayYear)
    throws java.rmi.RemoteException;
  ……
}
```

All these methods have an integer parameter for the year, and return a Java "java.util.Calendar" object representing a date. You should visit Java API documentation at http://download.oracle.com/javase/7/docs/api/ to learn how to print the dates from a *Calendar* object. In the center "Packages" column we click on "java.util", then click on the "Calendar" link in the "Class Summary" section of the "Package java.util" page. Here you learn how to retrieve the year, month and day components of a *Calendar* object.

7. With reference to file "SquareIntegerClient.java", you create the new file "USHolidaysClient.java" with the following contents:

```
import com._27seconds.www.Holidays.US.Dates.*;
import java.util.*;

public class USHolidaysClient {
  public static void main(String[] args) throws Exception {
    int year = 0; // year
    // The program expects to receive an integer (year) on command-line
    // Program quits if there is no such integer
    if (args.length == 1)  // there is one command-line argument
      year = Integer.parseInt(args[0]); // parse the string to an int
    else {
      System.out.println("Usage: java USHolidaysClient  [integer]");
      System.exit(-1);  // terminate the program
    }
    // Get the proxy factory
    USHolidayDatesLocator factory = new USHolidayDatesLocator();
    // Generate the web service proxy object
    USHolidayDatesSoap proxy = factory.getUSHolidayDatesSoap();
    // Access the Web service
    Calendar  cal;
    cal = proxy.getEaster(year);
    cal.add(Calendar.DAY_OF_MONTH, 1);
    System.out.println("Easter: \t" + (cal.get(Calendar.MONTH)+1) +
      "/" + cal.get(Calendar.DAY_OF_MONTH) + "/" + cal.get(Calendar.YEAR));
    cal = proxy.getMothersDay(year);
```

```
        cal.add(Calendar.DAY_OF_MONTH, 1);
        System.out.println("Mother's Day: \t" + (cal.get(Calendar.MONTH)+1) +
          "/" + cal.get(Calendar.DAY_OF_MONTH) + "/" + cal.get(Calendar.YEAR));
        cal = proxy.getFathersDay(year);
        cal.add(Calendar.DAY_OF_MONTH, 1);
        System.out.println("Father's Day: \t" + (cal.get(Calendar.MONTH)+1) +
          "/" + cal.get(Calendar.DAY_OF_MONTH) + "/" + cal.get(Calendar.YEAR));
        cal = proxy.getThanksgivingDay(year);
        cal.add(Calendar.DAY_OF_MONTH, 1);
        System.out.println("Thanksgiving: \t" + (cal.get(Calendar.MONTH)+1) +
          "/" + cal.get(Calendar.DATE) + "/" + cal.get(Calendar.YEAR));
        cal = Calendar.getInstance();
        System.out.println("Today: \t\t" + (cal.get(Calendar.MONTH)+1) +
          "/" + cal.get(Calendar.DAY_OF_MONTH) + "/" + cal.get(Calendar.YEAR));
    }
}
```

Note that the web service methods always return one day earlier (discovery by testing), therefore I added one day to each of the returned dates. Java Calendar uses 0 for January, therefore you need to add 1 to the month value for printing.

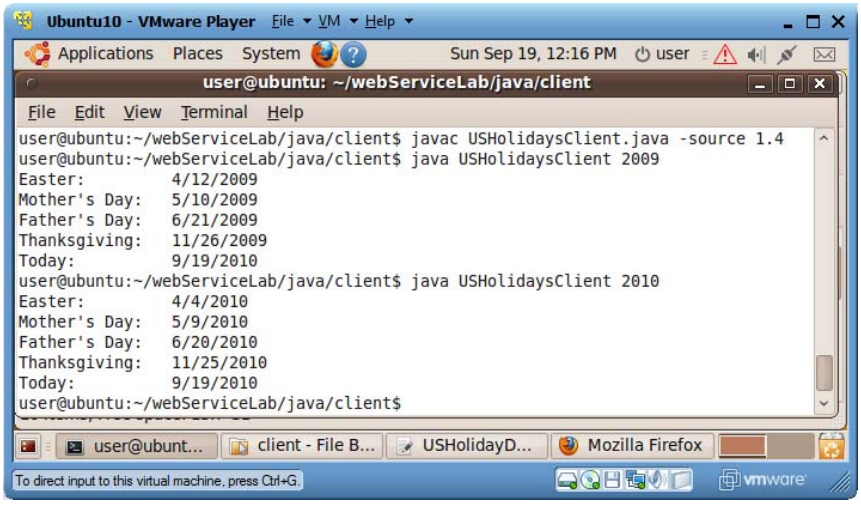
8. To compile the client program, type in the terminal window

   **javac USHolidaysClient.java -source 1.4**

9. To run the client program, type in the terminal window

   **java USHolidaysClient 2010**

   and you can change 2010 to any valid year. The following is the screen capture of one of my test session.




**Question 28:** Do you need know web service server implementation before you could develop a web service client to get its service?

**Question 29:** To use a new web service over the Internet, do you need to install special software, like plug-ins, for that service?

## 7.4 Review Projects

**Question 30:** There is a public .NET web service with its WSDL file at http://www.restfulwebservices.net/wcf/CurrencyService.svc?wsdl that provides conversion rate for given two currencies. Develop a Java web application to consume this service.

**Question 31:** Develop a Java web service that supports method "double add(double x, double y)" for returning the summation of two floating point numbers; and develop a Java client program to consume this web service. To convert string "12.5" into floating-point number 12.5, you could use "double d = Double.parseDouble("12.5");".

# 8 Introduction to Cryptography

## 8.1 Concepts

Secure communications on the Internet or web is the foundation of network security and web security. Cryptography is the practice and study of how to hide information from potential enemies, hackers or the public. The sender encrypts a message with a small piece of secret information (*key*), and then sends the encrypted message to the receiver. The receiver decrypts the encrypted message with a small piece of secret information (a key that is same or different from the key used by the sender) and recovers the original message. People who don't have the right keys would not be able to read the message even if they steal a copy of the encrypted version.

There are two categories of cryptographic systems: single key symmetric ciphers or dual-key public key ciphers.

### *8.1.1 Symmetric Secret Key Ciphers*

With this approach (also called *conventional ciphers*), the sender and the receiver use the same *secret key* (secret information) to encrypt and decrypt messages. Many algorithms can do both encryption and decryption. The popular symmetric key algorithms (ciphers) include DES and AES. When the input plain data is long, they divide the data into equal-sized data blocks (except the last block) and encrypt/decrypt the successive data blocks with the same algorithm and key. In this lab you will learn how to use GPG (GNU Privacy Guard), the open-source version of PGP (Pretty Good Privacy), to experiment with symmetric key encryption/decryption.

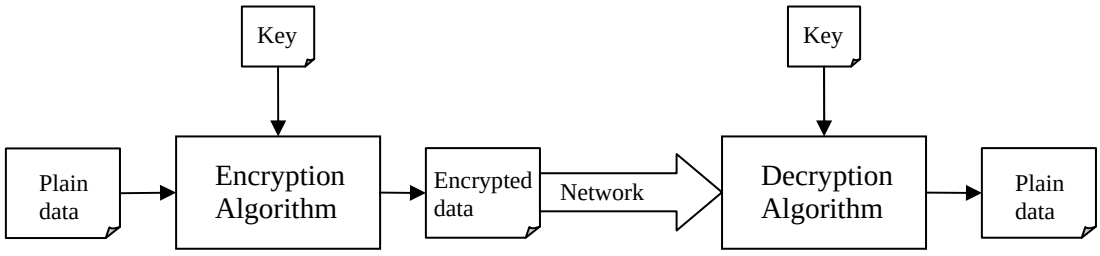



**Figure 3 Data encryption/decryption**

Symmetric key ciphers are also called secret key ciphers. They are much more efficient than public key ciphers described next.

### *8.1.2 Public Key Ciphers*

With this approach a pair of public key and private key will be generated together. You can use either of the two keys to encrypt the plain data, and then use the other key to decrypt the encrypted data. For example you can encrypt data with the public key and use the private key to decrypt the data. RSA and Diffie-Hellman are the two most widely used public key algorithms

Typically, the key owner will keep the private key and distribute the corresponding public key to his/her potential communication partners. There are two typical application scenarios:

1. Author and contents validation. If the key owner needs to distribute a message to his friends and assure them the message is really originated from the owner without modification by any third parties, the owner would encrypt the message with the private key. If the receivers could decrypt the message with this owner's public key, they know that the message is really sent by that owner and the message has not been modified.
2. Many-to-one private messages. If a friend needs to send a private message to Bob, he could encrypt his message with Bob's public key and then send the resulting message to Bob, and only Bob, the owner of the right private key, could decrypt the private message.

Each computer maintains the public/private keys of the computer user in a file called *key store*, and the owner needs to set up passwords to limit the access to the key store.

Public key ciphers are less efficient than symmetric/secret key ciphers. They are mainly used for distributing the secret keys used by symmetric key ciphers, and authenticate and validate documents (here the document contents are usually not encrypted).

In this lab you will use GPG (GNU Privacy Guard), the open-source version of PGP (Pretty Good Privacy), to experiment with public/private key creation and public key encryption/decryption.

### *8.1.3 Hash Function and Digital Signature*

While you could use public/private key pairs to authenticate the author of a message and validate the contents of the message, it would be slow if the message is long. Digital signatures are designed to make author and contents validation more efficient. When you digitally sign a document, you normally (not necessary) keep the document in plain form so everyone could read it, and you append a digital signature, which is a small piece of data, to the end of the plain document so the receiver could validate the author and validity of the public document if necessary.

You first need to compress the variable-length document into a short fixed-length string (popularly called *fingerprint*, *digest* or *hash code*). You use a hash function to do so. A hash function reads a long document, and produces a fixed-length short string, called *fingerprint* (*hash code* or *digest*), so that each bit of the fingerprint depends on as many bits of the input document as possible. Even though not possible in theory, in practice the hash function establishes a one-to-one mapping between the plain documents and the fingerprints with high probability: if someone modifies the plain document, its fingerprint would change. The application of a hash function on the same document always generates the same fingerprint. SHA-1 and MD5 are both examples of hash functions. While MD5 uses 128 bits for fingerprints, SHA-1 uses 160 bits for fingerprints so it is less likely to produce the same fingerprint from two different files. In this lab you will learn how to use SHA1 and MD5 to generate fingerprints (sums) of files so you could be sure whether the downloaded large files have been compromised. There are also hash functions SHA224, SHA256, SHA384 and SHA512, which are all variants of SHA1 and use more bits for fingerprints to reduce the chance of fingerprint collision (different files have the same fingerprint).

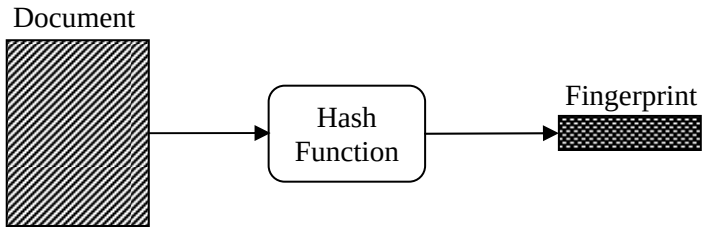



**Figure 4 Hash function**

Figure 5 shows how a document is digitally signed by its author. A hash function reduces the document into a fixed-size fingerprint, which is then encrypted by the author's private key into a digital signature. The digital signature is then appended to the end of the original document for distribution.

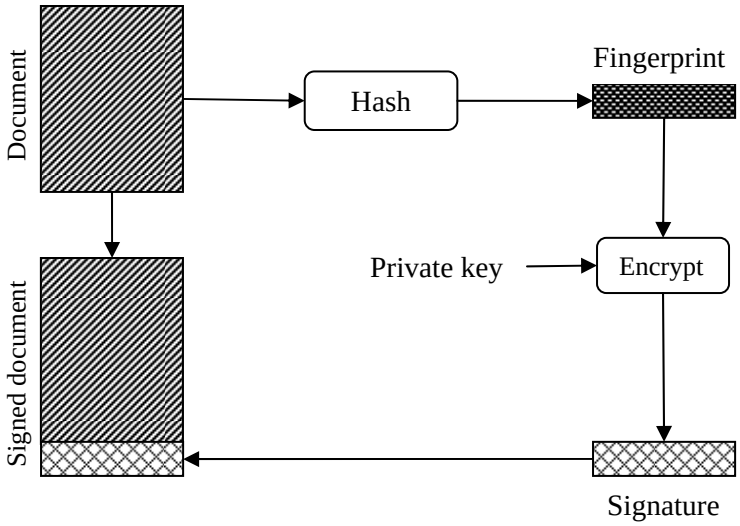



**Figure 5 Digital signing of a document**

Figure 6 shows how the signed document is authenticated for its author and validated against any compromises. The signed document is separated into the plain document and digital signature two parts. The same hash function maps the plain document to its fingerprint, which is then compared with the fingerprint (hash code) decrypted from the received signature with the author's public key. If the two fingerprints are the same, then the document is from the author and its contents are intact, otherwise the document has been compromised.

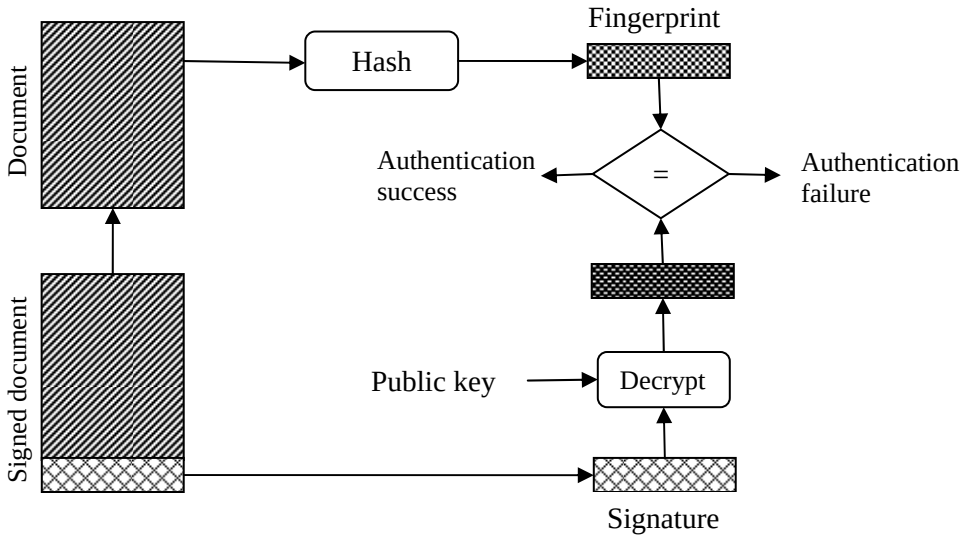



**Figure 6 Authentication and validation of a signed document**

### *8.1.4 Digital Certificates*

One challenge of using public key ciphers is how to assure the public that the public keys they receive are actually those from their owners. Hackers could forge a public key and then distribute it under another person's name.

Digital certificates are designed to solve this problem. A few certificate authority (CA) companies are set up and supposed to be trusted by the public (can we really totally trust them or their employees?). VeriSign, GTE and U.S. Postal Service are a few example CAs. These CAs have generated their public/private key pairs, and distributed their public keys to the public computers in some "safe" way including hardcoding them in OS distributions (are they really safe?). When a person needs to distribute his public key to the public, he needs to apply to one of the CAs to create a digital certificate for him. A digital certificate includes the following plain text information: certificate format version number, certificate serial number with the CA, algorithm and parameters for signing the certificate, CA name, period of validity of the certification, the name of the person or company for which this certificate distributes its public key, the public key of the person of company that requested for this certificate, the algorithm and parameters to use the public key, and the digital signature of the above plain contents signed with the CA's private key. This certificate signing process is exactly the one described in Figure 5 where document is replaced with the above certificate plain information. When a computer receives such a digital certificate, if it finds that the certificate is correctly signed by the CA (with a process described in Figure 6), then the public key and its owner's name will be added to its key store and the certificate owner becomes a trusted entity of the computer.

But be aware that anyone can apply for a digital certificate with a CA. There are three classes of digital certificates. Class 1 certificates will be issued by a CA as long as the applicant has a valid email address. Class 2 certificate applications also need to go through an *automated* address check (a postal letter will be sent to the applicant to warn the creation of the digital certificate). Only for class 3 certificates that the applicants really need to make in-person appearances to produce ID documents as well as business records for organizations.

## 8.2 Lab Objectives

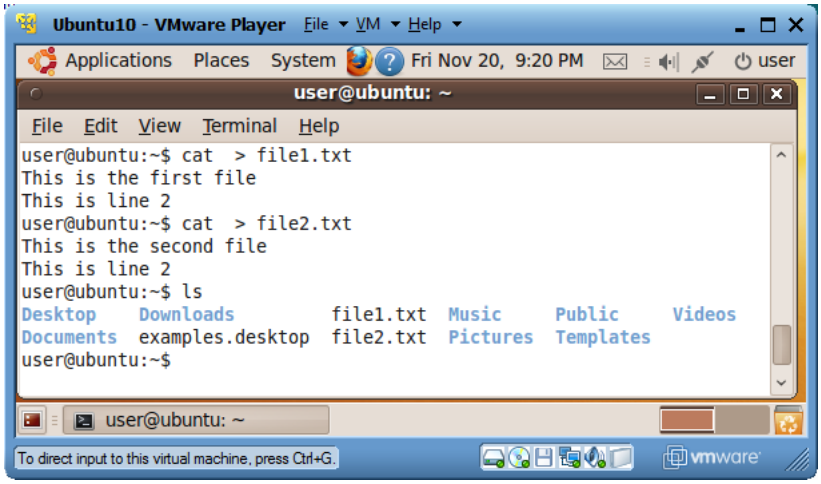
In this lab you will

1. Learn and practice how to use MD5 and SHA1 to generate hash codes of strings or large files, and verify whether a downloaded file is valid;
2. Learn and practice how to use GPG to encrypt/decrypt files with symmetric algorithms;
3. Learn and practice how to use GPG to generate public/private key pairs and certificates, distribute the certificate with public key to a friend, let the friend encrypt a document with the public key, and let the key owner decrypt the document with the private key.

## 8.3 Lab Guide

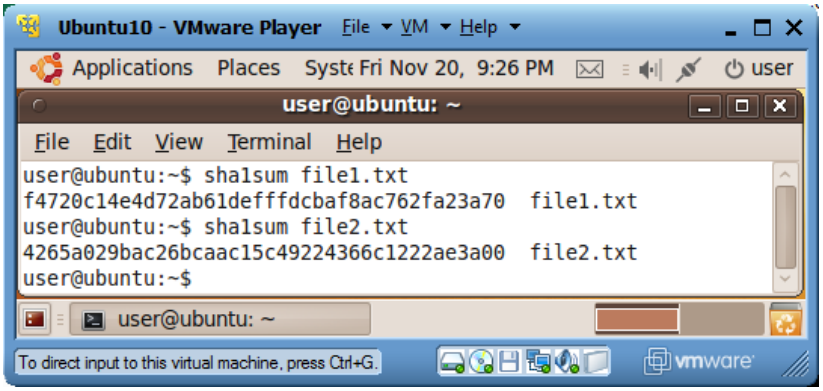
### *8.3.1 Hashing Files with MD5 and SHA-1*

1. Launch the Ubuntu VM with username “user” and password 12345678.
2. Start a terminal window in home folder ~ with menu item “Applications|accessories|Terminal”.
3. Create the first file “file1.txt” by typing
   $ cat > file1.txt [Enter]
   This is the first file[Enter]
   This is line 2[Enter]
   [Ctr-d]
4. Create the second file “file2.txt” by typing
   $ cat > file2.txt [Enter]
   This is the second file[Enter]
   This is line 2[Enter]
   [Ctr-d]

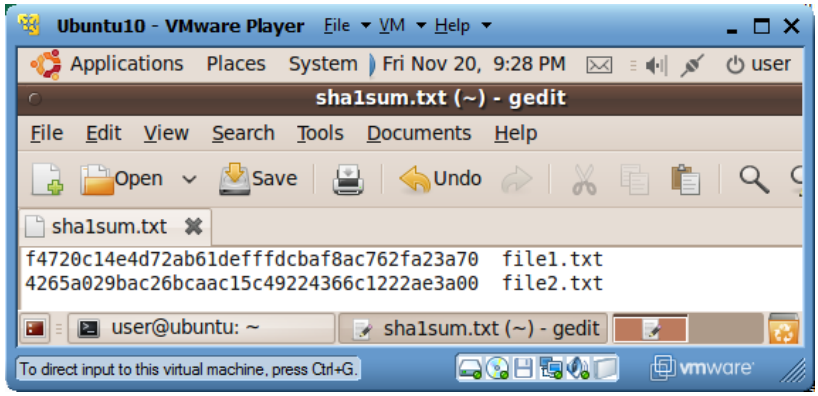



5. Run “sha1sum file1.txt” and “sha1sum file2.txt” to generate the hash codes (sums) for the two files. Each execution generates a line of two entries. The second entry is a file name, and the first

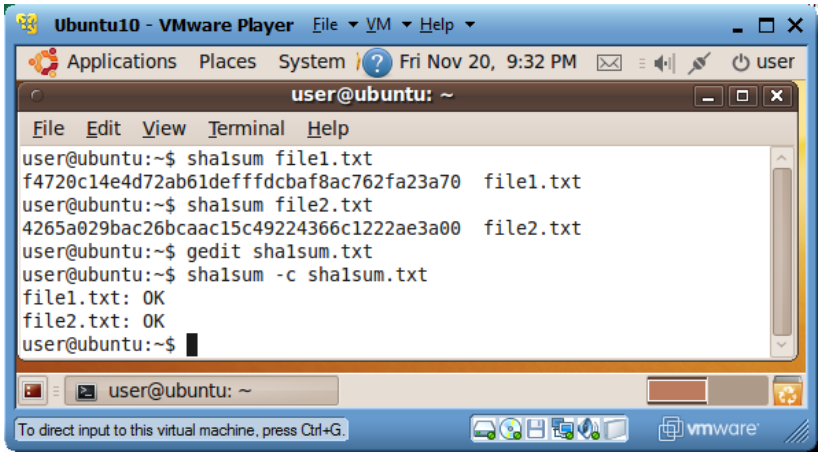
entry is the hash code of the contents of the file whose name is the second entry. The hash codes are printed in hexadecimal.



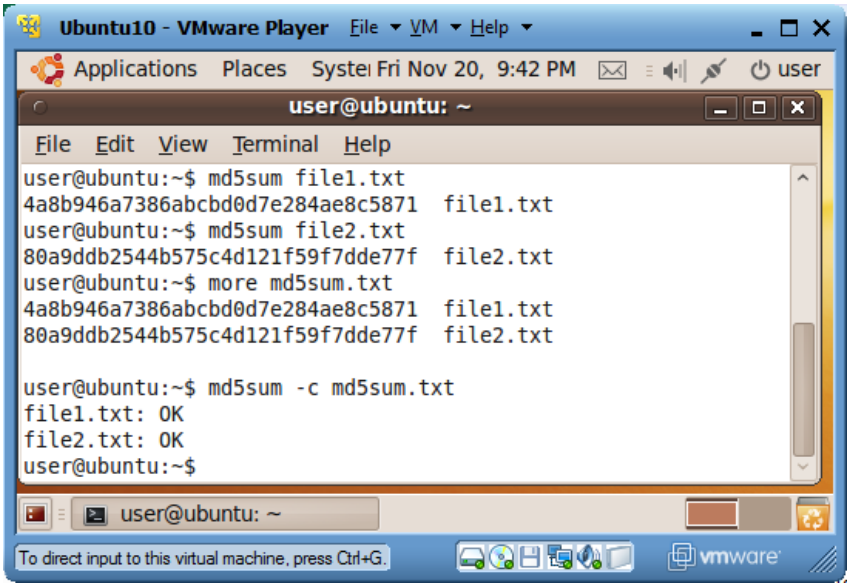

6. Run “gedit sha1sum.txt &” to create a new text file “sha1sum.txt”, and copy the two output lines of the last step into this file. Save the file.




7. Run “sha1sum -c sha1sum.txt”. In this case program “sha1sum” will read file “sha1sum.txt”. For each line in this file, it will check whether the SHA1 hash code generated for the contents of the second entry (file) is the same as the first entry (SHA1 hash code calculated beforehand). If they match, the program will print out OK for the file.



8. Now redo steps 4, 5 and 6 but replacing command “sha1sum” with “md5sum” and replacing file name “sha1sum.txt” with “md5sum.txt”. You will notice that the MD5 hash codes are shorter and they serve the same purpose of file contents validation.




When you download large files, like ISO disk image files, you should download their corresponding MD5Sum or SHA1SUM files so you could check whether the downloaded files are valid or corrupted.

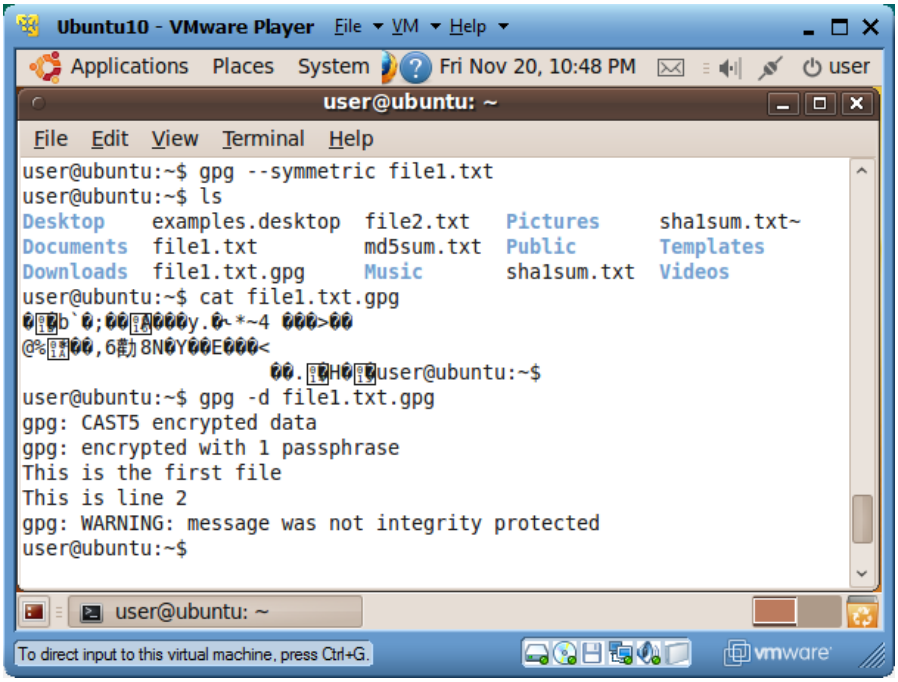
**Question 32:** Are all MD5 hash codes for different files of the same length?

**Question 33:** Do different files always lead to different MD5 or SHA1 hash codes?

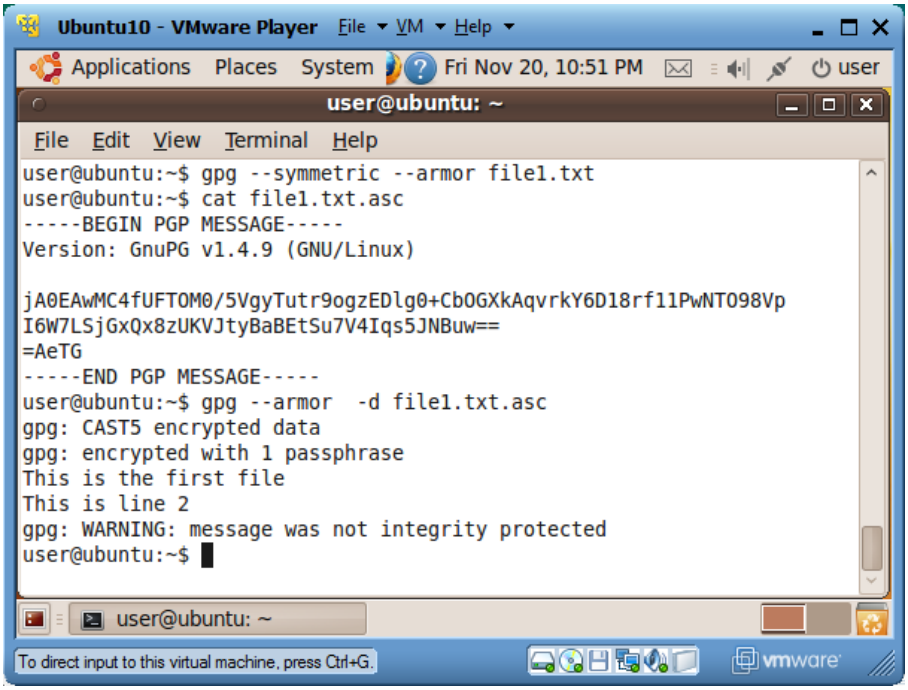
**Question 34:** In what sense SHA1 is better than MD5?

### *8.3.2 Symmetric Key Encryption/Decryption with GPG*

1. Launch the Ubuntu VM with username “user” and password 123456.
2. Start a terminal window in home folder ~ with menu item “Applications|accessories|Terminal”.
3. If you have not created file “file1.txt” yet, do so by following step 3 of the last exercise.
4. Run “gpg --symmetric file1.txt” to encrypt file ‘file1.txt”. The encrypted version is in file “file1.txt.gpg”. Upon request, enter a passphrase (password) as the secret key.
5. Run “cat file1.txt.gpg” to review the contents of file “file1.txt.gpg”.
6. Run “gpg -d file1.txt.gpg” to decrypt the file. Upon request, enter the same passphrase (password) that you used for the encryption of the file.




7. If you need to paste the encrypted data in email body instead of using email attachment, then you can run "gpg --symmetric --armor file1.txt" to generate the encrypted data in text form in file "file1.txt.asc". Upon request, enter a passphrase (password) as the secret key.
8. Run "cat file1.txt.asc" to review the contents of file "file1.txt.asc".
9. To decrypt file "file1.txt.asc", run "gpg --armor -d file1.txt.asc". Upon request, enter the same passphrase (password) that was used to encrypt the file.




GPG uses a strong cipher CAST5 to do symmetric encryption so it is much more resistant to attack than using WinZip. However the passphrase is the weak point: the longer and more complex the passphrase,

the more secure the file. A single dictionary word can be brute forced in only a few hours, so use a complex passphrase of multiple words broken up with letters and symbols.

**Question 35:** If you receive a secret message in an email body and the message is encrypted by a command like "gpg --symmetric --armor message.txt", which steps should you take to recover the message?

### *8.3.3 Public/Private Key Creation and Encryption/Decryption*

#### 8.3.3.1 Basic Concepts of PGP (GPG) Digital Certificates and Public Key Ciphers

*PGP (Pretty Good Privacy) is a c*omputer program that provides cryptographic privacy and authentication. PGP supports public/private key pairs to implement secure data communications between communicating parties. GPG (GNU Privacy Guard) is the open-source version of PGP.

Suppose Mike needs to send a secure message in file, say *msg-to-Alice*, to Alice so that no other people can read the message. Both Mike and Alice need to have used the following command to generate their own public/private key pairs:

```
gpg --gen-key
```

Each of them will be prompted to enter a name, an email address, and a comment, which together make the person's *user-ID*, in form of "name (comment) email-address", capable of identifying the person. Suppose Mike uses email address mike@pace.edu, and Alice uses email address alice@pace.edu. Each person will also be prompted to enter a *passphrase* to protect his/her private key. Each time a person uses his/her private key, he/she needs to enter the passphrase to prove his/her ownership to the private key.

Alice needs to export her public key into a file, say "*alice-pk*", with a command like

```
gpg --armor --output alice-pk --export alice@pace.edu
```

Alice needs to send her public key (in file "*alice-pk*") to Mike in any secure way, like with a USB flash disk, and making sure no substitution or modification of the public key file in the key distribution process. Mike now needs to import Alice's public key with the following command:

```
gpg --import alice-pk
```

Then Mike can use the following command to encrypt file *msg-to-alice* into a new file *secret-to-alice*:

```
gpg --recipient alice@pace.edu --output secret-to-alice --encrypt msg-to-alice
```

Now Mike can send file *msg-to-alice* to Alice in any way and only Alice could decrypt the file with command

```
gpg --output msg-from-mike --decrypt secret-to-alice
```

The decrypted message is now in file *msg-from-mike*. This command will only work if the computer has the private key of Alice.

You can also use command "gpg --list-keys" to list all public keys on your system, and use commands like "gpg --delete-key marge@pace.edu" to delete the selected public keys from your system.

In the above process, it is critical for Mike to be sure that the imported public key from Alice is from its real owner Alice. The *fingerprint* of a public key is an easier-to-compare short string uniquely identifying a public key. Alice and Mike can independently generate the fingerprints of Alice's public key and compare them in a trusting way like over the phone or in person. If the fingerprints are the same, Mike could *sign* Alice's public key to claim that he trusts the validity of the key, and this signing process will insert Mike's user ID in the key's signature list. To work on the tasks described in this paragraph, Mike could first run command "gpg --edit-key alice@pace.edu" to enter the key-editing user interface for Alice's public key, then use sub-command "fpr" to generate the signature for Alice's public key, use sub-command "sign" to sign Alice's public key with Mike's private key, and use sub-command "check" to review the key's list of endorsing signatures for the validity of Alice's public key.

From the above discussion we can see that for Mike to send a secure message to Alice, Alice needs to inform Mike of her public key as well as her email address used to generate the public key. Since the email address is part of the public key and listed when Mike imports Alice's public key, actually Alice only needs to pass her public key to Mike.

#### 8.3.3.2 A Detailed Lab Guide for GPG

This lab exercise guides you to practice the above PGP (GPG) concepts with GPG in our *Ubuntu* VM.

1. Create Linux Accounts for Alice and Mike

    a. Launch your *Ubuntu* VM, and start a terminal window.

    b. Run command "sudo adduser alice" to create a Linux account for Alice. Use 123456 as password.

    c. Run command "sudo adduser mike" to create a Linux account for Mike. Use 123456 as password.

    d. Run command "sudo visudo" to launch file "/etc/sudoers.tmp" in a text editor, insert the following two lines at the end of the file, and then use Crtl+O to write out the revised contents, and use Ctrl+X to exit the editor. This step will enable Alice and Mike to use "sudo".

```
alice   ALL=(ALL) NOPASSWD: ALL
mike  ALL=(ALL) NOPASSWD: ALL
```

2. Run as Alice and Mike in two terminal windows

    a. In the terminal window, run "sudo login", and then login as Alice.

    b. Start a new terminal window, run "sudo login", and then login as Mike.

3. Generate keys for Alice

    a. In Alice's terminal window, run "gpg --gen-key" to generate her public and private keys. Enter "DSA and Elgamal" for key kind, 2048 for key size, "key does not expire" for key expiration

date, “Alice” for real name, alice@pace.edu for email address, “Alice’s keys” as comment, “O” for okay, and “Alice’s passphrase” for passphrase. You may need to type over 284 random keys to generate enough entropy so the keys could be created.

4. Generate keys for Mike

    a. In Mike’s terminal window, run “gpg --gen-key” to generate his public and private keys. Enter “DSA and Elgamal” for key kind, 2048 for key size, “key does not expire” for key expiration date, “Michael” for real name, mike@pace.edu for email address, “Mike’s keys” as comment, “O” for okay, and “Mike’s passphrase” for passphrase. You may need to type over 284 random keys to generate enough entropy so the keys could be created.

5. Export Alice’s public key to Mike

    a. In Alice’s terminal window, run “gpg --armor --output alice-pk --export alice@pace.edu” to dump Ali’s public key in file “alice-pk”. You can run “more alice-pk” to review the public key.

    b. Run “sudo cp alice-pk /home/mike” to copy Alice’s public key file “alice-pk” to Mike’s home folder.

    c. In Mike’s terminal window, verify the existence of file “/home/mike/alice-pk” by running “ls” in Mike’s home folder ~ (/home/mike).

    d. In the same Mike’s terminal window, run “gpg --import alice-pk” to import Alice’s public key into Mike’s key store.

    e. In the same Mike’s terminal window, run “gpg --edit-key alice@pace.edu” to enter the editing session for Alice’s public key. Type sub-command “fpr” to review the fingerprint of Alice’s public key. Type sub-command “sign” to sign this key with Mike’s key. You will be asked to enter Mike’s passphrase, which is “Mike’s passphrase”. Type sub-command “check” to review who is on the signature list of Alice’s public key, and we will see Alice (self-signature) and Mike on the list to confirm the validity of the key. You type sub-command “quit” to exit the editing session, and confirm to save the changes.

6. Create and encrypt a message

    a. In Mike’s terminal window, run “cat > msg-to-alice” followed by the ENTER key, type “Alice’s secret message”, and then type key combination Ctrl+D to close the file. You just created a new text file “msg-to-alice” with contents “Alice’s secret message”.

    b. In Mike’s terminal window, run “gpg --recipient alice@pace.edu --output secret-to-alice --encrypt msg-to-alice” to generate a new file “secret-to-alice” containing the encrypted version file “msg-to-alice”.

    c. In Mike’s terminal window, run “more secret-to-alice” to review the encrypted version of the message.

d. In Mike's terminal window, run "sudo cp secret-to-alice /home/alice" to copy file "secret-to-alice" to Alice's home folder "/home/alice".

e. In Alice's terminal window, run "ls" in Alice's home folder ~ (/home/alice) to verify the existence of file "secret-to-alice".

7. Decrypt the message

a. In Alice's terminal window, run command "gpg --output msg-from-mike --decrypt secret-to-alice" to decrypt the contents of file "secret-to-alice" and save the result in a new file "msg-from-mike". Upon request for passphrase, enter "Alice's passphrase".

b. In Alice's terminal window, run "more msg-from-mike" to review the decrypted message from Mike.

## 8.4 Review Questions

**Question 36:** Suggest some secure ways for distributing symmetric or public keys.

**Question 37:** Is email a secure way for distributing symmetric or public keys.

**Question 38:** Suppose Tom has the public key of Lisa. What is the best way for Tom to send his public key to Lisa?

**Question 39:** Suppose Tom has the public key of Lisa. What is the best way for Tom to send a secret message to Lisa?

**Question 40:** If a Word file is digitally signed, is the file also normally encrypted so it cannot be eavesdropped?

**Question 41:** Can you totally trust a company if that company has a digital certificate signed by VeriSign?

**Question 42:** Can technologies alone completely solve the network or web security problems?

# 9 Introduction to Java Security

## 9.1 Concepts

Java is a popular programming language for server-side computing for two main reasons. First it is among the first languages that support the more efficient light-weight threads, instead of the heavy-weight processes, to support multi-tasking and multi-processor computing which have important impact on server performance. Second as an interpreted language it could check each bytecode instruction for security vulnerabilities just before the instruction is scheduled to run. At the same time Java is also popular on client platforms. Java JRE plug-ins enable web browsers to run Java applets to extend browser functionality, and standalone Java applications provide GUI-rich services running directly on the operating systems. This tutorial introduces the concepts and tools for supporting Java security.

Since the Java applets are downloaded from the web and not explicitly activated by the users, they introduce severe security concerns. By default all applets always run under the constraints of a Java security manager. The example constraints for applets include

- Applets cannot access user computer resources like keyboard and files.
- Applets can only communicate with the web site from which they were downloaded.

In this tutorial you will learn how to assign special rights to secure applets so they could provide more functionality.

On the other hand, by default Java applications are usually activated by the users and thus considered safe and not constrained by the Java security manager. But if an application is downloaded from the web, we would not be sure of its security. In this tutorial you will learn how to run such stand-alone applications under the supervision of the Java security manager so you could control which resources of yours could be used by the applications.

In the tutorial "Introduction to Cryptography" you learned how to use GPG (GNU version of PGP) to generate public/secret keys and digital signatures to secure data communications. Java has utilities to accomplish the similar tasks. Java also has application programming interfaces (API) to secure applications. In this tutorial you will learn how to secure your applets and applications with Java security utilities, and review sample Java code to see how selected security tasks could be accomplished with the Java APIs.

**Question 1:** Why is Java popular for server-side or enterprise computing?

**Question 2:** What are the main differences between Java applications and Java applets?

**Question 3:** Why the Java platform normally trusts applications but not applets?

**Question 4:** What are the differences between Java utilities and Java API?

### *9.1.1 Basic Terms*

Encryption/decryption algorithms are the basic tools for computer security. To encrypt a plain document, we need a key, which is a small piece of fixed-length data. The encryption algorithm reads the plain

document and a key and produces the encrypted document. The decryption algorithm reads the encrypted document and a corresponding key to regenerate the original plain document. There are two categories of encryption/decryption algorithms. The symmetric or secret key algorithms use the same key for encryption and decryption. These algorithms are more efficient for encrypting/decrypting large volume of data. The public key algorithms use a pair of public and private keys: if a document is encrypted by one of the two keys, the other key can be used to decrypt the document. These algorithms are mainly for secret key exchange, identity authentication and data validation.

For a person to distribute a document secretly, he could first generate a pair of private/public keys as his own identity (he could reuse these keys), use the private key to encrypt the document, and send the encrypted document and the public key to the receiver. The receiver then can use the public key to decrypt the document, and be assured that the document is from the sender (identity authentication) and the document has not been modified (data validation). But here we assume that the public key itself has been distributed in a secure way, which is normally implemented with digital signatures and certificates explained below.

Sometimes the document contents are public, and we just need to assure the document receiver that the document is originated from the right sender and it has not been modified along the way. In this case you could use digital signatures to achieve sender identity authentication and data validation without the time-consuming encryption/decryption process for the document itself. The document is first transformed by a hash function (typically MD5 or SHA1) into a fixed-length short sequence of bytes, called a fingerprint, so that each byte of the fingerprint depends on many characters of the document and any change to the document would change the fingerprint with high probability. The fingerprint is then encrypted with the sender's private key into a fixed-length digital signature. As long as the receiver gets a copy of the plain document, its digital signature, and the sender's public key, the sender could reproduce the fingerprint from the digital signature and the sender's public key, generate his own copy of fingerprint from the plain document, and compare the two fingerprints. If they are the same, the document was originated from the sender and it has not been compromised along the way. Otherwise the document should not be trusted.

Now we can consider the earlier problem of how we could securely distribute the public keys, a critical step in identity authentication and data validation. A digital certificate is used to certify the identity and public key of a digital signer, who is a person or a company and needs to assure other people of the authenticity of his documents or applications). A certificate is a small record containing the digital signer's public key and identity information (work unit, company, address), and the digital signature of the public key and identity information generated with a private key of the certifier, the person or company that certifies the authenticity of information in this certificate. If the certifier is the digital signer himself, the certificate is self-certified and it does not authenticate the information in the certificate. If the certifier is another person or company trusted by a user of the certificate, then the user's trust to the certifier can now be used to authenticate the information in the certificate. A few certificate authority (CA) companies, including VeriSign and GTE, are set up and supposed to be trusted by the public, and their self-certified certificates are distributed to user computers as trusted certificates either by software (OS or application) installation, or by the user's agreement. To distribute his public key to the public, a signer generates a self-certified certificate containing his public key and identity information, sends it with payment to one of the CAs to apply for certifying the certificate by the CA for a period of time (six month or longer). The CA would verify the signer's information, replace the self-certified certificate with the one certified by the CA, and send it back to the signer for distribution. The trust chain described here could be extended: If A certifies B, B certifies C, and C certifies D; and the user trusts A, then the user can trust B, C and D too.

From the above discussion you can see that the key/certificate management is critical to the security of a computer.

**Question 5:** What is identity authentication?

**Question 6:** What is data validation?

**Question 7:** What is the most important task in computer security based on cryptography?

**Question 8:** What is the difference between a fingerprint and a signature of a document?

**Question 9:** What is the difference between a public key and its digital certificate?

### *9.1.2 Java Security Framework*

A computer may have multiple users. Each user has its own home folder. Let us assume that a user John has login name "john". If the computer runs Windows, John has "C:\users\john" as his home folder, which has "file://C:/users/john" as its URL. If the computer runs Linux, John has "/home/john" as his home folder, which has "file:///home/john" as its URL.

Java JDK is the tool kit for developing Java programs, and Java JRE is the tool kit supporting the execution of Java programs. A Java JDK installation normally includes a copy of Java JRE. When you install Java JDK on a Linux computer, the folder holding the JDK installation is called the Java home (we represent it with [Java home] in this document), and the folder "jre" nested inside the Java home is called the Java JRE (Java runtime environment) home (we represent it with [JRE home] in this document). If you install Java JDK on a Windows computer, by default your Java home folder is "C:\Program Files\Java\jdk1.x_y" (where x and y specify version numbers); if you also see a folder "C:\Program Files\Java\jre#" (where # is a number), your Java JRE home folder is "C:\Program Files\Java\jre#" (replacing # with the actual digit). Most Java utilities for security management are in folder "[Java home]/bin", so it is important to include this "bin" folder in the operating system "PATH" value. Most Java security policy related files are in folder "[JRE home]/lib/security".

The *security manager* is the main mechanism for Java to assign access rights to Java programs. All applets always run under the control of the Java security manager and there is no way to opt out. A Java application, Java program that runs outside of a web browser, by default doesn't run under the control of a security manager so it has full rights on the computer. If you don't completely trust this application, you should run the application under the control of the Java security manager with a Java command-line switch as shown below:

```
java     -Djava.security.manager   [Java Class Name]
```

Following the principle of separating policies from the mechanisms, the Java security manager does not hard code access rights to Java programs. When an applet or an application using Java security manager starts, Java security manager first reads file "[JRE home]/lib/security/java.security" to find where to load the java security policy files in the order specified by lines similar to

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

where the value of "policy.url.1" specifies the default security policy in file "[JRE home]/lib/security/java.policy" (SUN chose a bad name "java.home" to indicate what we call JRE home) for all users; and the value of "policy.url.2" specifies the security policy in file "[user home]/.java.policy"

(don't forget the leading period); if user name is "john", "[user home]" is "/home/john" on a Linux computer, and "C:\users\john" on a Windows computer) for a particular user. You could also insert lines like

```
policy.url.3=file:${user.home}/john.policy
```

to add the extra Java policy files with decreasing priority. A policy file loaded later could override policies specified in a policy file loaded earlier.

While you are not supposed to modify file "[JRE home]/lib/security/java.policy", you could freely modify file "[user home]/.java.policy" to modify existing policies and add new policies. The contents of Java security policy files have strict syntax requirements so it is easier to use the graphic user interface of Java utility "policytool" to review, edit, insert and delete policies.

**Question 10:** What is the home folder of *user* on the *Ubuntu* VM? How to write it in URL format?

**Question 11:** What is your home folder on your Windows PC if your login name is "john"? How to write it in URL format?

**Question 12:** What is the difference between Java JRE and Java JDK?

**Question 13:** Which folder holds the most important Java security files?

**Question 14:** Which file specifies the location of your Java security policy files?

**Question 15:** Which is your default personal Java security policy file?

**Question 16:** Which Java utility is for helping you create Java security policy files?

**Question 17:** What is the Java security manager? When is it used?

### *9.1.3 Key Management*

Java maintains public/private key pairs and digital certificates in keystore files with any file names and at any file system locations. A computer can have multiple keystore files. You can use Java utility "keytool" to create a new public/private key pair and insert it into an existing keystore file, or create a new keystore file first if necessary. A keystore file is protected by a keystore password. Each public/private key pair in the keystore is also protected by its own key password, which could be the same as the keystore password. When you create a new public/private key pair, you assign an alias or nickname to the pair for you later referring to the pair.

For more information on Java security, please visit the "Java SE Security" page at http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html.

**Question 18:** Where are public/private keys normally stored?

**Question 19:** Which Java utility helps you maintain keystores?

## 9.2 Lab Objectives

In this lab you will

1. Learn and practice how to review your current Java policies;
2. Learn and practice how to create a Java applet, observe how it is limited in accessing system resources, and create a policy to enable it to access some system resources;
3. Learn and practice how to run a Java application with Java security manager, and create a policy to enable it to access some system resources;
4. Learn and practice how to use a Java utility to create public/private key pairs and digital certificates;

## 9.3 Lab Guide

### *9.3.1 Reviewing Java Security Framework*

1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab" to change the working folder to "~/JavaSecurityLab".
4. Run "gedit GetProperties.java" to review the source code of file "GetProperties.java" as shown below:

```
class GetProperties {
  public static void main(String[] args) {
    String s;
    try {
      s = System.getProperty("os.name", "not specified");
      System.out.println("  Your operating system is: " + s);
      s = System.getProperty("java.version", "not specified");
      System.out.println("  Your Java version is: " + s);
      s = System.getProperty("user.home", "not specified");
      System.out.println("  Your user home directory is: " + s);
      s = System.getProperty("java.home", "not specified");
      System.out.println("  Your JRE installation directory is: " + s);
      s = System.getProperty("java.ext.dirs", "not specified");
      System.out.println("  Your Java extension directories are: " + s);
    } catch (Exception e) {
      System.err.println("Caught exception: " + e);
    }
  }
}
```

"os.name", "java.version", "user.home", "java.home" and "java.ext.dirs" are Java property names for OS name, Java version, user home folder, Java JRE home folder, and Java extension folders. Method "String System.getProperty(String name, String defaultValue)" returns the value of the specified Java property name, or the default value if the property name is not defined.

5. Run "javac GetProperties.java" to compile the source code into bytecode file "GetProperties.class".

6. Run "java GetProperties" to execute file "GetProperties.class". You will see the following printout:

```
Your operating system is: Linux
Your Java version is: 1.6.0_25
Your user home directory is: /home/user
Your JRE installation directory is: /home/user/jdk1.6.0_25/jre
Your Java extension directories are:
  /home/user/jdk1.6.0_25/jre/lib/ext:/usr/java/packages/lib/ext
```

7. Click menu item "Places|Home Folder" to launch the file explorer, and click to drill down to folder "/home/user/jdk1.6.0_25/jre/lib/security".
8. Right-click file "java.security" and choose "Open with Text Editor" to review the contents of the file. Scroll to find the following two lines.

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

9. Right-click file "java.policy" and choose "Open with Text Editor" to review the contents of the file. The following first "grant" statement

```
grant codeBase "file:${{java.ext.dirs}}/*" {
   permission java.security.AllPermission;
};
```

   grants all permission to all Java Jar files in Java extension folders. The second "grant" statement enables all Java programs to read selected Java properties.

   Java has two folders for extending its functions. Folder "[JRE home]/lib/ext" is for holding Jar files (zipped Java classes) used when these classes are not found in the standard Java library. The folder "[JRE home]/lib/endorsed" is for holding Jar files overriding the same named classes in the standard Java library.

10. Run "cd ~" to change working folder to be the user home folder. Run "ls -alg" to verify that there is no file ".java.policy" in the user home folder yet.

**Question 20:** What are the two major vulnerabilities of Java security framework?

**Question 21:** How should you resolve the vulnerabilities of Java security framework?

### *9.3.2 Creating Public/Private Keys and Digital Certificates*

1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab" to change the working folder to "~/JavaSecurityLab".
4. Run Java utility "keytool" as in the following line (on the same line, no line break) to create a pair of public/private keys, assign alias "PaceKey" to this pair of keys, assign key password "Seidenberg" to this air of keys, create a new keystore file "PaceKeystore" and set its keystore

password to be “Pace” if there is no such a file in the current folder yet, access keystore file “PaceKeystore” with keystore password “PaceUniversity”, and insert the new key pair in the keystore.

```
keytool -genkey -alias PaceKey -keypass Seidenberg -keystore PaceKeystore -storepass PaceUniversity
```

When prompted, enter “John Smith” as first and last names, “Seidenberg” for organizational unit, “Pace University” as organization, “New York” as city, “NY” as state, “US” as name of two-letter country code, and “y” to confirm your information. The information that you just entered interactively is called your *distinguished-name* information. Since you don’t have a keystore file “PaceKeystore” in the current folder, this file is created in the current folder. This command also stored in this keystore a pair of new public/private keys and a self-certified digital certificate that includes the public key and the distinguished-name information. This certificate will be valid for 90 days, the default validity period if you don’t specify a *–validity* option. The certificate is associated with the public/private key pair in a keystore entry referred to by the alias “PaceKey”.

```
user@ubuntu:~/JavaSecurityLab$ keytool -genkey -alias PaceKey -keypass Seidenberg -keystore PaceKeystore -storepass PaceUniversity
What is your first and last name?
  [Unknown]:  John Smith
What is the name of your organizational unit?
  [Unknown]:  Seidenberg
What is the name of your organization?
  [Unknown]:  Pace University
What is the name of your City or Locality?
  [Unknown]:  New York
What is the name of your State or Province?
  [Unknown]:  NY
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=John Smith, OU=Seidenberg, O=Pace University, L=New York, ST=NY, C=US correct?
  [no]:  y
```

If you don’t specify the keystore file name, “keytool” will use the default file “[user home]/.keystore”.

5. Run the following command to export a copy of your newly generated public key, with alias “PaceKey”, in a self-certifying digital certificate file “PaceKey.cer”.

```
keytool -export -keystore PaceKeystore -alias PaceKey -file  PaceKey.cer
```

“keytool” will ask you for the keystore password “PaceUniversity”, and then generate the new certificate file “PaceKey.cer”.

```
user@ubuntu:~/JavaSecurityLab$ keytool -export -keystore PaceKeystore -alias PaceKey -file PaceKey.cer
Enter keystore password:  PaceUniversity
Certificate stored in file <PaceKeys.cer>
```

6. Run “keytool -printcert -file PaceKey.cer” to review the contents of the certificate in file “PaceKey.cer”, and “keytool” will print out the following information:

```
   user@ubuntu:~/JavaSecurityLab$ keytool -printcert -file PaceKey.cer
   Owner: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York, ST=NY,
C=US
   Issuer: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York,
ST=NY, C=US
   Serial number: 4ca4dd45
   Valid from: Thu Sep 30 11:56:05 PDT 2010 until: Wed Dec 29 10:56:05 PST
2010
   Certificate fingerprints:
        MD5:  CA:09:3C:EC:12:D9:5D:20:E8:1E:6A:FB:88:CE:B2:18
        SHA1: F7:CD:4B:18:66:55:C8:2B:BB:9E:AD:92:A8:DF:54:9A:F7:96:93:E7
        Signature algorithm name: SHA1withDSA
        Version: 3
```

Note the certificate fingerprints in MD5 and SHA1 formats. If you send this certificate to a friend, you could both run this command to find the fingerprints of this certificate, and compare them to see whether the certificate (signed public key) has been compromised in transit.

7. If you were doing the real work, you may now contact one of the CAs, send it your self-signed certificate file “PaceKey.cer” with your payment, and the CA will verify your distinguished-name information and send back to you your digital certificate signed by the CA. The CA certified certificate is the one that you are supposed to distribute to your partners. In this tutorial we skip this step and just distribute the self-certified certificate in file “PaceKey.cer”.
8. In the real situation, your digital certificate should be distributed to your partners, and imported to your partners’ keystores. In this tutorial you will also act as your partner. You will import the newly generated digital certificate in file “PaceKey.cer” into a new keystore file named “receiverKeystore” in the current folder with the following command

```
   keytool -import -alias Pace -file PaceKey.cer -keystore
receiverKeystore
```

Upon being asked for keystore password, enter “123456”. When being asked whether you trust this certificate, respond with “y” for yes.

```
   user@ubuntu:~/JavaSecurityLab$ keytool -import -alias Pace -file
PaceKey.cer -keystore receiverKeystore
   Enter keystore password:  123456
   Re-enter new password:    123456
   Owner: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York, ST=NY,
C=US
   Issuer: CN=John Smith, OU=Seidenberg, O=Pace University, L=New York,
ST=NY, C=US
   Serial number: 4ca4dd45
   Valid from: Thu Sep 30 11:56:05 PDT 2010 until: Wed Dec 29 10:56:05 PST
2010
   Certificate fingerprints:
        MD5:  CA:09:3C:EC:12:D9:5D:20:E8:1E:6A:FB:88:CE:B2:18
        SHA1: F7:CD:4B:18:66:55:C8:2B:BB:9E:AD:92:A8:DF:54:9A:F7:96:93:E7
        Signature algorithm name: SHA1withDSA
        Version: 3
   Trust this certificate? [no]:  y
   Certificate was added to keystore
```

**Question 22:** What is the more secure way to run *keytool* to generate the public/private keys?

### *9.3.3 Ensuring Data Confidentiality with Cryptography*

Protecting confidential data is a very important task of computer security. Most of the labs in this tutorial focus on identity authentication and data validation and they don't encode/decode the data files. In this lab you will try out Prof. Lixin Tao's implementation of the *Simplified DES* algorithm described in file "JavaSecurityLab/cipher-des/C-SDES.pdf". This implementation aims at helping students understand the algorithm, which is not the objective of this lab. With the hands-on experience with applying the DES cipher you could integrate it into the latter labs to add the data confidentiality component into them. DES is a very fundamental algorithm in cryptography. If you are taking a course in which cryptography is a topic, make sure your instructor explains this algorithm and you understand its design and my implementation. I have used extra comments and clear code design to help you read my DES implementation. But not all students working on this lab needs to read my source code.

1. Launch the Ubuntu VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab/cipher-des" to change work folder to "~/JavaSecurityLab/cipher-des".
4. Run "javac S_DES_File.java" to compile all the four Java source code files into their corresponding bytecode files.
5. Run "java S_DES_File" to find out how to run this program.

```
usage: java S_DES_File inputFile outputFile key-bits [decode]
```

   The program takes one input file and one output file. For encoding with DES, the input file is the data file, and the output file is its encoded version. For decoding, the input file is the encoded file, and the output file is the decoded data file. In either case you need to provide "key-bits", the secret key in form of 1-10 binary bits (the professional implementation will use much longer keys; short keys allow you to trace the DES algorithm if you turn on the debugging mode of the program). Make sure that you use the same key to encode and decode a file. The "decode" switch is needed only when you decode a file.

6. Run "java S_DES_File GettysburgAddress.txt secret 1010101010" to use the DES algorithm to encode file "GettysburgAddress.txt" into a binary file "secret" with the secret key 1010101010.
7. Run "more secret" to confirm that you could not figure out the contents of file "secret".
8. Run "java S_DES_File secret GettysburgAddress2.txt 1010101010 decode" to decode file "secret" into plain file "GettysburgAddress2.txt" with the same secret key.
9. Run "more GettysburgAddress2.txt" to confirm that you have recovered the original file contents. Therefore your encode/decode process works correctly.

You can use this program to encode/decode any file of any size no matter it is a binary file or a text file.

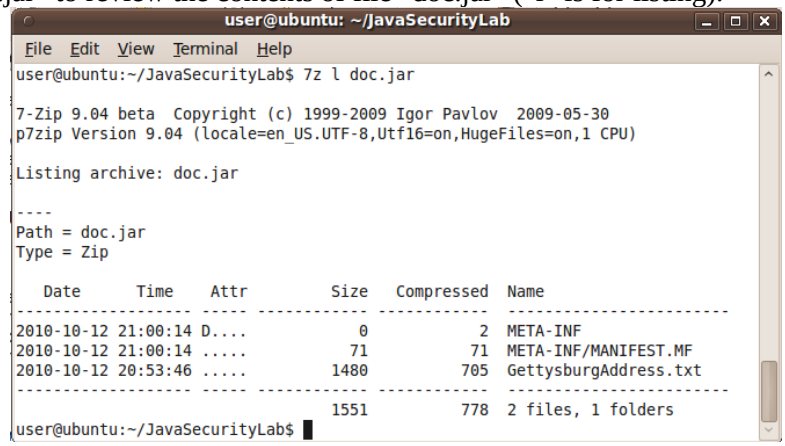
### *9.3.4 Securing File Exchange with Java Security Utilities*

In this lab you study how to use the public/private keys and the digital certificate in the last lab to securely send a document. The receiver of the document should be able to verify that the document was from the right person (identity authentication), and the document has not been modified in transit (data validation).

First let us recall what you have done in your second lab in this tutorial. You have created a pair of public/private keys in keystore file "~/JavaSecurityLab/PaceKeystore", assigned alias "PaceKey" to this

pair of keys, exported the self-signed digital certificate for the public key, imported the digital certificate for the public key into another keystore file "~/JavaSecurityLab/receiverKeystore", and assigned alias "Pace" to this certificate. In this lab the imaginary document sender will use the private key with alias "PaceKey" in keystore file "~/JavaSecurityLab/PaceKeystore" to sign the Jar file of a sample document, and the imaginary document receiver will perform identity authentication and data validation for the received document using the public key with alias "Pace" in keystore file "~/JavaSecurityLab/receiverKeystore".

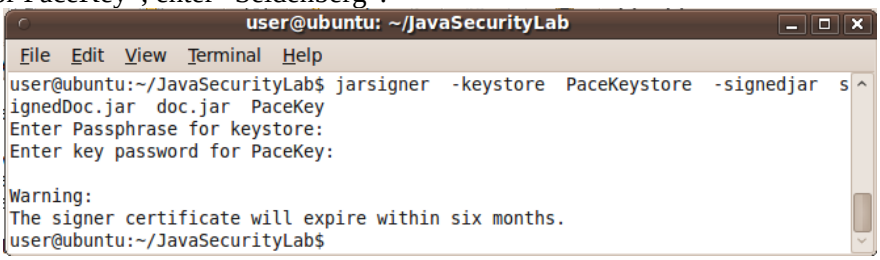
1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab" to change work folder to "~/JavaSecurityLab".
4. Run "more GettysburgAddress.txt" to review its contents. You use file "GettysburgAddress.txt" as example of a secret document for file exchange.
5. Run "jar cvf doc.jar GettysburgAddress.txt" to generate a Jar file "doc.jar" for transmission.
6. Run "7z l doc.jar" to review the contents of file "doc.jar" ("l" is for listing).




7. Now run the following command to sign file "doc.jar" with the private key with alias "PaceKey" in keystore file "PaceKeystore":

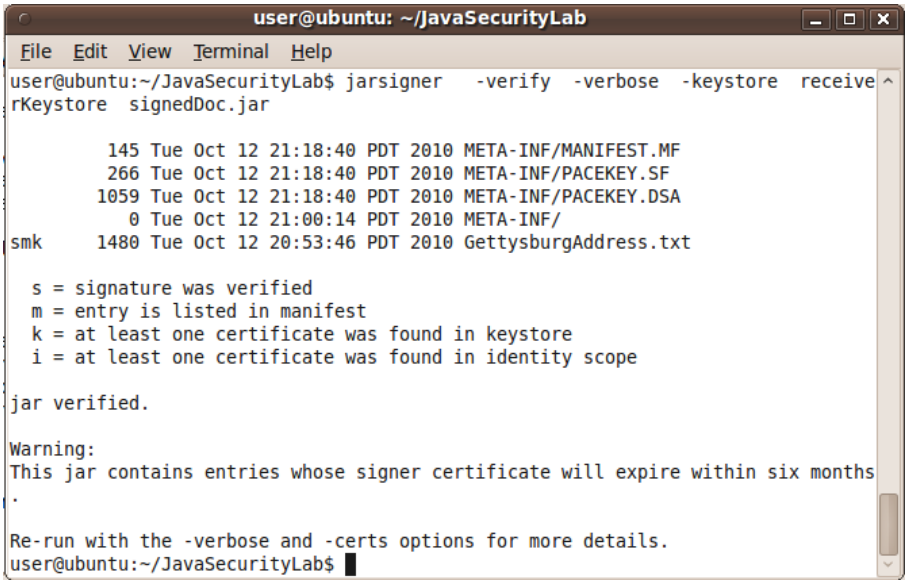
```
jarsigner -keystore PaceKeystore -signedjar signedDoc.jar doc.jar PaceKey
```

When asked for "passphrase for keystore", enter "PaceUniversity". When asked for "key password for PaceKey", enter "Seidenberg".




8. Now you will function as the document receiver. You just received file "signedDoc.jar", and you have the digital certificate for the document signer in your keystore file "receiverKeystore" (you created them in the previous lab). Run the following command to authenticate the signer and make sure the Jar file has not been tampered with:

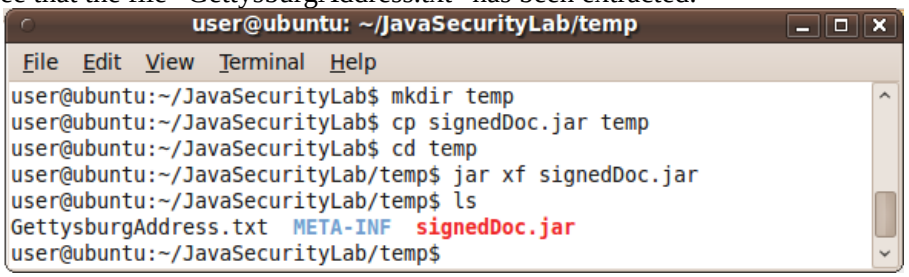
```
jarsigner -verify -verbose -keystore receiverKeystore signedDoc.jar
```




The message shows that the Jar file was signed by the right person and the jar file has not been tampered with.

9. Run the following commands to retrieve file “GettysburgAddress.txt” from file “signedDoc.jar”
    a. “mkdir temp” to create a new folder “temp”;
    b. “cp signedDoc.jar temp” to copy file “signedDoc.jar” into folder temp;
    c. “cd temp” to change work folder to “temp”;
    d. “jar xf signedDoc.jar” to extract the contents of file “signedDoc.jar”;
    e. “ls” to list the files in folder “temp”.

You will see that the file “GettysburgAddress.txt” has been extracted.




This concludes your lab for securely exchanging files.

### *9.3.5 Granting Special Rights to Applets Based on Code Location*

1. Launch the *Ubuntu* VM with username “user” and password 123456.
2. Start a terminal window in home folder ~ with menu item “Applications|Accessories|Terminal”.
3. Run “cd ~/JavaSecurityLab/applet1” to change working folder to “~/JavaSecurityLab/applet1”.
4. Run “gedit WriteFile.java” to review the contents of the applet source file.

```
import java.awt.*;
import java.applet.*;
import java.io.*;

public class WriteFile extends Applet {
  public void paint(Graphics g) {
```

```
      try {
        String fileName = System.getProperty("user.home") +
                    System.getProperty("file.separator") + //   / or \
                    "data.txt";
        File f = new File(fileName);
        PrintWriter output = new PrintWriter(new FileWriter(f), true); // auto-flush
        output.println(new java.util.Date() + ": Pace University Java Tutorial");
        g.drawString("Data were successfully written to file \"" + fileName +
                     "\"", 10, 10);
      }
      catch (SecurityException e) {
        g.drawString("WriteFile: security exception is caught", 10, 10);
        e.printStackTrace();  // print error messages to terminal window
                              // for debugging
      }
      catch (IOException ioe) {
        g.drawString("WriteFile: i/o exception is caught", 10, 10);
      }
    }
  }
```

An applet is a Java program that is embedded in an HTML file and run inside a web browser. An applet class always extends the base class “Applet”. Its method “paint(Graphics g)” executes every time the web page containing it is reloaded. This applet first opens the file “[user home]/data.txt” (creating the file first if there is no such a file yet), and replaces its contents with the current time followed by “: Pace University Java Tutorial”. If for some reason the file cannot be opened or write fails, an exception will be thrown and displayed. Method call “System.getProperty("file.separator")” finds out whether the file separator in a file path is “/” or “\” so this applet could run on either Windows or Linux. Method call “g.drawString("string", 10, 10)” displays “string” in the applet area of the web browser 10 pixels to the right and 10 pixels down relative to the origin of the applet area specified with the “width” and “height” attributes of the <applet> element in an HTML file.

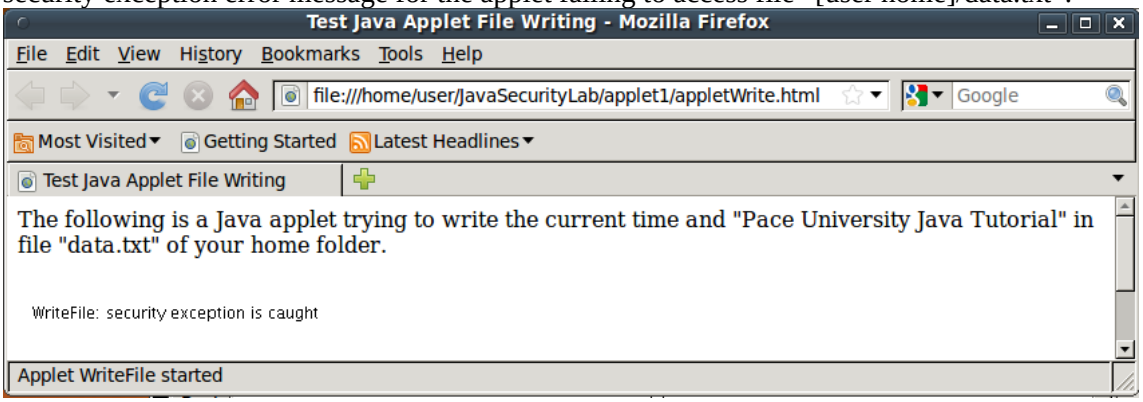
5. Shutdown the gedit editor.
6. Run “javac  WriteFile.java” to compile the source code into bytecode file “WriteFile.class”.
7. Run “gedit appletWrite.html” to review the contents of file “appletWrite.html”.

```
<html>
<head>
<title>Test Java Applet File Writing</title>
</head>

<body>
<p>The following is a Java applet trying to write the current time and
"Pace University Java Tutorial" in file "data.txt" of your home
folder.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100>
</applet>
</body>
</html>
```

An applet is embedded in an HTML file with the “applet” tag. The applet element can use attribute “code” to specify the applet bytecode file path relative to the folder containing the current HTML file, and the “width” and “height” attributes to specify the display area of the applet in a web browser.

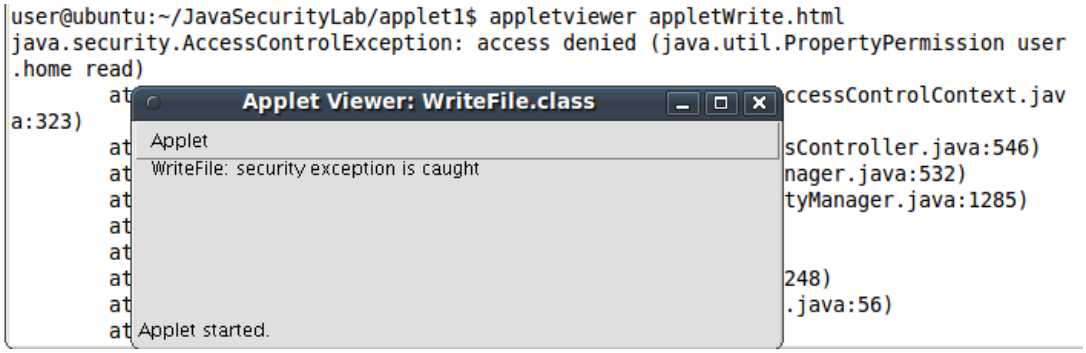
8. Shutdown the gedit editor.

9. Double-click on file “appletWrite.html” to display it in a web browser, and you will see the security exception error message for the applet failing to access file “[user home]/data.txt”:

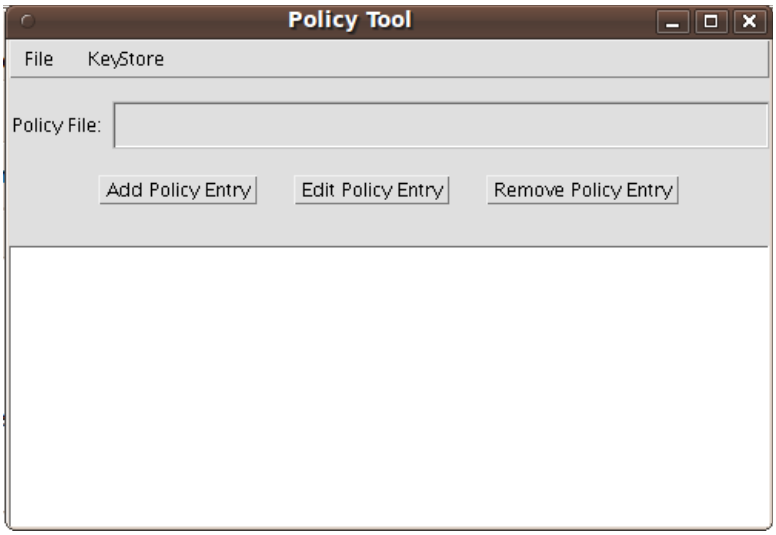



The applets always run under the Java security manager, which disables applets to access local files and many other resources. Since you didn’t launch the web page with a terminal window, you cannot read the exception message printed by the “e.printStackTrace();” statement of the applet.

10. Now you will launch the web page with Java utility “appletviewer” so you could read more information about exceptions. Shutdown the web browser. Run “appletviewer appletWrite.html” to see the following applet window running the applet:

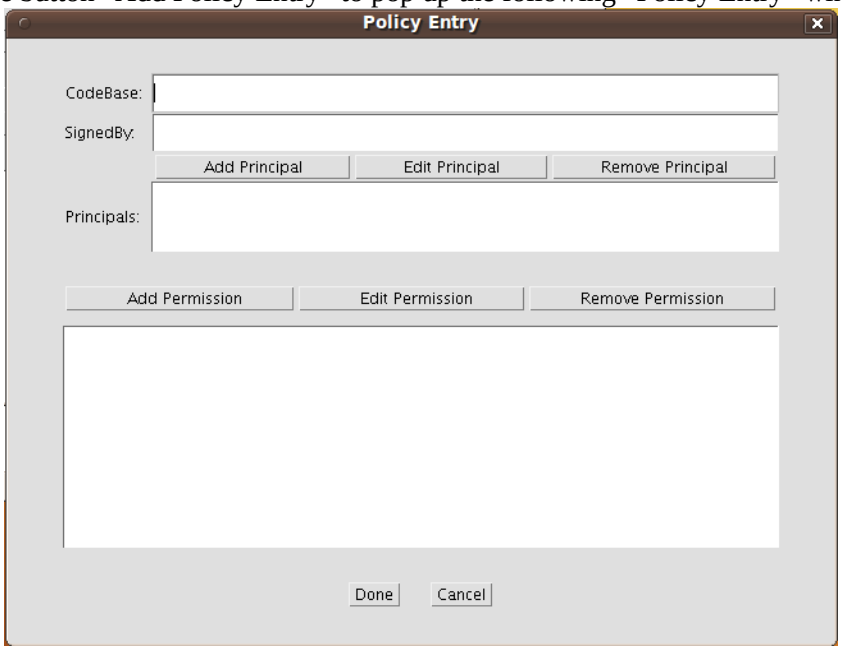



11. From the terminal window you could see a long list of error messages. You should focus on the first few lines, which told us that the applet has problem in accessing Java property “user.home”. Actually the exception only pointed out the first security problem. The second one is that the applet could not access local file “[user home]/data.txt”.
12. Now you will grant special rights for this applet to complete its task. Run Java utility “policytool” to launch the following *Policy Tool* window.

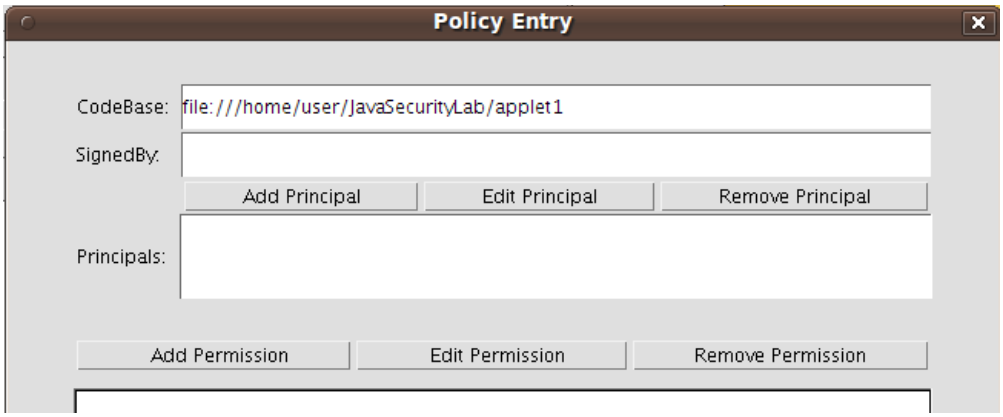



If you see contents in the "Policy File" textbox, it just means you already have a Java policy file for your account, and you can still continue in this lab.

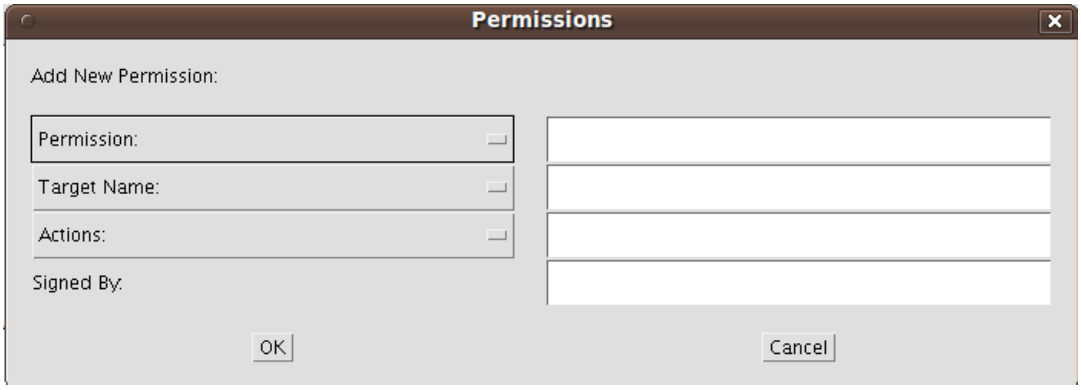
13. Click on the button "Add Policy Entry" to pop up the following "Policy Entry" window.



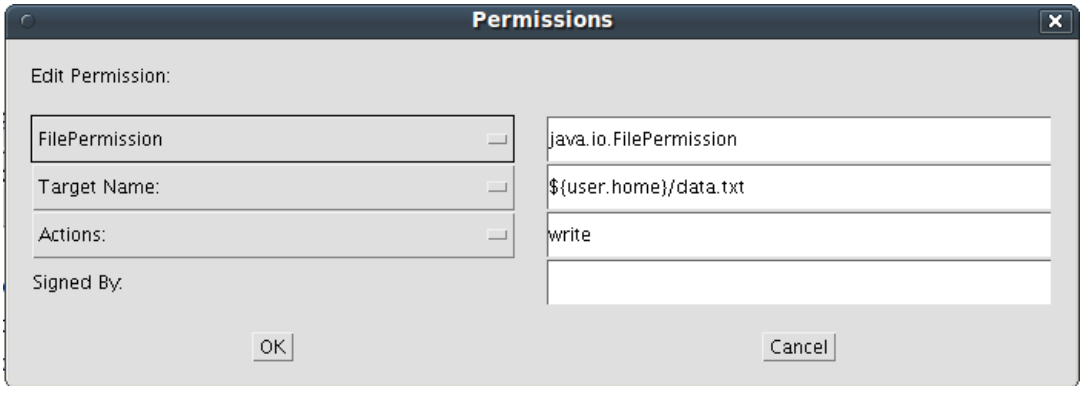

14. You can use the *CodeBase* textbox to specify special rights to all Java code in a folder (CodeBase, specified with an absolute path) or its subfolders (terminating the *CodeBase* path with "/-"), or use the *SignedBy* textbox to specify special rights to all Java code signed by a particular private key where the *SignedBy* textbox holds the alias for the corresponding digital certificate stored in a keystore, or use both of the *CodeBase* and the *SignedBy* textboxes to specify special rights only to those Java code signed by a specific private key and located in a specific folder or its subfolders. In this exercise you only use the *CodeBase* textbox.
15. In the *CodeBase* textbox type "file:///home/user/JavaSecurityLab/applet1", the URL for folder "~/JavaSecurityLab/applet1". This will allow you to specify rights for all code in this folder. (If you add "/-" to the end of the URL, leading to "file:///home/user/JavaSecurityLab/applet1/-", code in folder "applet1" or its subfolders have the same special rights.)



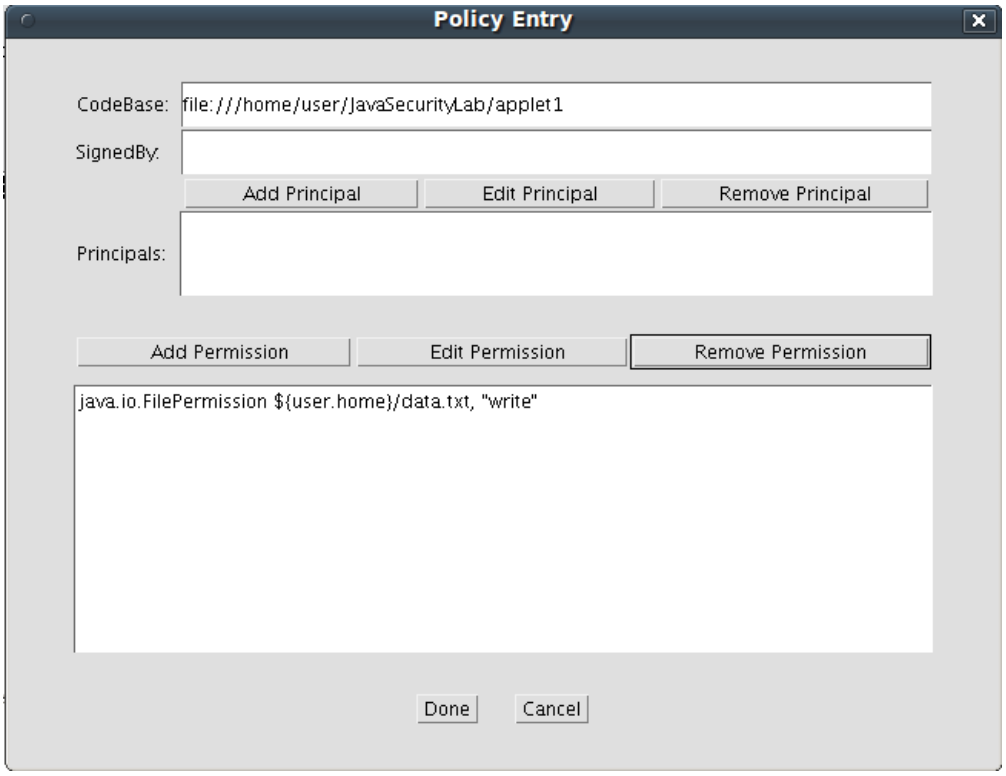

16. Click button “Add Permission” to get the following “Permissions” window.



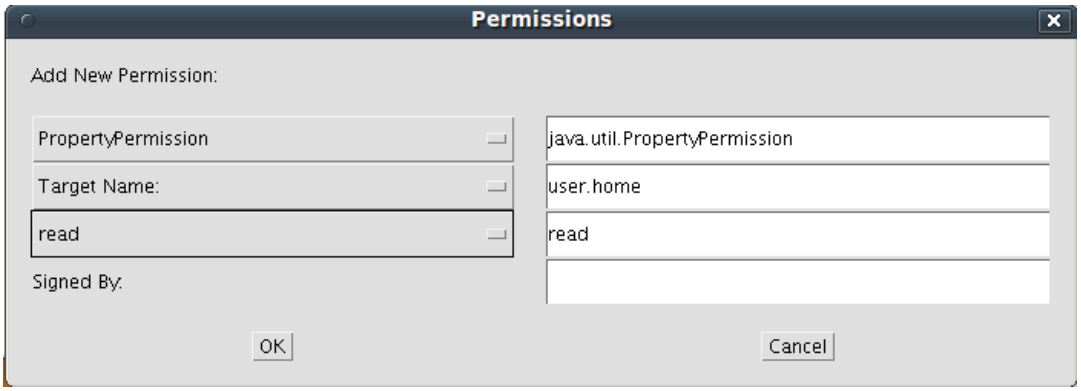

17. Click the “Permission” drop-down list and select “FilePermission”. Type file name “${user.home}/data.txt” in the textbox to the right of the “Target Name” drop-down list. Click the “Actions” drop-down list and select “write”.



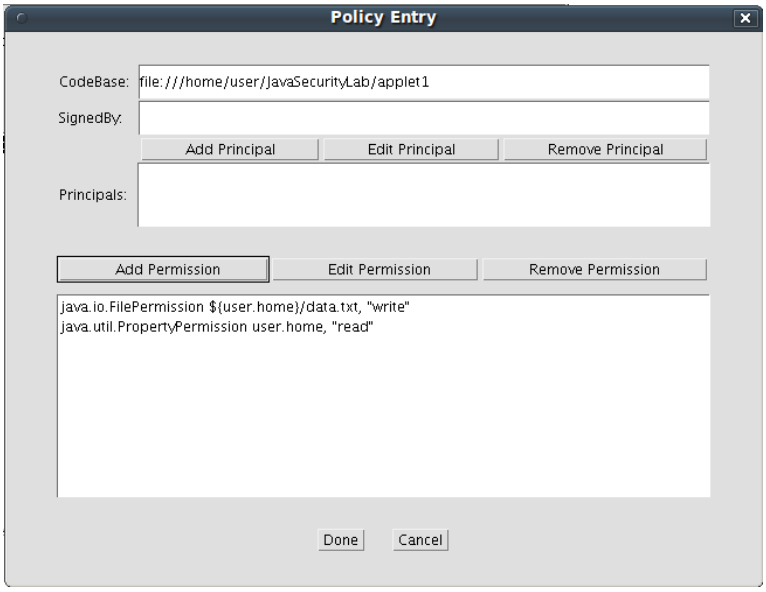

18. Click the “OK” button. The new permission displays in a line in the *Policy Entry* window.



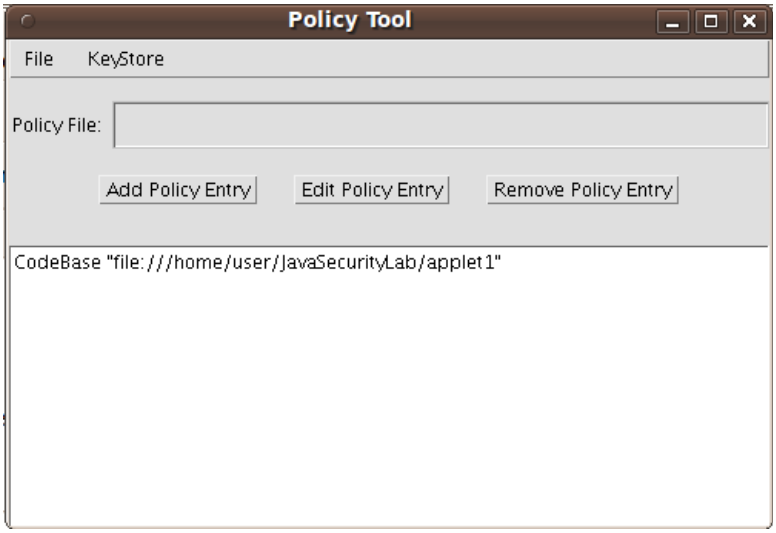

19. Now you will add the second policy to allow the applet to read the value of Java property "user.home". Click the "Add Permission" button again to launch the *Permissions* window. Choose "PropertyPermission" for "Permission", type "user.home" for "Target name", choose "read" for "Actions", and then click on the OK button to close the *Permissions* window.




20. Now you see the following *Policy Entry* window:

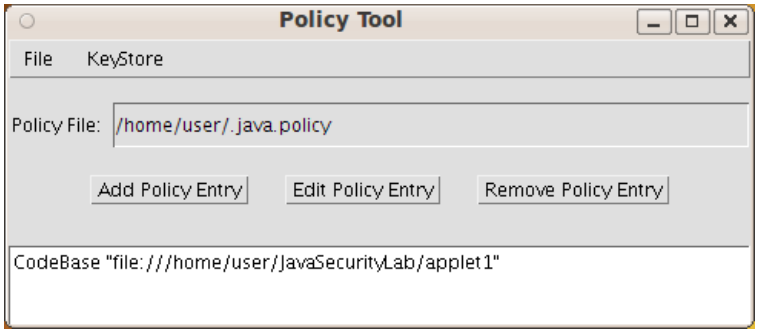



21. Click the “Done” button to return to the “Policy Tool” window.




22. Click menu item “File|Save As”, enter “/home/user/” followed by the *Enter* key in the “Enter path or folder name” textbox, and enter “.java.policy” in the “Enter file name” textbox.

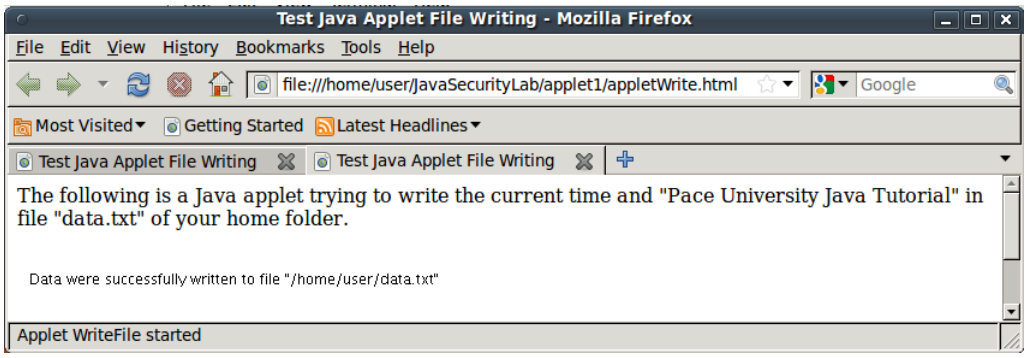

23. Click the “Save” button to close the “Save As” window.



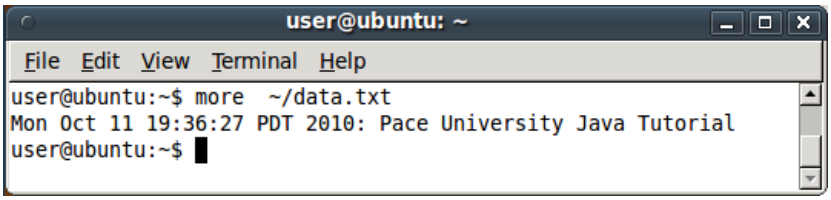

24. Click menu item “File|Exit” to shut down the *Policy Tool* window.
25. Run “more ~/.java.policy” to review the contents of the Java policy file:

```
grant codeBase "file:///home/user/JavaSecurityLab/applet1" {
  permission java.io.FilePermission "${user.home}/data.txt", "write";
  permission java.util.PropertyPermission "user.home", "read";
};
```

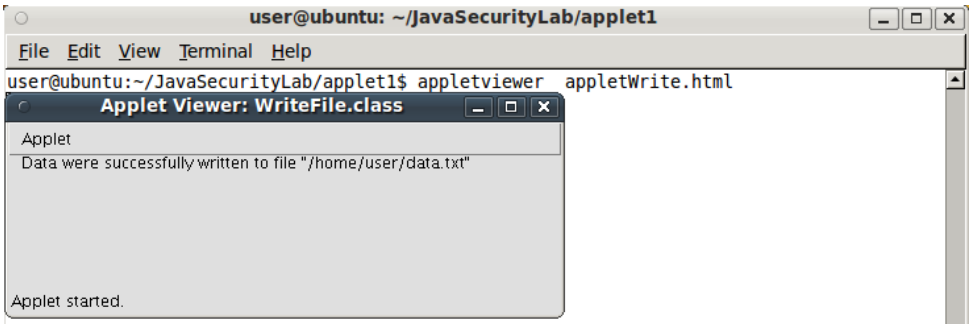
    If you know the syntax for granting rights to Java code, you could directly edit the policy file. The utility “policytool” is optional to make the task of granting rights easier.
26. In folder “/home/user/JavaSecurityLab/applet1”, double click file “appletWrite.html” and display it in a web browser, and this time you see the applet message “Data were successfully written to file “/home/user/data.txt” ”.




27. Run “more ~/data.txt” to review the contents of file “data.txt”.




28. Now you will launch the web page with Java utility “appletviewer” again. Run “appletviewer appletWrite.html” to see the following applet window running the applet successfully:




Now you have successfully granted security rights to an applet located at a specific location so it could write to a local file.

**Question 23:** Should you use relative path or absolute path to specify *CodeBase*?

**Question 24:** Should you use relative path or absolute path to specify a file as the value of the *Target Name* of the *Permissions* window?

**Question 25:** What is the meaning of “${user.home}”?

### *9.3.6 Granting Special Rights to Applets Based on Code Signing*

In the last lab you learned how to grant special rights to applets based on code location. In this lab you learn how to do the same thing based on which private key has signed the code. You will learn how to make a Jar file of Java classes, sign the Jar file with a private key, write an HTML file and use the Jar file to provide code for an applet, and grant file write to any Java code signed by the particular private key.

First let us recall what you have done in your second lab in this tutorial. You have created a pair of private/public keys in keystore file "~/JavaSecurityLab/PaceKeystore", assigned alias "PaceKey" to this pair of keys, exported the self-signed digital certificate for the public key, imported the digital certificate for the public key into another keystore file "~/JavaSecurityLab/receiverKeystore", and assigned alias "Pace" to this certificate in the latter keystore. In this lab the imaginary developer will use the private key with alias "PaceKey" in keystore file "~/JavaSecurityLab/PaceKeystore" to sign the Jar file of the applet class, and the imaginary user will grant special rights to all programs signed by the private key whose corresponding public key is in a digital certificate with alias "Pace" in keystore file "~/JavaSecurityLab/receiverKeystore".

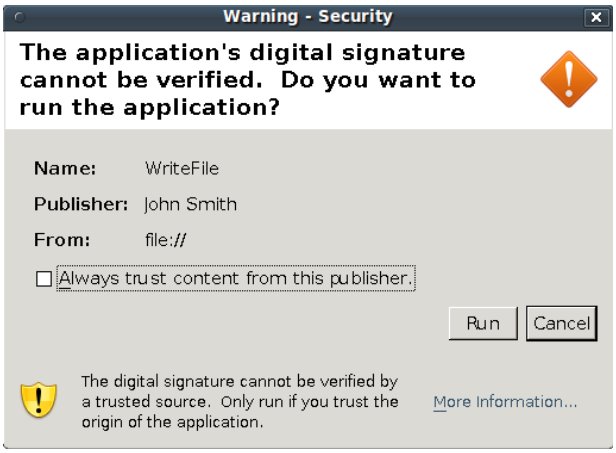
1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab/applet2" to change working folder to "~/JavaSecurityLab/applet2".
4. Run "gedit WriteFile.java" to review the contents of the applet source file. This file is the same as the one that you used in the last lab.
5. Run "javac WriteFile.java" to compile the source code into bytecode file "WriteFile.class".
6. Run "jar cvf WriteFile.jar WriteFile.class" to create a Jar file "WriteFile.jar". A Jar file is basically a zip file containing a set of Java classes and resources. The Jar format is easier for deployment of a set of related Java classes on a computer.
7. Run "7z l WriteFile.jar" to review the contents of file "WriteFile.jar" (the 7z command-line switch "l" means listing) .
8. Now assuming you were the developer, and you use your private key with alias "PaceKey" to sign the Jar file. Run the following command

```
jarsigner -keystore ../PaceKeystore -signedjar signedWriteFile.jar
WriteFile.jar PaceKey
```

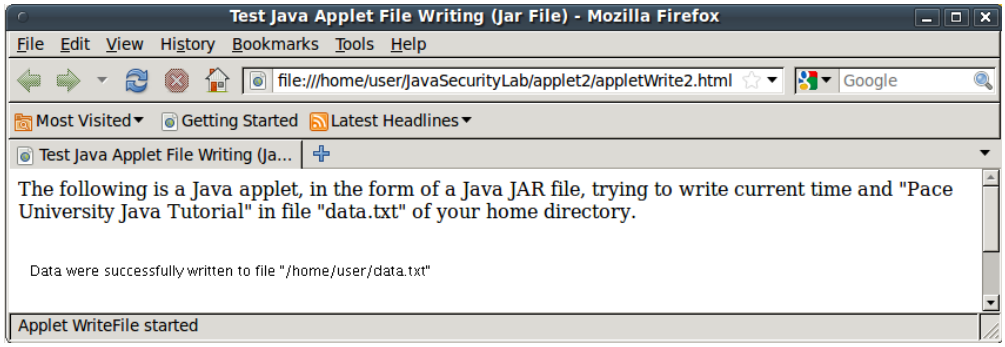
   on the same line to use the private key with alias "PaceKey" in keystore file "../PaceKeystore" to sign the jar file "WriteFile.jar" and write the output in file "signedWriteFile.jar". When asked for keystore passphrase (password), enter "PaceUniversity". When asked for key passphrase, enter "Seidenberg". A new file "signedWriteFile.jar" will be generated.
9. Run "7z l signedWriteFile.jar" and note the new files in the Jar file for the digital signature of the original Jar file "WriteFile.jar".
10. Run "gedit appletWrite2.html" to review its contents:

```
<html>
<head>
<title>Test Java Applet File Writing (Jar File)</title>
</head>
<body>
<p>The following is a Java applet, in the form of a Java JAR file, trying
to write the current time and "Pace University Java Tutorial" in file
"data.txt" of your home directory.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100
        archive="signedWriteFile.jar">
</applet>
</body>
</html>
```

    If the applet class is contained in a Jar file, you can use the "archive" attribute of the "applet" tag to specify the Jar file.
11. In the file explorer, double-click file "appletWrite2.html" to display it in the web browser. A security warning window will pop up:

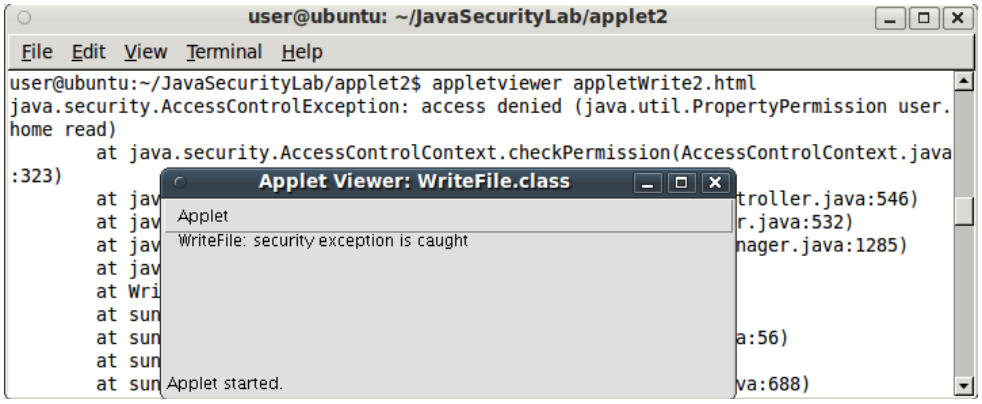



12. Click the "Run" button to tell your computer that you trust this applet signed by John Smith, and the web browser will write data to file "~/data.txt" successfully.

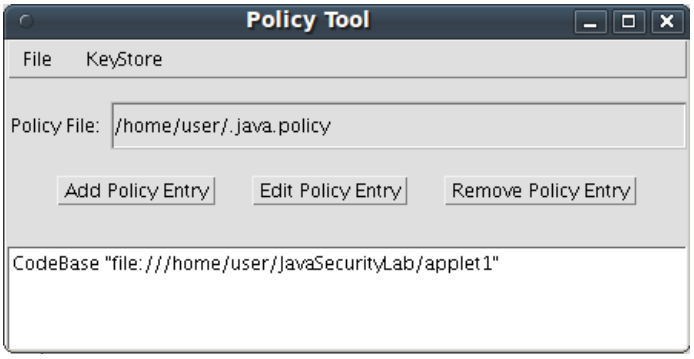



If you shut down the web browser and repeat this step, you will be warned against this applet again. If you check the "Always trust content from this publisher" checkbox, your computer will remember that you always trust programs signed by the private key of John Smith and it will not warn you again before it runs programs signed by the same private key. But for this lab we will not check this checkbox. Declaring that you trust programs signed by a particular private key this way is too dangerous. You want to use the Java security policy to define which rights that you want to grant programs signed by this particular private key.

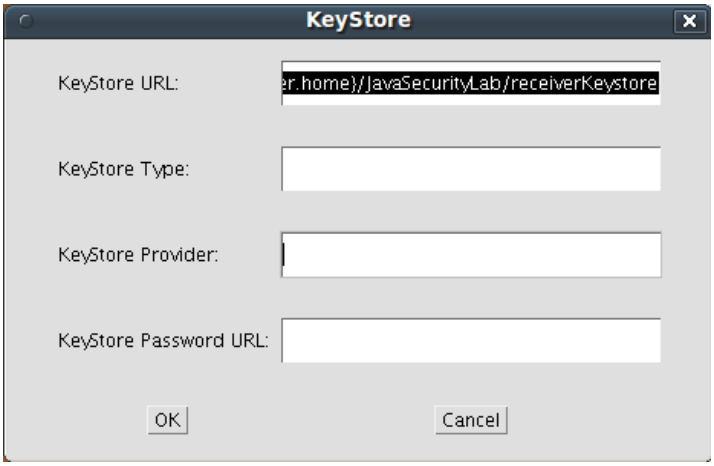
13. Shut down the web browser.
14. Run "appletviewer appletWrite2.html" and you get the security warnings similar to what you got in the last exercise:



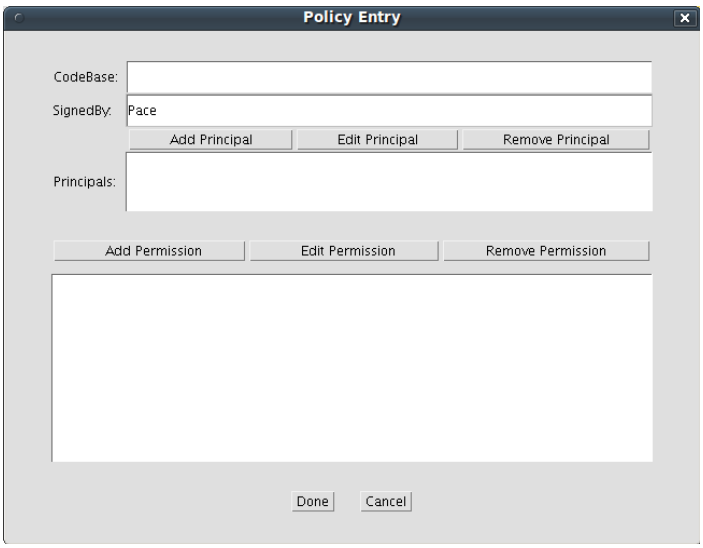
15. Now you are ready to grant access rights to all programs signed by this particular private key. Run “policytool” to display the following window.



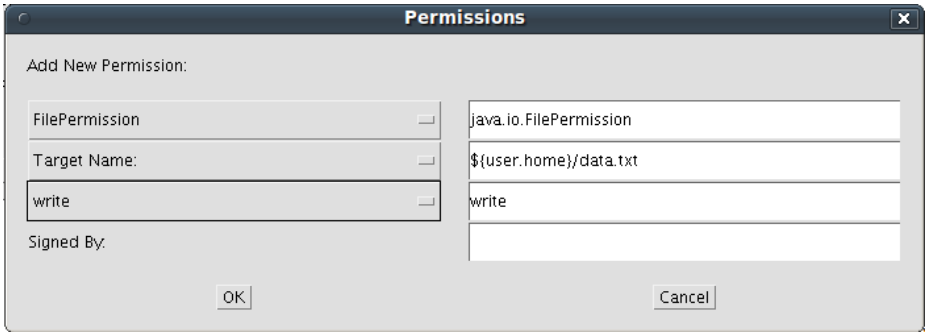

16. You first need to assign a keystore file for your policy file. Click menu item “KeyStore|Edit” to pop up the “KeyStore” window, and enter “${user.home}/JavaSecurityLab/receiverKeystore” in the “KeyStore URL” textbox (the screen capture only shows part of this value; you can use Ctr-v to paste the value from this guide).



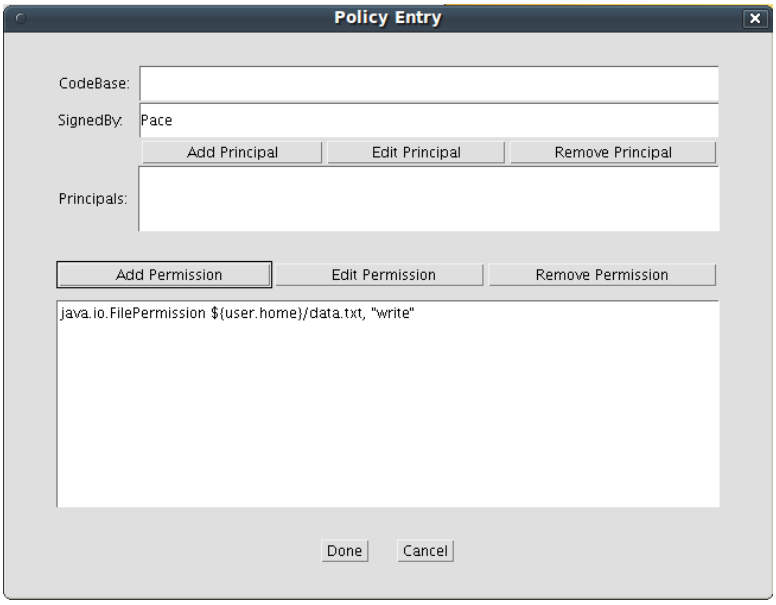

17. Click the “OK” button to close the “KeyStore” window.
18. Click the “Add Policy Entry” button to launch the “Policy Entry” window. Enter “Pace” in the “SignedBy” textbox.




19. Click the "Add Permission" button to pop up the "Permissions" window. Choose "FilePermission" for "Permission", enter "${user.home}/data.txt" for "Target Name", and choose "write" for "Actions".

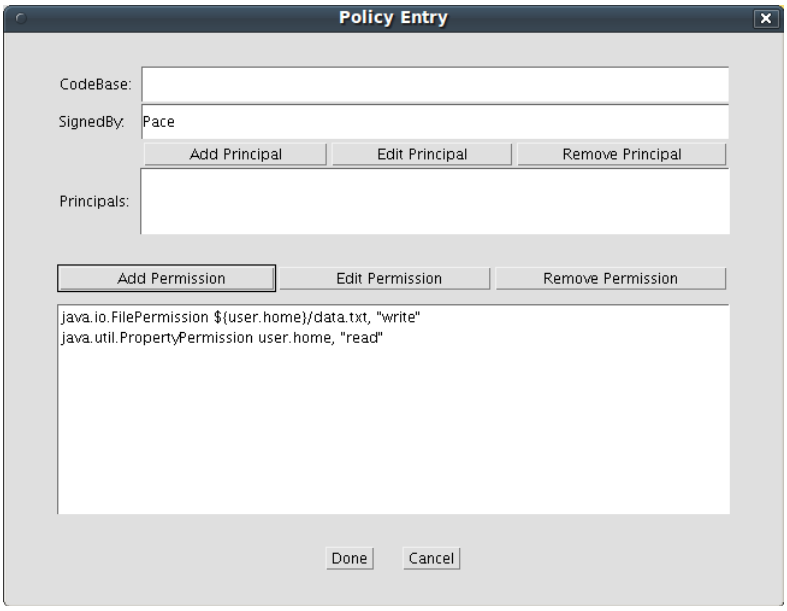



20. Click on the "OK" button to close the window.

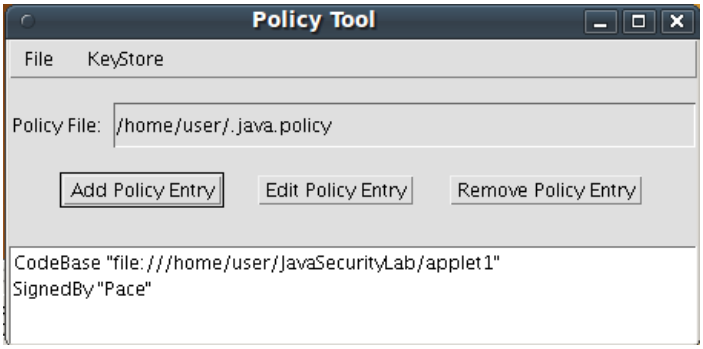



29. Click the "Add Permission" button again to pop up the "Permissions" window again. Choose "PropertyPermission" for "Permission", type "user.home" for "Target name", choose "read" for "Actions", and then click the *OK* button to close the "Permissions" window.


21. Now you see the following *Policy Entry* window:



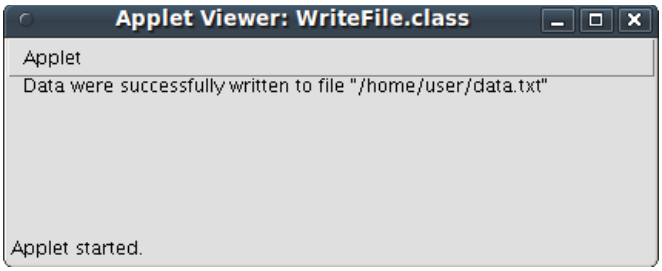

22. Click the “Done” button to return to the “Policy Tool” window.




23. Click menu item “File|Save” to save the new policy, and click menu item “File|Exit” to close the “Policy Tool” window.
24. Run “more ~/.java.policy” to review the current contents of the policy file.

```
keystore "/home/user/JavaSecurityLab/receiverKeystore", "jks";

grant codeBase "file:///home/user/JavaSecurityLab/applet1" {
  permission java.io.FilePermission "${user.home}/data.txt", "write";
  permission java.util.PropertyPermission "user.home", "read";
};

grant signedBy "Pace" {
  permission java.io.FilePermission "${user.home}/data.txt", "write";
  permission java.util.PropertyPermission "user.home", "read";
};
```

25. Now run “appletviewer appletWrite2.html” again and the applet will successfully write data in file “~/data.txt”, as shown by the screen capture below.




This concludes the lab for granting security rights based on code signing. You are encouraged to try to use both CodeBase and SignedBy to grant security rights to signed programs at specific file system locations.

**Question 26:** What is the implication if you check "Always trust content from this publisher" in a security warning window when you try to display a web page?

**Question 27:** Why "appletviewer" is a preferred tool for studying and developing web applications containing signed applets?

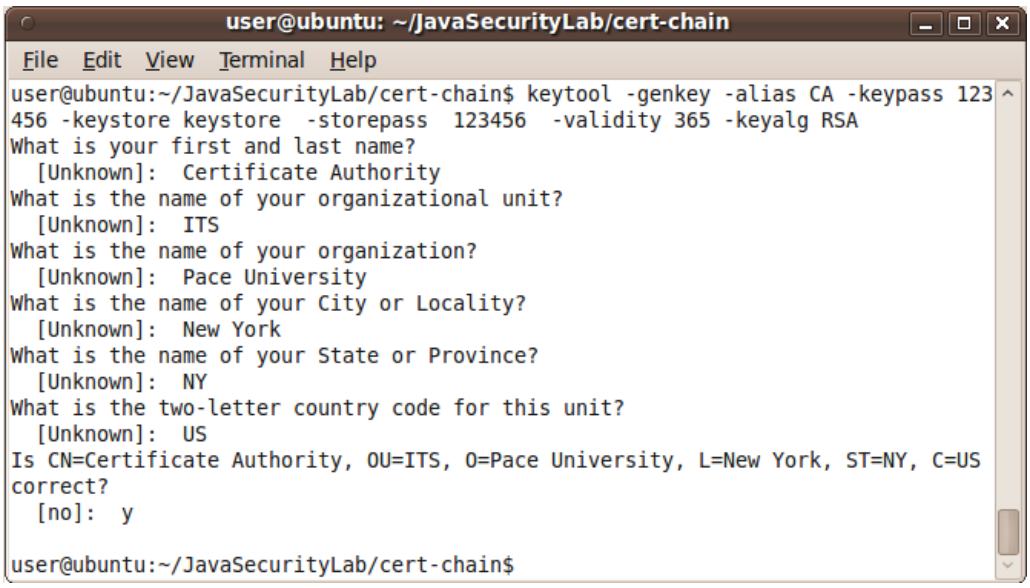
### *9.3.7 Creating a Certificate Chain to Implement a Trust Chain*

In this lab you will work as a CA (Certificate Authority) to sign another certificate SchoolKey. You will show that if your computer trusts code signed by the CA, it will also trust code signed by SchoolKey. Therefore the certificate chain supports a trust chain. You will use a Java program "SignCertificate.java" to sign a certificate with the private key associated with another certificate. Due to the complexity we will not explain this program and just use it as a tool.

1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab/cert-chain" to change the work folder to "~/JavaSecurityLab/cert-chain".
4. Run "javac SignCertificate.java" to compile the program into bytecode file "SignCertificate.class". You will see 39 warnings for the compilation, but it is harmless.
5. Run "more WriteFile.java" and "more appletWrite2.html" to review the contents of the two files. They are the same ones that you used in the last exercise.
6. Run "javac WriteFile.java" to compile the program into bytecode file "WriteFile.class". This is the same class that you used in the previous exercise.
7. Run "jar cvf WriteFile.jar WriteFile.class" to create the Jar file, as you did in the last exercise.
8. Now you are going to function as the CA (certificate authority) to create the CA's public/private key pair and its corresponding self-certified certificate in a new keystore with name "keystore" in the current folder. Run the following command on the same line:

```
   keytool -genkey -alias CA -keypass 123456 -keystore keystore  -storepass  123456  -
validity 365 -keyalg RSA
```

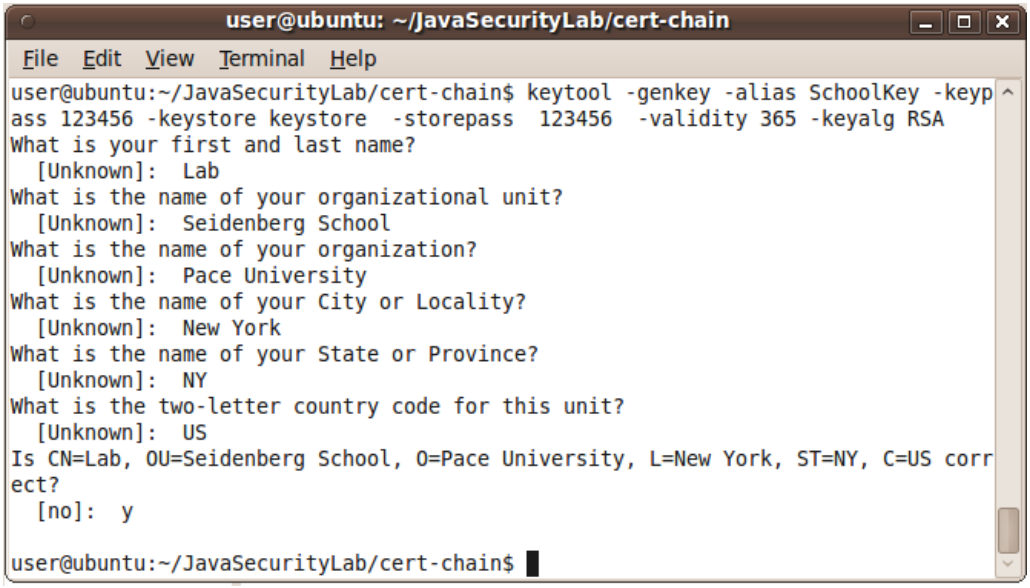
   Argument "-validity 365" specifies that the certificate would be valid for 365 days. Argument "-keyalg RSA" specifies that the key generation algorithm would be RSA. Here you use trivial password 123456 to ease your typing. You would type "Certificate Authority" for the first and last names, "ITS" for organizational unit, "Pace University" for organization, "New York" for city, "NY" for state, and "US" for country code, and "y" to confirm your input.



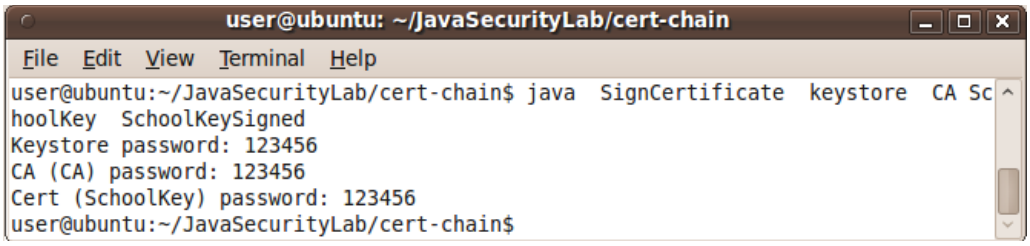

9. Now you will function as an IT staff of Seidenberg School of Pace University to create the master public/private key pair and certificate for the School in the same keystore. You will associate the keys and certificate with alias "SchoolKey". Run the following command:

```
   keytool -genkey -alias SchoolKey -keypass 123456 -keystore keystore   -storepass
123456  -validity 365 -keyalg RSA
```

You would type "Lab" for the first and last names, "Seidenberg School" for organizational unit, "Pace University" for organization, "New York" for city, "NY" for state, and "US" for country code.




10. Run "java SignCertificate keystore CA SchoolKey SchoolKeySigned" to use the private key with alias "CA" to sign the certificate with alias "SchoolKey", and associate alias "SchoolKeySigned" with the signed certificate in the keystore. Enter 123456 for all the three required passwords.



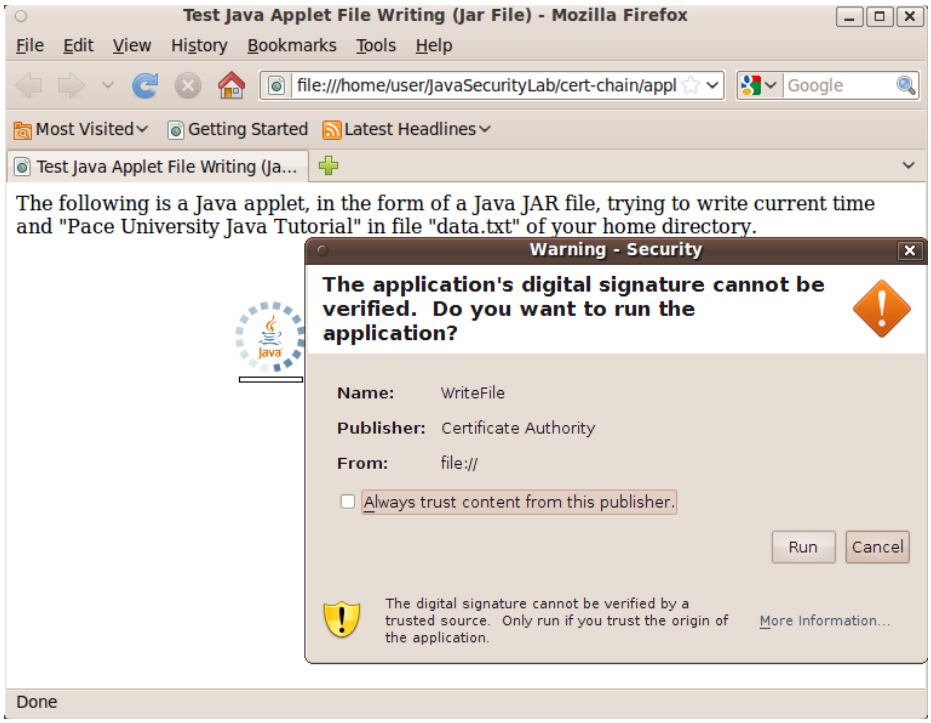

11. Run “keytool -export -alias SchoolKeySigned -keystore keystore -file SchoolKeySigned.cert” to export the certificate associated with “SchoolKeySigned” into file “SchoolKeySigned.cert”. Upon request enter 123456 for keystore password.
12. Run “keytool -import -alias SchoolKey -keystore keystore -file SchoolKeySigned.cert” to import the certificate in file “SchoolKeySigned.cert” back in the keystore entry associated with alias “SchoolKey”. Upon request enter 123456 for keystore password.
13. Run “keytool -list -v -keystore keystore” to review the contents of the keystore. Upon request enter 123456 as keystore password. You will see the following output:

```
   user@ubuntu:~/JavaSecurityLab/cert-chain$ keytool -list -v -keystore keystore
   Enter keystore password: 123456

   Keystore type: JKS
   Keystore provider: SUN

   Your keystore contains 3 entries

   Alias name: schoolkeysigned
   Creation date: Oct 18, 2010
   Entry type: PrivateKeyEntry
   Certificate chain length: 1
   Certificate[1]:
   Owner: CN=Lab, OU=Seidenberg School, O=Pace University, L=New York, ST=NY, C=US
   Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
   Serial number: 4cbd3110
   Valid from: Mon Oct 18 22:48:00 PDT 2010 until: Tue Oct 18 22:48:00 PDT 2011
   Certificate fingerprints:
        MD5:  CD:FD:F7:48:6E:A1:95:DD:87:76:02:5F:08:DA:3B:8B
        SHA1: F5:6E:E4:FA:E1:C6:62:9A:0F:E8:43:C1:17:43:98:A2:18:E2:B3:B8
        Signature algorithm name: MD5withRSA
        Version: 3


   *******************************************
   *******************************************


   Alias name: ca
   Creation date: Oct 18, 2010
   Entry type: PrivateKeyEntry
   Certificate chain length: 1
   Certificate[1]:
   Owner: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY, C=US
   Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
   Serial number: 4cbd2e03
   Valid from: Mon Oct 18 22:34:59 PDT 2010 until: Tue Oct 18 22:34:59 PDT 2011
   Certificate fingerprints:
        MD5:  0B:B7:B1:48:36:30:FE:DB:E3:EA:35:17:6C:02:9C:C0
        SHA1: BF:E3:0C:C1:1B:4C:36:BD:41:81:24:C4:7A:E2:2A:24:F6:4A:89:B9
```

```
        Signature algorithm name: SHA1withRSA
        Version: 3


   ***************************************
   ***************************************



   Alias name: schoolkey
   Creation date: Oct 18, 2010
   Entry type: PrivateKeyEntry
   Certificate chain length: 2
   Certificate[1]:
   Owner: CN=Lab, OU=Seidenberg School, O=Pace University, L=New York, ST=NY, C=US
   Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
   Serial number: 4cbd3110
   Valid from: Mon Oct 18 22:48:00 PDT 2010 until: Tue Oct 18 22:48:00 PDT 2011
   Certificate fingerprints:
        MD5:  CD:FD:F7:48:6E:A1:95:DD:87:76:02:5F:08:DA:3B:8B
        SHA1: F5:6E:E4:FA:E1:C6:62:9A:0F:E8:43:C1:17:43:98:A2:18:E2:B3:B8
        Signature algorithm name: MD5withRSA
        Version: 3
   Certificate[2]:
   Owner: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY, C=US
   Issuer: CN=Certificate Authority, OU=ITS, O=Pace University, L=New York, ST=NY,
C=US
   Serial number: 4cbd2e03
   Valid from: Mon Oct 18 22:34:59 PDT 2010 until: Tue Oct 18 22:34:59 PDT 2011
   Certificate fingerprints:
        MD5:  0B:B7:B1:48:36:30:FE:DB:E3:EA:35:17:6C:02:9C:C0
        SHA1: BF:E3:0C:C1:1B:4C:36:BD:41:81:24:C4:7A:E2:2A:24:F6:4A:89:B9
        Signature algorithm name: SHA1withRSA
        Version: 3


   ***************************************
   ***************************************



   user@ubuntu:~/JavaSecurityLab/cert-chain$
```

You can see that there are two certificates associated with alias “SchoolKey”. The first one is for the CA’s private key to sign the “SchoolKey” certificate. The second one is for the CA private key to sign its own certificate. This is your chain of certificates.

14. Now run “jarsigner -keystore keystore -signedjar signedWriteFile.jar WriteFile.jar CA” to sign the Jar file with the private key of the CA. The signed Jar file is “signedWriteFile.jar”. Upon request enter 123456 for keystore password.
15. Double click file “appleteWrite2.html” in file explorer and choose to display it. You will be asked whether you trust the applet signed by CA.




16. Check “Always trust content from this publisher” and click the “Run” button, and the applet will succeed in writing data to file “~/data.txt”.


17. Run “jarsigner -keystore keystore -signedjar signedWriteFile.jar WriteFile.jar SchoolKey” to resign the Jar file with the private key associated with “SchoolKey”. Upon request enter 123456 for keystore password.
18. Double click file “appleteWrite2.html” in file explorer and choose to display it. You will NOT be asked whether you trust the applet signed by SchoolKey, and the applet will succeed in writing data to file “~/data.txt”. This is because you already told the computer that you always trust code signed by the CA. Since the certificate for “SchoolKey” is signed by the CA, your trust to the CA is transfer to your trust to the certificate for “SchoolKey”.

This lab provides clear idea and hands-on knowledge on how a CA signs your certificates, and how a certificate chain implements a trust chain. While your chain has only two certificates, it could be extended to any length.

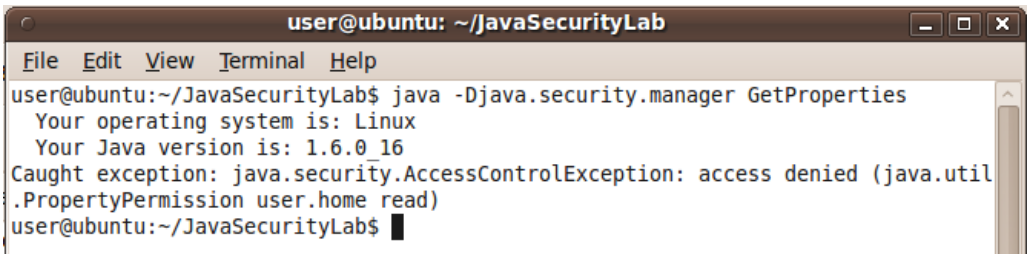
### *9.3.8 Protecting Your Computer from Insecure Java Applications*

Sometimes you need to run an application but you don't have full trust on it. You need to find out what are the minimum resources that this application needs to access and in which form to access (read or write or both). Then you can run the application under control of the Java security manager to limit its access to most resources on your system, and explicitly grant it access to those necessary resources to obtain the desired service from the application with reduced danger to your computer. In this lab you will use the Java program "GetProperties.java" as the insecure Java application.

1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab" to change the working folder to "~/JavaSecurityLab".
4. Run "gedit GetProperties.java" to review the source code of file "GetProperties.java".
5. Run "javac GetProperties.java" to compile the source code into bytecode file "GetProperties.class".
6. Run "java GetProperties" to execute file "GetProperties.class" without the control of a Java security manager. You will see the following printout:

```
Your operating system is: Linux
Your Java version is: 1.6.0_16
Your user home directory is: /home/user
Your JRE installation directory is: /home/user/jdk1.6.0_25/jre
Your Java extension directories are:
  /home/user/jdk1.6.0_25/jre/lib/ext:/usr/java/packages/lib/ext
```

7. Run "java -Djava.security.manager GetProperties" to execute file "GetProperties.class" under the control of a Java security manager. You will see the following printout:

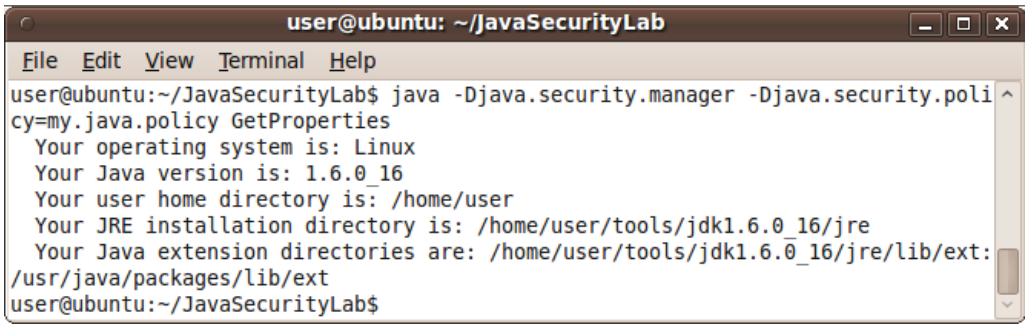



   Now the class runs as an applet and it cannot access Java property "user.home".
8. Run "more my.java.policy" to review the contents of file "my.java.policy":

```
grant codeBase "file:///home/user/JavaSecurityLab/" {
  permission java.util.PropertyPermission "user.home", "read";
  permission java.util.PropertyPermission "java.home", "read";
  permission java.util.PropertyPermission "java.ext.dirs", "read";
};
```

   While you normally specify special access rights of applications in the default user's Java policy file "~/.java.policy", as you have done so far, you could also save part of the policies in a separate file and later apply the policies in this file by specifying this separate file as a Java command line argument, as demonstrated in the next step.
9. Run the following command to run the Java application "GetProperties" under the control of Java security manager, and grant the access rights to the application as specified in the Java policy file "my.java.policy":

```
java -Djava.security.manager -Djava.security.policy=my.java.policy GetProperties
```




As you can see, this time the application runs successfully. You could also copy the contents of file "my.java.policy" into "~/.java.policy" so you don't need to specify a Java policy file on Java command line.

This concludes the lab for explaining how to run a Java application under the control of a Java security manager so it runs in (almost) isolation, and use a Java policy file to selectively grant rights to the application for it to deliver its services.

**Question 28:** Why a user should bother to run a Java application using the Java security manager?

**Question 29:** How to run a Java application under the supervision of the Java security manager?

**Question 30:** What is the recommended way to study or develop multiple Java programs using different and maybe conflicting security policies?

### *9.3.9 Securing File Exchange with Java Security API and Newly Created Keys*

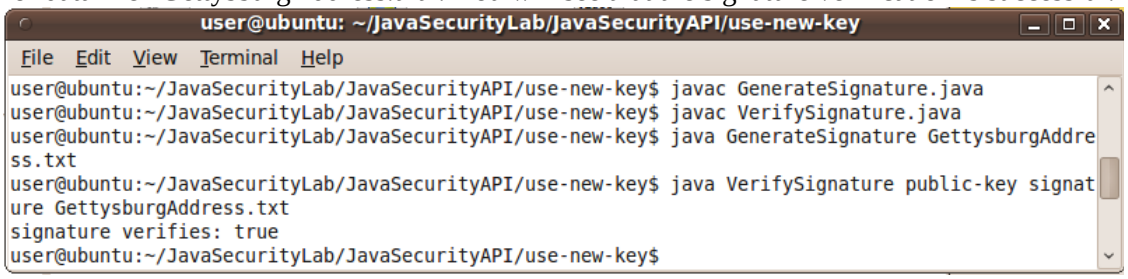
Java has a security API that allows a Java program to carry out some of the tasks normally conducted by Java security utilities. These tasks include generating public/private key pairs, exporting and importing these keys to/from files or keystores, and generating digital certificates and signatures. The coming three labs guide you to explore these functions against the background of secure file exchange against identity attacks or contents modification. If you also need to maintain the confidentiality of the file contents, you can use one of the Java ciphers of the Java Cryptography Extension/Artitecture (JCE, http://download.oracle.com/javase/1.4.2/docs/guide/security/CryptoSpec.html, http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html) to encode the file before its signing and decode the file after it has been verified for the correct signature. These three labs differ in where the private key comes from. In the current lab, the private key is dynamically created. In the following labs, the private key is retrieved from a file or from a keystore.

The objective of these three labs is to give you the first impression and main idea of the Java security API and how does it work. You should not try to understand each line of the code unless you have taken two Java courses.

In this lab, you need to securely send a data file, "GettysburgAddress.txt", to another person. You will first use a Java program to create a pair of public/private keys, generate the digital signature of the data file, and export the public key and the signature into files. The receiver of the data file will use the second

Java program to read in the public key, the signature and the data file to verify whether the signature is correct.

1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab/JavaSecurityAPI/use-new-key" to change working folder to "~/JavaSecurityLab/JavaSecurityAPI/use-new-key".
4. Run "gedit GenerateSignature.java &" to review the contents of Java source code file "GenerateSignature.java".
5. Run "javac GenerateSignature.java" to compile the file into bytecode file "GenerateSignature.class".
6. Run "gedit VerifySignature.java &" to review its contents of Java source code file "VerifySignature.java".
7. Run "javac VerifySignature.java" to compile the file into bytecode file "VerifySignature.class".
8. Run "java GenerateSignature GettysburgAddress.txt" to sign data file "GettysburgAddress.txt", and generate the public key file "public-key" and signature file "signature".
9. Run "java VerifySignature public-key signature GettysburgAddress.txt" to verify the signature for data file "GettysburgAddress.txt". You will see that the signature verification is successful.




Now we list important segments of the programs and explain them.

#### 9.3.9.1 Java command-line arguments

```
class GenerateSignature {
   public static void main(String[] args) … { …… }
}
```

When a Java program runs, it first runs the main() method. The main method has an array of strings, args[…], as its parameter for holding command-line arguments. When you run "java Prog a b c", args[0] will hold string "a", args[1] will hold string "b", and args[2] will hold string "c".

#### 9.3.9.2 Generating public/private keys

```
1. KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
2. SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
3. keyGen.initialize(1024, random);
4. KeyPair pair = keyGen.generateKeyPair();
5. PrivateKey priv = pair.getPrivate();
6. PublicKey pub = pair.getPublic();
7. byte[] key = pub.getEncoded();
8. FileOutputStream keyfos = new FileOutputStream("public-key");
9. keyfos.write(key);
10. keyfos.close();
```

1. This line creates a key generator object, sets its key generator algorithm to be DSA (digital signature algorithm) implemented by SUN (now Oracle).
2. This line creates a secure random number generator object, and sets its random number algorithm to be SHA1PRNG implemented by SUN. The key generator needs random numbers to generate random keys.
3. This line specifies that the key generator will use the random generator created on line 2 and generate keys of length 1024 bytes.
4. This line generates the key pair.
5. This line retrieves the private key.
6. This line retrieves the public key.
7. Lines 7-10 are for writing the public key in file “public-key”. Line 7 saves the key in an array of bytes encoded for input/output.
8. This line opens file “public-key” for output.
9. This line writes the key bytes out into the file.
10. This line closes the file to reserve system resources.

#### 9.3.9.3 Generating a digital signature

```
1. Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
2. dsa.initSign(priv);
3. FileInputStream fis = new FileInputStream(args[0]);
4. BufferedInputStream bufin = new BufferedInputStream(fis);
5. byte[] buffer = new byte[1024];
6. while (bufin.available() != 0) {
7.   int len = bufin.read(buffer);
8.   dsa.update(buffer, 0, len);
9. };
10. bufin.close();
11. byte[] realSig = dsa.sign();
12. FileOutputStream sigfos = new FileOutputStream("signature");
13. sigfos.write(realSig);
14. sigfos.close();
```

1. This line creates a signature generator object, sets its signing algorithm to be SHA1 (a type of hash function for generating fingerprints) with DSA (digital signature algorithm) implemented by SUN (now Oracle).
2. This line specifies the private key for the signature generator.
3. This line opens the file whose name is specified by the first command-line argument (the first string after the class name that you run).
4. This line provides the opened file with a data buffer so you could know whether you still have data to read on line 6 before you actually read the data on line 7.
5. This line creates an array for 1024 bytes as a data buffer. Since you use a private key of 1024 bytes long, you will process 1024 bytes a time to generate the fingerprint.
6. Lines 6-9 are a loop. Code “bufin.available()” returns number of bytes in the buffer that have not been read yet. Code on lines 7 and 8 will be executed repeatedly until the buffer is empty.
7. This line reads up to 1024 bytes into the 1024-byte buffer (the last read may have less than 1024 bytes). The actual number of read bytes is saved in variable “len”.
8. This line applies the DSA algorithm on the newly read data.
9. This line marks the end of the loop.
10. This line closes the file to release the file buffer.
11. This line actually generates the signature of the input data and save it in an array of bytes.
12. Lines 12-14 are for writing the signature out in a file named “signature”.

### *9.3.10 Securing File Exchange with Java Security API and Keys in Files*

Typically you generate a pair of public/private key pairs and use it for multiple times. For example you may generate a pair of keys to sign all of your documents, and another pair to sign your programs. You can either save the key pairs in files or a keystore. In this lab you will save the key pair in files for reuse.

1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab/JavaSecurityAPI/key-in-file" to change working folder to "~/JavaSecurityLab/JavaSecurityAPI/key-in-file".
4. Run "gedit GenerateKeys.java &" to review the contents of Java source code file "GenerateKeys.java". This program generates a pair of public/private keys, and save them in files "public-key" and "private-key" for reuse.
5. Run "javac GenerateKeys.java" to compile the file into bytecode file "GenerateKeys.class".
6. Run "gedit GenerateSignature2.java &" to review the contents of Java source code file "GenerateSignature2.java". This program import the private key from file "private-key", use it to sign the data file, and write the resulting signature in file "signature".
7. Run "javac GenerateSignature2.java" to compile the file into bytecode file "GenerateSignature2.class".
8. Run "gedit VerifySignature.java &" to review the contents of Java source code file "VerifySignature.java". This is the same file that you used in the last two labs.
9. Run "javac VerifySignature.java" to compile the file into bytecode file "VerifySignature.class".
10. Run "java GenerateKeys" to generate the key pair in files "public-key" and "private-key".
10. Run "java GenerateSignature2 private-key GettysburgAddress.txt" to sign data file "GettysburgAddress.txt" with the private key in file "private-key", and save the generated signature in file "signature".
11. Run "java VerifySignature public-key signature GettysburgAddress.txt" to verify the signature for data file "GettysburgAddress.txt". You will see that the signature verification is successful.


#### 9.3.10.1 Importing a private key from a file

```
1. FileInputStream keyfis = new FileInputStream(args[0]);
2. byte[] encKey = new byte[keyfis.available()];
3. keyfis.read(encKey);
4. keyfis.close();
5. PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(encKey);
6. KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
7. PrivateKey priv = keyFactory.generatePrivate(privKeySpec);
```

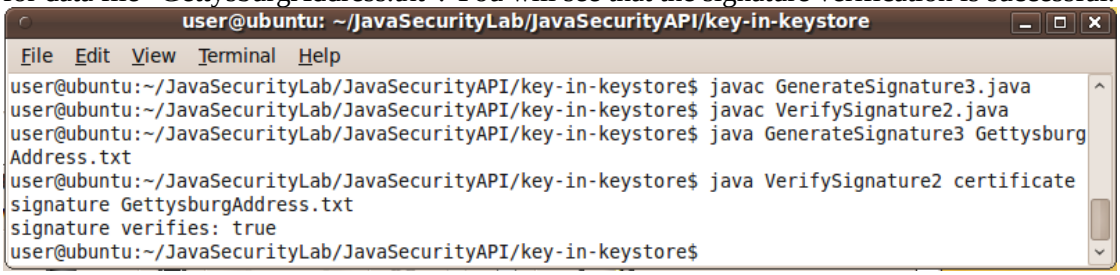
1. Line 1 opens the private key file whose name is specified by the first command-line argument to "java" (the first string after the class name).

2. Line 2 checks the data size in the file, and creates an array of bytes of the same size.
3. Line 3 reads the contents of the file into the array of bytes.
4. Line 4 closes the file for recycling the resources associated with the file.
5. Line 5 generates a PKCS8 encoding of the private key. When you write keys to files, the public keys are encoded in the X509 format, and the private keys are encoded in the PKCS8 format.
6. Line 6 creates a key factory object using the DSA algorithm implemented by SUN.
7. Line 7 creates the private key based on the PKCS8 specification of the key.

### *9.3.11 Securing File Exchange with Java Security API and Keys in a Keystore*

In this lab you will repeat what you did in the last two labs but use the digital certificate and private key, associated with alias "PaceKey", in PaceKeystore. You created this keystore and the key/certificate in an earlier lab.

1. Launch the *Ubuntu* VM with username "user" and password 123456.
2. Start a terminal window in home folder ~ with menu item "Applications|Accessories|Terminal".
3. Run "cd ~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore" to change work folder to "~/JavaSecurityLab/JavaSecurityAPI/key-in-keystore".
4. Run "gedit GenerateSignature3.java &" to review the contents of Java source code file "GenerateSignature3.java".
5. Run "javac GenerateSignature3.java" to compile the file into bytecode file "GenerateSignature3.class".
6. Run "gedit VerifySignature2.java &" to review the contents of Java source code file "VerifySignature2.java".
7. Run "javac VerifySignature2.java" to compile the file into bytecode file "VerifySignature2.class".
8. Run "java GenerateSignature3 GettysburgAddress.txt" to sign data file "GettysburgAddress.txt" with the private key with alias "PaceKey" in keystore "PaceKeystore", write the signature in file "signature", and write the certificate in file "certificate".
9. Run "java VerifySignature2 certificate signature GettysburgAddress.txt" to verify the signature for data file "GettysburgAddress.txt". You will see that the signature verification is successful.




#### 9.3.11.1 Importing a private key and certificate pair from a keystore

```
1. KeyStore ks = KeyStore.getInstance("JKS");
2. FileInputStream ksfis = new FileInputStream("../../PaceKeystore");
3. BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
4. ks.load(ksbufin, "PaceUniversity".toCharArray());
5. PrivateKey priv = (PrivateKey)ks.getKey("PaceKey",
     "Seidenberg".toCharArray());
6. java.security.cert.Certificate cert = ks.getCertificate("PaceKey");
7. byte[] encodedCert = cert.getEncoded();
8. FileOutputStream certfos = new FileOutputStream("certificate");
9. certfos.write(encodedCert);
```

```
10. certfos.close();
```

1. Line 1 creates a keystore object of type JKS (Java Keystore). The keystores for Java have nothing to do with the keystores for PGP.
2. Line 2 opens the keystore file “PaceKeystore” two levels up in the file system.
3. Line 3 provides the file with a data buffer.
4. Line 4 loads the contents of the keystore file into the in-memory keystore object with the keystore password “PaceUniversity”. Since the load() method needs an array of characters for the password, we first converted the string “PaceUniversity” into an array of characters.
5. Line 5 retrieves the private key associated with alias “PaceKey”; and it provides the keystore object with the password “Seidenberg” for the key.
6. Line 6 retrieves the certificate in the keystore object associated with alias “PaceKey”. You may note that you don’t need to provide a key password to retrieve a certificate because a certificate is public.
7. Lines 7-10 write the certificate out into file “certificate”. Line 7 gets a copy of the certificate encoded for input/output.
8. Line 8 opens file “certificate” for output.
9. Line 9 writes the encoded certificate in file “certificate”.
10. Line 10 closes the file for recycling the resources associated with the file.

#### 9.3.11.2 Importing a public key from a digital certificate file

```
1. FileInputStream certfis = new FileInputStream(args[0]);
2. java.security.cert.CertificateFactory cf =
     java.security.cert.CertificateFactory.getInstance("X.509");
3. java.security.cert.Certificate cert = cf.generateCertificate(certfis);
4. PublicKey pubKey = cert.getPublicKey();
```

1. Line 1 opens the digital certificate’s file specified as the first command-line argument to command “java” (the first string after the class name).
2. Line 2 creates a certificate factory object of type “X.509”.
3. Line 3 uses the certificate factory object to generate a certificate object based on the contents of the certificate file.
4. Line 4 retrieves the public key from the certificate object.

## 9.4 Review Questions

**Question 31:** Why Java security is important?

**Question 32:** What are the most vulnerable folders for Java security?

**Question 33:** Why applets always run using Java security manager?

**Question 34:** List three resources that programs using Java security manager cannot access by default?

**Question 35:** Why you should bother to run some applications using Java security manager?

**Question 36:** How to selectively grant access rights to applets or applications?

## 9.5 Appendix

### *9.5.1 File "GetProperties.java"*

```
class GetProperties {
  public static void main(String[] args) {
    String s;
    try {
      s = System.getProperty("os.name", "not specified");
      System.out.println("  Your operating system is: " + s);
      s = System.getProperty("java.version", "not specified");
      System.out.println("  Your Java version is: " + s);
      s = System.getProperty("user.home", "not specified");
      System.out.println("  Your user home directory is: " + s);
      s = System.getProperty("java.home", "not specified");
      System.out.println("  Your JRE installation directory is: " + s);
      s = System.getProperty("java.ext.dirs", "not specified");
      System.out.println("  Your Java extension directories are: " + s);
    } catch (Exception e) {
      System.err.println("Caught exception: " + e);
    }
  }
}
```

### *9.5.2 File "WriteFile.java"*

```
import java.awt.*;
import java.applet.*;
import java.io.*;

public class WriteFile extends Applet {
  public void paint(Graphics g) {
    try {
      String fileName = System.getProperty("user.home") +
              System.getProperty("file.separator") +     //   / or \
              "data.txt";
      File f = new File(fileName);
      PrintWriter output = new PrintWriter(new FileWriter(f), true); // auto-flush
      output.println(new java.util.Date() + ": Pace University Java Tutorial");
      g.drawString("Data were successfully written to file \"" + fileName + "\"",
                   10, 10);
    }
    catch (SecurityException e) {
      g.drawString("WriteFile: security exception is caught", 10, 10);
      e.printStackTrace(); // print error messages to terminal window for debugging
    }
    catch (IOException ioe) {
      g.drawString("WriteFile: I/O exception is caught", 10, 10);
    }
  }
}
```

### *9.5.3 File "appletWrite.html"*

```
<html>
<head>
<title>Test Java Applet File Writing</title>
</head>
<body>
<p>The following is a Java applet trying to write the current time
and "Pace University Java Tutorial" in file "data.txt" of your home
folder.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100>
</applet>
</body>
</html>
```

### *9.5.4 File "appletWrite2.html"*

```
<html>
<head>
<title>Test Java Applet File Writing (Jar File)</title>
</head>
<body>
<p>The following is a Java applet, in the form of a Java JAR file, trying to write
current time and "Pace University Java Tutorial" in file "data.txt" of your home
directory.</p>
<br/>
<applet code="WriteFile.class" width=420 height=100  archive="signedWriteFile.jar">
</applet>
</body>
</html>
```

### *9.5.5 File "my.java.policy"*

```
grant codeBase "file:///home/user/JavaSecurityLab/" {
  permission java.util.PropertyPermission "user.home", "read";
  permission java.util.PropertyPermission "java.home", "read";
  permission java.util.PropertyPermission "java.ext.dirs", "read";
};
```

### *9.5.6 File "GenerateSignature.java"*

```
import java.io.*;
import java.security.*;

// Generate a pair of public/private keys, generate signature of the input data
// with the private key, write out the signature and the public key in files
// "signature" and "public-key".
class GenerateSignature {
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.out.println("Usage: java GenerateSignature fileToSign");
      System.exit(-1);
    }
    // Generate a key pair
```

```
      // set key generator algorithm DSA implemented by SUN
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
      // set random number algorithm SHA1PRNG implemented by SUN
      SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
      keyGen.initialize(1024, random); // set key length be 1024 bytes
      KeyPair pair = keyGen.generateKeyPair(); // generate key pair
      PrivateKey priv = pair.getPrivate();
      PublicKey pub = pair.getPublic();
      // Create a Signature object and initialize it with algorithm SHA1withDSA
      // implemented by SUN.
      Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
      dsa.initSign(priv); // set the private key
      // Read in data, up to 1024 bytes a time, to generate fingerprint
      // args[0]: first command-line argument, data file name
      FileInputStream fis = new FileInputStream(args[0]); // open data file
      // provide buffer for data input
      BufferedInputStream bufin = new BufferedInputStream(fis);
      byte[] buffer = new byte[1024];
      while (bufin.available() != 0) { // while there are still unread data
        // read up to 1024 bytes, set actual length in len
        int len = bufin.read(buffer);
        dsa.update(buffer, 0, len);    // apply hash to the recent data segment
      };
      bufin.close();
      // Generate a signature by encoding the fingerprint with the private key
      byte[] realSig = dsa.sign();
      // Save the signature in file "signature"
      FileOutputStream sigfos = new FileOutputStream("signature");
      sigfos.write(realSig);
      sigfos.close();
      // Save the public key in file "public-key"
      byte[] key = pub.getEncoded();
      FileOutputStream keyfos = new FileOutputStream("public-key");
      keyfos.write(key);
      keyfos.close();
    };
  }
```

### *9.5.7 File "VerifySignature.java"*

```
  import java.io.*;
  import java.security.*;
  import java.security.spec.*;

  // Import public key and signature from the files, and use them to authenticate
  // the author and validate the contents of the data file
  class VerifySignature {
    public static void main(String[] args) throws Exception {
      if (args.length != 3) {
        System.out.println("Usage: java VerifySignature publicKeyFile" +
          " signatureFile dataFile");
        System.exit(-1);
      }
      // import encoded public key
      FileInputStream keyfis = new FileInputStream(args[0]);
      // create a byte array as large as the file contents
      byte[] encKey = new byte[keyfis.available()];
      keyfis.read(encKey);
      keyfis.close();
      // use file contents to create a public key specification object
      X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);
```

```
      // create a key factory using algorithm DSA implemented by SUN
      KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
      // recover public key from its specification
      PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
      // input the signature bytes
      FileInputStream sigfis = new FileInputStream(args[1]);
      byte[] sigToVerify = new byte[sigfis.available()];
      sigfis.read(sigToVerify );
      sigfis.close();
      // create a Signature object using algorithm SHA1withDSA implemented by SUN
      Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
      // initialize the Signature object with the public key
      sig.initVerify(pubKey);
      // Read in data, up to 1024 bytes a time, to generate fingerprint
      // args[2]: third command-line argument, data file name
      FileInputStream datafis = new FileInputStream(args[2]); // open data file
      // provide buffer for data input
      BufferedInputStream bufin = new BufferedInputStream(datafis);
      byte[] buffer = new byte[1024];
      int len;
      while (bufin.available() != 0) { // while there are still unread data
        // read up to 1024 bytes, set actual length in len
        len = bufin.read(buffer);
        sig.update(buffer, 0, len);    // apply hash to the recent data segment
      };
      bufin.close();
      // decode the input signature into fingerprint with the public key,
      // compare it with the fingerprint just created above.
      boolean verifies = sig.verify(sigToVerify);
      System.out.println("signature verifies: " + verifies);
    }
  }
```

### *9.5.8 File “GenerateKeys.java”*

```
  import java.io.*;
  import java.security.*;

  // Generate a pair of public/private keys, and save them in files "public-key" and
  // "private-key".
  class GenerateKeys {
    public static void main(String[] args) throws Exception {
      // Generate a key pair
      // set key generator algorithm DSA implemented by SUN
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
      // set random number algorithm SHA1PRNG implemented by SUN
      SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
      keyGen.initialize(1024, random); // set key length be 1024 bytes
      KeyPair pair = keyGen.generateKeyPair(); // generate key pair
      PrivateKey priv = pair.getPrivate();
      PublicKey pub = pair.getPublic();
      // Save the public key in file "public-key"
      byte[] key = pub.getEncoded();
      FileOutputStream keyfos = new FileOutputStream("public-key");
      keyfos.write(key);
      keyfos.close();
      // Save the private key in file "private-key"
      key = priv.getEncoded();
      keyfos = new FileOutputStream("private-key");
      keyfos.write(key);
```

```
      keyfos.close();
    };
  }
```

### *9.5.9 File "GenerateSignature2.java"*

```
  import java.io.*;
  import java.security.*;
  import java.security.spec.*;

  // Import a private key from a file, generate signature of the input data with the
  // private key, and write out the signature in file "signature".
  class GenerateSignature2 {
    public static void main(String[] args) throws Exception {
      if (args.length != 2) {
        System.out.println("Usage: java GenerateSignature2 privateKeyFile" +
                           " fileToSign");
        System.exit(-1);
      }
      // Import the encoded private key from file args[0]
      FileInputStream keyfis = new FileInputStream(args[0]);
      // create a byte array as large as the file contents
      byte[] encKey = new byte[keyfis.available()];
      keyfis.read(encKey);
      keyfis.close();
      // use file contents to create a private key specification object
      PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(encKey);
      // create a key factory using algorithm DSA implemented by SUN
      KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
      // recover private key from its specification
      PrivateKey priv = keyFactory.generatePrivate(privKeySpec);
      // Create a Signature object and initialize it with algorithm SHA1withDSA
      // implemented by SUN.
      Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
      dsa.initSign(priv); // set the private key
      // Read in data, up to 1024 bytes a time, to generate fingerprint
      // args[1]: second command-line argument, data file name
      FileInputStream fis = new FileInputStream(args[1]); // open data file
      // provide buffer for data input
      BufferedInputStream bufin = new BufferedInputStream(fis);
      byte[] buffer = new byte[1024];
      int len;
      while (bufin.available() != 0) { // while there are still unread data
        // read up to 1024 bytes, set actual length in len
        len = bufin.read(buffer);
        dsa.update(buffer, 0, len);    // apply hash to the recent data segment
      };
      bufin.close();
      // Generate a signature by encoding the fingerprint with the private key
      byte[] realSig = dsa.sign();
      // Save the signature in a file "signature"
      FileOutputStream sigfos = new FileOutputStream("signature");
      sigfos.write(realSig);
      sigfos.close();
    };
  }
```

### *9.5.10 File “GenerateSignature2.java”*

```
import java.io.*;
import java.security.*;
import java.security.spec.*;

// Import a private key from a file, generate signature of the input data with the
// private key, and write out the signature in file "signature".
class GenerateSignature2 {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.println("Usage: java GenerateSignature2 privateKeyFile " +
                         " fileToSign");
      System.exit(-1);
    }
    // Import the encoded private key from file args[0]
    FileInputStream keyfis = new FileInputStream(args[0]);
    // create a byte array as large as the file contents
    byte[] encKey = new byte[keyfis.available()];
    keyfis.read(encKey);
    keyfis.close();
    // use file contents to create a private key specification object
    PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(encKey);
    // create a key factory using algorithm DSA implemented by SUN
    KeyFactory keyFactory = KeyFactory.getInstance("DSA", "SUN");
    // recover private key from its specification
    PrivateKey priv = keyFactory.generatePrivate(privKeySpec);
    // Create a Signature object and initialize it with algorithm SHA1withDSA
    // implemented by SUN.
    Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
    dsa.initSign(priv); // set the private key
    // Read in data, up to 1024 bytes a time, to generate fingerprint
    // args[1]: second command-line argument, data file name
    FileInputStream fis = new FileInputStream(args[1]); // open data file
    // provide buffer for data input
    BufferedInputStream bufin = new BufferedInputStream(fis);
    byte[] buffer = new byte[1024];
    int len;
    while (bufin.available() != 0) { // while there are still unread data
      // read up to 1024 bytes, set actual length in len
      len = bufin.read(buffer);
      dsa.update(buffer, 0, len);    // apply hash to the recent data segment
    };
    bufin.close();
    // Generate a signature by encoding the fingerprint with the private key
    byte[] realSig = dsa.sign();
    // Save the signature in file "signature"
    FileOutputStream sigfos = new FileOutputStream("signature");
    sigfos.write(realSig);
    sigfos.close();
  };
}
```

### *9.5.11 File “GenerateSignature3.java”*

```
import java.io.*;
import java.security.*;
import java.security.spec.*;

// Import a private key from a keystore, generate signature of the input data with
// the private key, and write out the signature in file "signature".
class GenerateSignature3 {
```

```
    public static void main(String[] args) throws Exception {
      if (args.length != 1) {
        System.out.println("Usage: java GenerateSignature3 fileToSign");
        System.exit(-1);
      }
      // Import the private key and certificate with alias "PaceKey" from keystore
      // "../../PaceKeystore"
      KeyStore ks = KeyStore.getInstance("JKS"); // get an empty keystore object
      // open keystore file
      FileInputStream ksfis = new FileInputStream("../../PaceKeystore");
      // load keystore contents in buffer
      BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
      // load keystore contents into the keystore object; supply keystore password
      ks.load(ksbufin, "PaceUniversity".toCharArray());
      // retrieve private key for alias "PaceKey" with key password "Seidenberg"
      PrivateKey priv = (PrivateKey)ks.getKey("PaceKey", "Seidenberg".toCharArray());
      // Retreive the corresponding certificate and save it in a file
      java.security.cert.Certificate cert = ks.getCertificate("PaceKey");
      byte[] encodedCert = cert.getEncoded(); // encode the certificate for I/O
      // save the certificate in a file named "certificate" */
      FileOutputStream certfos = new FileOutputStream("certificate");
      certfos.write(encodedCert);
      certfos.close();
      // Create a Signature object and initialize it with algorithm SHA1withDSA
      // implemented by SUN.
      Signature dsa = Signature.getInstance("SHA1withDSA", "SUN");
      dsa.initSign(priv); // set the private key
      // Read in data, up to 1024 bytes a time, to generate fingerprint
      // args[0]: first command-line argument, data file name
      FileInputStream fis = new FileInputStream(args[0]); // open data file
      // provide buffer for data input
      BufferedInputStream bufin = new BufferedInputStream(fis);
      byte[] buffer = new byte[1024];
      int len;
      while (bufin.available() != 0) { // while there are still unread data
        // read up to 1024 bytes, set actual length in len
        len = bufin.read(buffer);
        dsa.update(buffer, 0, len);    // apply hash to the recent data segment
      };
      bufin.close();
      // Generate a signature by encoding the fingerprint with the private key
      byte[] realSig = dsa.sign();
      // Save the signature in file "signature"
      FileOutputStream sigfos = new FileOutputStream("signature");
      sigfos.write(realSig);
      sigfos.close();
    };
  }
```

### *9.5.12 File "VerifySignature2.java"*

```
  import java.io.*;
  import java.security.*;
  import java.security.spec.*;

  // Import certificate and signature from the files, get public key from
  // certificate, and use them to authenticate the author and validate the contents
  // of the data file
  class VerifySignature2 {
    public static void main(String[] args) throws Exception {
```

```
      if (args.length != 3) {
        System.out.println("Usage: java VerifySignature2 certificateFile" +
                           " signatureFile dataFile");
        System.exit(-1);
      }
      // Import a digital certificate from args[0]
      FileInputStream certfis = new FileInputStream(args[0]);
      java.security.cert.CertificateFactory cf =
         java.security.cert.CertificateFactory.getInstance("X.509");
      java.security.cert.Certificate cert = cf.generateCertificate(certfis);
      // retrieve public key from the certificate
      PublicKey pubKey = cert.getPublicKey();
      // input the signature bytes
      FileInputStream sigfis = new FileInputStream(args[1]);
      byte[] sigToVerify = new byte[sigfis.available()];
      sigfis.read(sigToVerify );
      sigfis.close();
      // create a Signature object using algorithm SHA1withDSA implemented by SUN
      Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
      // initialize the Signature object with the public key
      sig.initVerify(pubKey);
      // Read in data, up to 1024 bytes a time, to generate fingerprint
      // args[2]: third command-line argument, data file name
      FileInputStream datafis = new FileInputStream(args[2]); // open data file
      // provide buffer for data input
      BufferedInputStream bufin = new BufferedInputStream(datafis);
      byte[] buffer = new byte[1024];
      int len;
      while (bufin.available() != 0) { // while there are still unread data
        // read up to 1024 bytes, set actual length in len
        len = bufin.read(buffer);
        sig.update(buffer, 0, len);    // apply hash to the recent data segment
      };
      bufin.close();
      // decode the input signature into fingerprint with the public key,
      // compare it with the fingerprint just created above.
      boolean verifies = sig.verify(sigToVerify);
      System.out.println("signature verifies: " + verifies);
    }
  }
```

# 10 Acknowledgement

Part of this material is based upon work supported by the *National Science Foundation*'s *Course Curriculum, and Laboratory Improvement (CCLI)* program under Grant No. 0837549. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the *National Science Foundation*.